FYS-STK4155 – Project 2[†]

Sajjad Ahmadigoltapeh[1], Jan Fredrik Kismul[2], and Silje Helene Løvbrekke[3]

[1]sajjadah@uio.no
[2]j.f.kismul@fys.uio.no
[3]siljehlo@matnat.uio.no

November 8, 2019

Abstract

In this study we use credit card data from Taiwan to compare how logistic regression and neural networks classify the data. The error of test and train data in neural network converges to 0.18 and 0.177 after 100 epochs. Logistic regression using Newton method is able to classify features for 15000 customers with an accuracy of approximately 0.816, while a neural network with 2 hidden layer and 10 neurons presents the accuracy of 0.823 within 100 epochs. Consequently, neural networks proved to be superior. Furthermore, it is realized that in this case it does not make a difference to use regularization parameters. Also we continue to study the Franke's function to compare neural networks used for regression to the methods from project 1, namely OLS, Ridge and Lasso. A neural network with 4 layers with [100,100,50,50] nodes in each layer, learning rate 0.01 and activations [relu6, leaky relu, relu6, leaky relu] for each layer achieved a R^2-score of 0.9905277 on the test set and an MSE of 0.0008016.

Contents

[†]Project folder: `https://github.com/jkismul/MachineLearning/tree/master/Project2`

1 Introduction

In many counties world wide there is an increasing rate of personal debt. Credit cards are easily acquired and most cardholders, irrespective of their repayment ability, overused credit cards for consumption and accumulated heavy credit and cash–card debts. The crisis caused the blow to consumer finance confidence and it is a big challenge for both banks and cardholders. As a result of this and advances in machine learning and data mining one can use data about customers for risk prediction to determine the customers ability to uphold and pay their debt [1]. We will use logistic regression and neural networks for this classification problem and analyse the data to determine whether a customer is a risky customer or not. Further we will exploit the flexibility of neural networks and use it for a regression problem as well as to see if we are able to find a better fit to the data set from project number one. We will use our results to compare with the results from Ordinary Least squares, Ridge and Lasso from project 1.

In this study, the main aim is comparison of a simple neural network system with the different linear regression methods for regression and logistic regression for classification. The regression part is includes finding a suitable value for weight and bias via minimizing the error and then predict the possibility for liquidating the loan.

In the first part of the report, necessary mathematical tools for regression and classification (including the back propagation algorithm for training neural network) are derived. Then, different methods for classification and regression are then applied to the credit card data are compared.

2 Theory and methods

2.1 Logistic regression

Logistic regression is used to solve problems, where each data points belongs to a specific category with relevant response of $y_i = 1$ or $y_i = 0$. Logistic regression is named after the logistic function (sigmoid function) which is used to calculate the probability of data point x:

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

In case we have a set of data points with several categories, the relevant response of $y_i = 1$ or $y_i = 0$ is calculated as bellow:

$$\hat{p}(y_i = 0|x_i, \hat{\beta}) = \frac{1}{1 + \exp\left(-\beta_j \phi_j(x_i)\right)}$$

$$\hat{p}(y_i = 1|x_i, \hat{\beta}) = 1 - \hat{p}(y_i = 0|x_i, \hat{\beta})$$

where $\phi_j(x_i) = X_{ij}$ is written same as in linear regression that gives:

$$\hat{p} = \frac{1}{1 + \exp(-X\hat{\beta})}$$

2.1.1 Cost function

Suppose, a given response of $y_i = 1$, therefore, p_i should give the likelihood as predicted by the logistic regressor. In addition, for $y_i = 0$, the statement of $1 - p_i$ presents the relevant likelihood. Therefore, the product the total likelihood is:

$$L = \prod_i l_i = \prod_i p_i^{y_i}(1-p_i)^{1-y_i}, \tag{1}$$

which should be maximised by the logistic regressor. Since the logarithm is monotonous and gives a convex function, it is possible to rewrite the previous equation as:

$$\log(L) = \sum_i y_i \log(p_i) + (1-y_i)\log(1-p_i), \tag{2}$$

Finally, with a minus sign the equation (2) is converted to minimization problem and the final cost function is written as following so-called the cross-entropy :

$$Q(\hat{\beta}) = -\sum_i (y_i \log(p_i) + (1-y_i)\log(1-p_i)), \tag{3}$$

2.1.2 Minimisation

The derivative of sigmoid function and cost function is:

$$\begin{aligned}\frac{d}{dx}(\frac{1}{1+e^{-x}}) &= \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}}(1-\frac{1}{1+e^{-x}})\end{aligned}$$

then, using the assumption of $t_i = \phi_j(x_i)\beta_j = X_{ij}\beta_j$ one can write the derivative of the cost function as:

$$\frac{\partial Q}{\partial \beta_k} = \frac{\partial Q}{\partial p_j}\frac{\partial p_j}{\partial \beta_k} = \frac{\partial Q}{\partial p_j}\frac{\partial p_j}{\partial t_j}\frac{\partial t_j}{\partial \beta_k} = \sum_j \frac{\partial Q}{\partial p_j}\frac{\partial p_j}{\partial t_j}\frac{\partial t_j}{\partial \beta_k} = \sum_j \frac{\partial Q}{\partial p_j} p_j(1-p_j)X_{jk}$$

we know,

$$\frac{\partial Q}{\partial p_j} = -\frac{y_j}{p_j} + \frac{1-y_j}{1-p_j} = \frac{-y_j(1-p_j)+p_j(1-y_j)}{p_j(1-p_j)} = \frac{p_j - y_j}{p_j(1-p_j)}$$

giving

$$\frac{\partial Q}{\partial \beta_k} = (p_j - y_j)X_{jk}$$

in matrix form it could be rewritten as:

$$\nabla_{\hat{\beta}} Q = X^T(\hat{p} - \hat{y}) \tag{4}$$

It is worth to note that there is no analytical solution for $\nabla Q = \hat{0}$, however the cost function i.e. equation (4) can be minimised with the standard minimisation methods.

2.1.3 Regularisation

To reduce the variance and avoid the overfitting, it is possible to regularize the cost function by adding a term such as λ. This technique so-called logistic regression regularization.

$$Q = -\sum_i (y_i \log(p_i) + (1 - y_i) \log(1 - p_i)) + \tfrac{1}{2}\lambda ||\hat{\beta}||_2^2 \tag{5}$$

which adds a simple term to the gradient of the cost function,

$$\nabla_{\hat{\beta}} Q = X^T(\hat{p} - \hat{y}) + \lambda \hat{\beta}. \tag{6}$$

2.2 Neural networks

Neural network (NN) is a popular technique in machine learning and its name refers to this fact that the idea of neural network is supposed to mimic a biological system of neurons. The neural network system includes some layers and each layer contains certain amount of neurons. Based on the neurons function in biological systems, a mathematical model can be developed for both classification and regression problem. Figure (1) shows the schematic drawing of a neural network with one hidden layer.

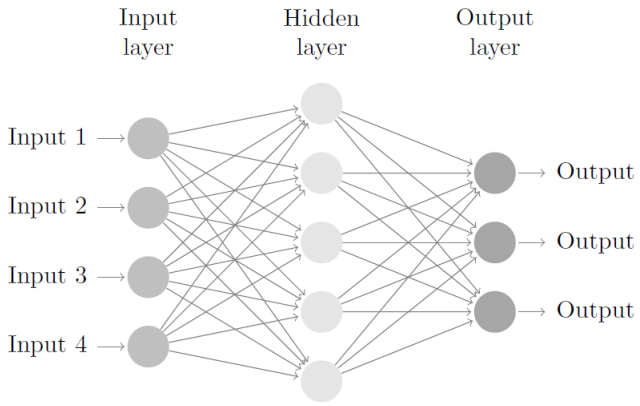

Figure 1: Schematic of a neural network which includes one hidden layer

Mathematically, neural network includes a complex set of transformations and activation functions that are utilized for data training and final prediction is done recursively. Each pair of one transformation and one activation function is called *layer*. The number of outputs from a layer is called the number of *neurons* inside the layer. Actually, the layer number l contains a weight-matrix W^l, a bias vector $\hat{b}^l$ and an activation function f^l that takes input of $\hat{a}^{l-1}$ then, applies the affine transformation $\hat{z}^l = W^l\hat{a}^{l-1} + \hat{b}^l$ and finally returns the output $f^l(\hat{z}^l)$. The input layer (first layer), applies its affine transformation to the input vector and then its activation function. The second layer receives the output of the input layer and repeats the same process. These chain type processes are repeated at each layer and finally reach to the final layer, which activation function is tailored to either classification or regression.

2.2.1 Back-propagation

The relevant cost function of neural network should be minimized by using some sort of gradient method. Therefore, it is required to calculate the derivative of the cost function with respect to the fitting parameters, namely the weight matrices W^l and the bias vectors $\hat{b}^l$. Since, some complicated math functions are applied to calculate the prediction, deriving a closed-form formula for derivatives of the cost function with respect to weights and biases is not possible. However, it is possible to find the derivative of the last layer with respect to the weights and biases, then derive a recursion relation that gives the derivatives of former layers. This method is known as back propagation algorithm. Derivative of the cost function with respect to the biases is derived as bellow:

$$\frac{\partial Q}{\partial b_j^l} = \frac{\partial Q}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_j^l} = \frac{\partial Q}{\partial z_k^l}\frac{\partial}{\partial b_j^l}(W_{ki}^l a_i^{l-1} + b_k^l) = \frac{\partial Q}{\partial z_k^l}\delta_{jk} = \frac{\partial Q}{\partial z_j^l} = \delta_j^l$$

then, δ_j^l is rewritten as:

$$\delta_j^l = \frac{\partial Q}{\partial z_j^l} = \frac{\partial Q}{\partial a_k^l}\frac{\partial a_k^l}{\partial z_j^l} = \frac{\partial Q}{\partial a_k^l}\frac{\partial f^l(z_k^l)}{\partial z_j^l} = \frac{\partial Q}{\partial a_j^l}\frac{\partial f^l}{\partial z_j^l}$$

furthermore, the derivative of cost function with respect to the weight matrix (W_{jk}^l) is derived as bellow:

$$\frac{\partial Q}{\partial W_{jk}^l} = \frac{\partial Q}{\partial z_i^l}\frac{\partial z_i^l}{\partial W_{jk}^l} = \delta_i^l\frac{\partial}{\partial W_{jk}^l}(W_{im}^l a_m^{l-1} + b_i^l) = \delta_j^l a_k^{l-1}$$

combination of derivative of cost function with respect to biases and weight matrix, gives the derivative of the cost function with respect to the weights and biases at the last layer. But, it is important to note that this does not give anything about the gradients at former layers, hence we need to calculate δ_j^l for $l < L$. Again by using the chain rule we have:

$$\delta_j^l = \frac{\partial Q}{\partial z_j^l} = \frac{\partial Q}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial a_i^l}\frac{a_i^l}{z_j^l} = \delta_k^{l+1}\frac{\partial}{\partial a_i^l}(W_{km}^{l+1}a_m^l + b_k^l)\frac{\partial f^l(z_i^l)}{\partial z_j^l} = \delta_k^{l+1}W_{kj}^{l+1}f^{l\prime}(z_j^l) \tag{7}$$

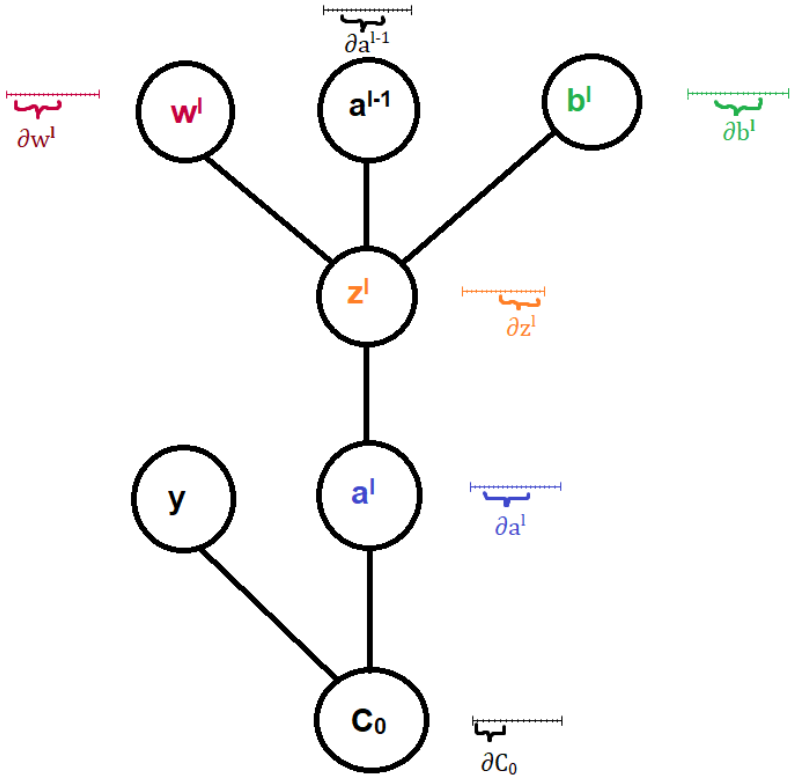

Figure 2: Illustration on how the cost function is affected by its partial derivatives.

The closed-form of the above results is written as:

$$\hat{\delta}^L = \nabla_{\hat{a}^L} Q \circ f^{L'}(\hat{z}^L), \tag{8}$$

where Q is cost function, f^L is final activation function and $\circ$ is an operand that denotes the element-wise product.

$$\hat{\delta}^l = (W^{l+1,T}\hat{\delta}^{l+1}) \circ f^{l'}(\hat{z}^l), \tag{9}$$

apparently, $\hat{\delta}^L$ can be calculated from previous equation, so $\hat{\delta}^l$ can be calculated recursively. Finally, the gradients is written as:

$$\nabla_{\hat{b}^l} Q = \hat{\delta}^l \qquad \text{and} \qquad \nabla_{W^l} Q = \hat{\delta}^l \otimes \hat{a}^{l-1} \tag{10}$$

In fact, the equations (7), (8), (9) and (10) present a recipe for calculating the cost function gradient in a neural network.

2.2.2 Activation Functions

In a neural network system, each neuron receives an input and applies the activation (transfer) function to generate an output. Therefore, the activation function of a neuron defines the output of that neuron. One activation function, which is broadly used in the hidden layers, is the sigmoid function and its derivative is defined as:

$$f'(z) = f(z)(1 - f(z)),$$

the hyperbolic tangent is another activation function which its derivation is defined as:

$$f'(z) = 1 - f(z)^2,$$

and the rectified linear unit (RELU) which is defined as bellow and usually performs the best [2]:

$$f(z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases} \implies f'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}.$$

2.2.3 Cost Functions and Output Layers

The cost function for a regression problem that applies neural network is defined as:

$$Q = \tfrac{1}{2}||\hat{a}^L - \hat{y}||_2^2,$$

and relevant gradient is written as following:

$$\nabla_{\hat{a}^L} Q = \hat{a}^L - \hat{y}.$$

For the output layer of a neural network system, the identity is used as the activation function, giving:

$$f^{L\prime}(\hat{z}^L) = \hat{1}$$

and

$$\hat{\delta}^L = \hat{a}^L - \hat{y}$$

However, in binary classification problems one may use the logistic function as the activation function of the output layer in order to get a probability a^L that its value is between zero and one. Then, the cost function is written as following:

$$Q = -(y \log(a^L) + (1 - y) \log(1 - a^L)) \tag{11}$$

and the relevant gradient is written as:

$$\nabla_{\hat{a}^L} Q = \frac{\partial Q}{\partial a^L} = -\frac{y}{a^L} - \frac{1 - y}{1 - a^L} = \frac{a^L - y}{a^L(1 - a^L)} \tag{12}$$

that gives:

$$\delta^L = \frac{\partial Q}{\partial a^L}\frac{\partial f^L}{\partial z^L} = \frac{a^L - y}{a^L(1-a^L)} \cdot f^L(z^L)(1 - f^L(z^L)) = a^L - y \tag{13}$$

2.3 Stochastic Minimisation Algorithms

See A.5 for a review of steepest descent, and gradient descent and Newton's method. Since, gradient descent and Newton's method sometimes stuck in local minimum, we can avoid this by using stochastic batch gradient techniques for minimization. This technique picks mini-batches of input training data set and trains the neural network on each batch. Suppose N input training data are given and b is number of batches, therefore N/b is number of iterations that input data are chosen from training data set with replacement. The closed-form for gradient descent is written as:

$$\hat{\beta}_{k+1} = \hat{\beta}_k - \alpha \nabla_{\hat{\beta}} Q(X_{[\hat{\imath},:]}, \hat{y}_{\hat{\imath}}), \tag{14}$$

where $Q(X_{[\hat{\imath},:]}, \hat{y}_{\hat{\imath}})$ denotes the gradient applied to $\hat{x}_j$ for all $j \in \hat{\imath}$, where $\hat{\imath}$ is iteration indices number that finally will reach to N/b. A further improvement is adding momentum, namely using a linear combination of the current and previous gradient, giving:

$$\hat{v}_k = \gamma \hat{v}_{k-1} + Q(X_{[\hat{\imath},:]}, \hat{y}_{\hat{\imath}}), \qquad \hat{\beta}_{k+1} = \hat{\beta}_k - \alpha \hat{v}_k \tag{15}$$

2.4 Weighted Cost Function

Another way to handle biased date is to add weights to the cost function. For weights w_0 and w_1 In our binary case for the credit card data it can be illustrated like this:

$$Q(\hat{\beta}) = -\sum_i (w_0 y_i \log(p_i) + w_1(1-y_i)\log(1-p_i)),$$

The weights can be used to simulate a leveling of significant size differences in the classes. It can also be used to penalize one class in case wrong classification of this class can lead to a loss for the client. For example in the credit card case we would like to avoid defaulters being classified as a secure customer, then a weight could be added to make sure a defaulter is classified correctly. This technique is not implemented in our code due to lack of time.

2.5 Principal component analysis

The Principal component analysis (PCA) is very useful technique when we encounter with a large set that includes correlated variables. Actually, PCA allow to replace a large set with a smaller number of representative variables that express most of the variables in the original set. Mathematically speaking, PCA method defines a line that is as close as possible to the data. This line can minimize the sum of the squared projection of each point to the line [3]. In the other word, we diagonalize the covariance matrix by a method such as SVD, then we select the larger eigen values that are more effective [4]. The covariance matrix is defined as:

$$\begin{aligned}\text{cov}[x,y] &= E[(x-E(x))(y-E(y))] \\ &= E[(x-\mu_x)(y-\mu_y)] \\ &= E[xy]-\mu_x\mu_y\end{aligned}$$

where: $\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i$.

Now covariance matrix can be written as:

$$\text{cov}[\hat{x},\hat{y}] = \begin{bmatrix}\text{cov}(\hat{x}^2) & \text{cov}(\hat{x},\hat{y}) \\ \text{cov}(\hat{y},\hat{x}) & \text{cov}(\hat{y}^2)\end{bmatrix} = \begin{bmatrix}\text{var}(\hat{x}) & \text{cov}(\hat{x},\hat{y}) \\ \text{cov}(\hat{y},\hat{x}) & \text{var}(\hat{y})\end{bmatrix}$$

since, covariance matrix gets values between zero and infinity, we scale the elements of covariance matrix as bellow:

$$\text{c}(\hat{x},\hat{y}) = \frac{\text{cov}(\hat{x},\hat{y})}{\sqrt{\text{var}(\hat{x})\text{var}(\hat{y})}}$$

then we define correlation matrix, namely:

$$\text{K}[\hat{x},\hat{y}] = \begin{bmatrix}1 & \text{c}(\hat{x},\hat{y}) \\ \text{c}(\hat{y},\hat{x}) & 1\end{bmatrix}$$

where:

$$\text{c}(\hat{x},\hat{y}) = \frac{\text{E}[(\hat{x}-\mu_x)(\hat{y}-\mu_y)]}{\sqrt{\text{var}(\hat{x})\text{var}(\hat{y})}}$$

$$\text{c}(\hat{y},\hat{x}) = \frac{\text{E}[(\hat{y}-\mu_y)(\hat{x}-\mu_x)]}{\sqrt{\text{var}(\hat{y})\text{var}(\hat{x})}}$$

3 Implementations

3.1 Object Oriented Programming

Object-oriented programming (OOP), is a method in which the code is written around data or object not logic. Then, the object is defined as a field of data, possessing an individual attributes and behaviour. The example of the object could be a person who is described by its properties such as name, address and etc. In object oriented program the developer efforts focus on manipulating the properties of object rather than writing a logic to control them. In this project one class is used for neural network.

3.2 Standardizing the Data Set

Neural network data set needs to be standardized in the same way as regression. Standardizing improves the accuracy of neural network without making any problem in its performance. Furthermore, this technique is really beneficial for training time because large input values can saturate activation functions (`sigmoid` and `RELU`) and slow down the training. Standardizing the credit card data is implemented as it is shown in following:

```
# Train-test split
trainingShare = 0.5
seed  = 1
XTrain, XTest, yTrain, yTest=train_test_split(Xt, y, train_size=trainingShare,
    random_state=seed, stratify = y)
#scale data, except one-hotted
sc = StandardScaler()
XTrain_fitting = XTrain[:,11:]
XTest_fitting = XTest[:,11:]
#removes mean, scales by std
XTrain_scaler = sc.fit_transform(XTrain_fitting)
XTest_scaler = sc.transform(XTest_fitting)
#puts together the complete model matrix again
XTrain_scaled=np.c_[XTrain[:,:11],XTrain_scaler]
XTest_scaled = np.c_[XTest[:,:11],XTest_scaler]
```

3.3 Exploratory Data Analysis

3.3.1 Visual Observation of Data

Before start working with any data set, it is smart to check the data set visually to avoid any trivial faults affect our training process. For example in our case which is credit card data a feature such as `age` cannot include minus values.

3.3.2 Treatment of Non-Sensible Data

In case of data sets where the relative number of non-sensible data points is small, one can handle the problematic data simply by removing it.

On the other hand, if the number of non-sensible data points is considerable, as in our case, this would exclude too much data from your data set. Then one must consider how to edit the data to be able to still include it in training and testing.

For the credit card data set, the features named `PAY_i` should only contain the values [-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], but as one can see by sorting the data set by these columns, the values [-2,0] appear with a worringly high frequency. If we were to simply remove all data with strange values, we would reduce out data set from 30000 observations to slightly above 4000. As this is a huge reduction, we consider alternative methods of keeping the observations. The simplest seems to us to be to redefine the -2 values to 0 values. These 0 values will then not matter when multiplied with

weights, and thus hopefully not affect the final outcome too much. This was implemented as shown below:

```
for i in [0,2,3,4,5,6]:
    col = 'PAY_{}'.format(i)
    df[col].replace(to_replace=-2, value = 0, inplace=True)
```

The features MARRIAGE and EDUCATION also contain undefined values. As these are categorical, and will be one-hot encoded, the method mentioned above will not work. Here we instead simply drop the non-sensible category from one-hotting, as seen below.

```
# Categorical variables to one-hots, setting nonsensical values to 0
onehotencoder1 = OneHotEncoder(categories='auto')
onehotencoder2 = OneHotEncoder(categories='auto',drop='first')

# sets number of elements in onehot vectors automatically from data.
Xt= ColumnTransformer([("one", onehotencoder1, [1]), ("two", onehotencoder2,
    [2,3]),], remainder="passthrough").fit_transform(X)
```

3.3.3 Up and Down Sampling

When the data set is biased in the sense that one class is over represented compared to the others one will experience when training a neural network that it will learn to favor classifying most things to the over represented class. To deal with this we can ”massage” our training data, and level the representation of the different classes. This can be done by either up sampling or down sampling.
When down sampling a data set, you pick the class that has lowest representation and level the others according to this size by randomly picking observations such that the final size matches the under represented class. Conversely, when we up sample we pick the class with most observations and then increase the other classes to its size in a similar way to bootstrapping, we pick observations at random with replacement till the desired size of the class is achieved. Up sampling should not be done to the test data, because it will create untrue results seeing how one observation would be represented multiple times.
Neural Networks is a very data hungry method, so one should generally choose to upsample.

```
def upsample(xtrain,ytrain,class_size=11000):
    index_0=[]
    index_1=[]
    for i in range(ytrain.shape[0]):
        if ytrain[i]==0:
            index_0.append(i)
        if ytrain[i]==1:
            index_1.append(i)

    x_split_0 = xtrain[index_0,:]
    x_split_1 = xtrain[index_1,:]

    #Picking observations without replacement for the over represented class
    idx0 = np.random.choice(x_split_0.shape[0],size=class_size,replace=False)
    #Picking observations with replacement for the under represented class
    idx1 = np.random.choice(x_split_1.shape[0],size=class_size,replace=True)
```

```
    x_0 = x_split_0[idx0,:]
    x_1 = x_split_0[idx1,:]

    y = np.zeros(class_size*2)
    y[class_size:]=1
    y=y[:,np.newaxis]
    X = np.vstack((x_0,x_1))
    Xy = np.hstack((X,y))
    np.random.shuffle(Xy)

    X_upsampled=Xy[:,:xtrain.shape[1]]
    y_upsampled=Xy[:,-1]
    y_upsampled = y_upsampled[:,np.newaxis]

    return X_upsampled,y_upsampled
```

The relevant results are printed in figure (13) and (14) and discussed in section (4.5)

3.4 Initializing the Weights and Biases

Initializing the weights and biases is carried out via standard and Xavier method. The former method, applied random numbers from `numpy` package and we called it standard method. The latter method uses Xavier technique for initializing the weights and biases. Xavier uses normal distribution around zero ($\mu = 0$) with standard deviation of $\sigma = \sqrt{\frac{2}{n_i+n_{i+1}}}$ where σ where n_i is number of nodes in layer i. The initializing code is implemented as following:

```
# standard method using random distribution
self.parameter_values['W'+str(layer_idx)]=np.random.randn(layer_output_size,
    layer_input_size)*0.1
self.parameter_values['b'+str(layer_idx)]=np.random.randn(layer_output_size,1)
    *0.1

# Xavier method for initializing
stdd = np.sqrt(2/(layer_output_size+layer_input_size))
self.parameter_values['W'+str(layer_idx)]=np.random.normal(0.0,stdd,
                      size=(layer_output_size,layer_input_size))
self.parameter_values['b'+str(layer_idx)] = 0
```

and relevant results are printed in figure (7) and discussed in section (4.2).

3.5 Code Verification

In order to verify that the code yield meaningful results, the code should be executed on a trivial standard data set such as scikitlearn data set make−moons and make−circles. Actually, without this verification, it is hard to say if our treatment of non-sensible data out of the real data set was good or not. The make−moons in our code is implemented as below:

```
# number of samples in the data set
N_SAMPLES = 1000
```

```
# ratio between training and test sets
TEST_SIZE = 0.1
from sklearn.datasets import make_moons
X, y = make_moons(n_samples = N_SAMPLES, noise=0.2, random_state=100)
X_train_moons, X_test_moons, y_train_moons, y_test_moons = train_test_split(X, y
    , test_size=TEST_SIZE, random_state=42)
Moons = simple(X_train_moons,y_train_moons,0.01,nodes_per_layer,
        activation_per_layer)
Moons.train()
Y_test_hat_moons = Moons.forward_propagation(X_test_moons)
acc_test_moons = Moons.accuracy(Y_test_hat_moons, y_test_moons)
print("Test set accuracy: {:.2f} ".format(acc_test_moons))
zerot_moons = np.zeros(y_test_moons.shape[0])
acc_test_moons2 = Moons.accuracy(y_test_moons, zerot_moons)
print("Zero set accuracy: {:.2f} ".format(acc_test_moons2))
```

The relevant results are plotted in figure (4) and discussed in section (4.1.1)

3.6 Covariance and Correlation Matrix

Covariance and correlation matrix allow us to recognize the linear dependency between the different features and the only difference between them is values of correlation matrix are standardized. We used *scikitlearn* functions to print covariance and correlation matrix:

```
creditpd = pd.DataFrame(XTrain_cc)
covmat = creditpd.cov().round(2)

# Generate a mask for the upper triangle
mask = np.zeros_like(covmat, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
plt.figure(figsize=(25,25))
sn.heatmap(data=covmat,mask=mask,cmap='hot_r', annot=True)
plt.show()
```

The relevant results are printed in figure (8) and (9) and discussed in section (4.3).

3.7 Principal Component Analysis

As the correlation matrix hints that some features may be relatively unimportant, we perform a PCA on the data to test this hypothesis. We applied scikitlearn's methods to implement PCA, covariance matrix and correlation matrix. The relevant results are plotted in figures (10), (11), (12) and discussed in section (4.4)

3.8 Neural Network

A multilayer percepton (MLP) system is modeled to find the optimum biases and weights and the algorithm is presented as bellow:

Algorithm 1: Multilayer percepton (MLP) system

set up $\hat{X}_{train}$
Function feed-forward():
  for i : (length(layer), i+)
    cal. $\hat{z}^l = W^l \hat{X}_{train}^{l-1} + \hat{b}^l$
    cal. $\hat{a}^l = f(\hat{z}^l)$
  return $\delta_j^l = f'(z_j^l)\frac{\partial Q}{\partial a_j^l}$ ▷ Q→ refers cost function
Function back-propagation():
  for ii : (length(layer), ii-)
    cal. $\hat{W}_{jk}^l = \hat{W}_{jk}^l - \eta \delta_j^l a_k^{l-1}$ ▷ updating weights
    cal. $\hat{b}_j^l = \hat{b}_j^l - \eta \frac{\partial Q}{\partial \hat{b}_j^l}$ ▷ updating bias
    $\frac{\partial Q}{\partial \hat{W}} = 0$, $\frac{\partial Q}{\partial \hat{b}} = 0$ ▷ minimize Q vs. $\hat{W}$ and $\hat{b}$ via Gradient descent
Return $\hat{W}$ **and** $\hat{b}$

3.8.1 Model Performance

We used *MSE* and R^2 from *scikitlearn* to evaluate the performance of NN for Franke's function. For details refer to appendix (A.8).

4 Results and Discussion

4.1 Accuracy Score and Area Ratio

Accuracy is not a suitable method for biased data therefore we calculate the area ratio defined in given scientific article [1], see equation (**??**). To check the the accuracy of classification figure (3) and table (1) are provided that give a comparison for *gradient descent (GD)*, *Newton Rophson*, *stochastic gradient descent (SGD)*, *stochastic gradient descent with mini batches (SGD-mini)* using error rate and area ratio. In figure (3), as it can be seen the horizontal axis presents number of total data and vertical axis presents cumulative number of target data. The bold black line shows ideal status, the diagonal black line is the baseline, the curved black line is the results from given scientific article and red dashed line is result of our model. As it can be observed in table (1) the results of GD, Newton, SGD and SGD-mini batches are very close to the results of scikitlearn. Whereas NN even shows lower error and higher area ration than scikitlearn.

Method	Error rate		Area Ratio	
	Training	Validation	Training	Validation
GD	0.191± 0.005	0.192± 0.005	0.440± 0.008	0.433± 0.008
Newton	0.184± NA	0.185± NA	0.473± NA	0.459± NA
SGD	0.184± 0.001	0.185± 0.001	0.471± 0.001	0.457± 0.001
SGD-mini	0.189± 0.007	0.190± 0.007	0.442± 0.034	0.427± 0.031
scikitlearn	0.188± 0.000	0.189± 0.000	0.465± 0.000	0.451± 0.000
NN	0.177± NA	0.180± NA	0.571± NA	0.549± NA
NN-PCA	0.176± NA	0.179± NA	0.566± NA	0.539± NA
NN 'decimated'	0.177± NA	0.181± NA	0.562± NA	0.544± NA

Table 1: Accuracy classification for credit card data set which compares the performance of *gradient descent (GD)*, *Newton Raphson*, *stochastic gradient descent (SGD)*, *stochastic gradient descent with mini batches (SGD-mini)*, *scikitlearn*, *NN*, *NN PCA* with 18 features and *NN 'decimated*(Explained in the text).

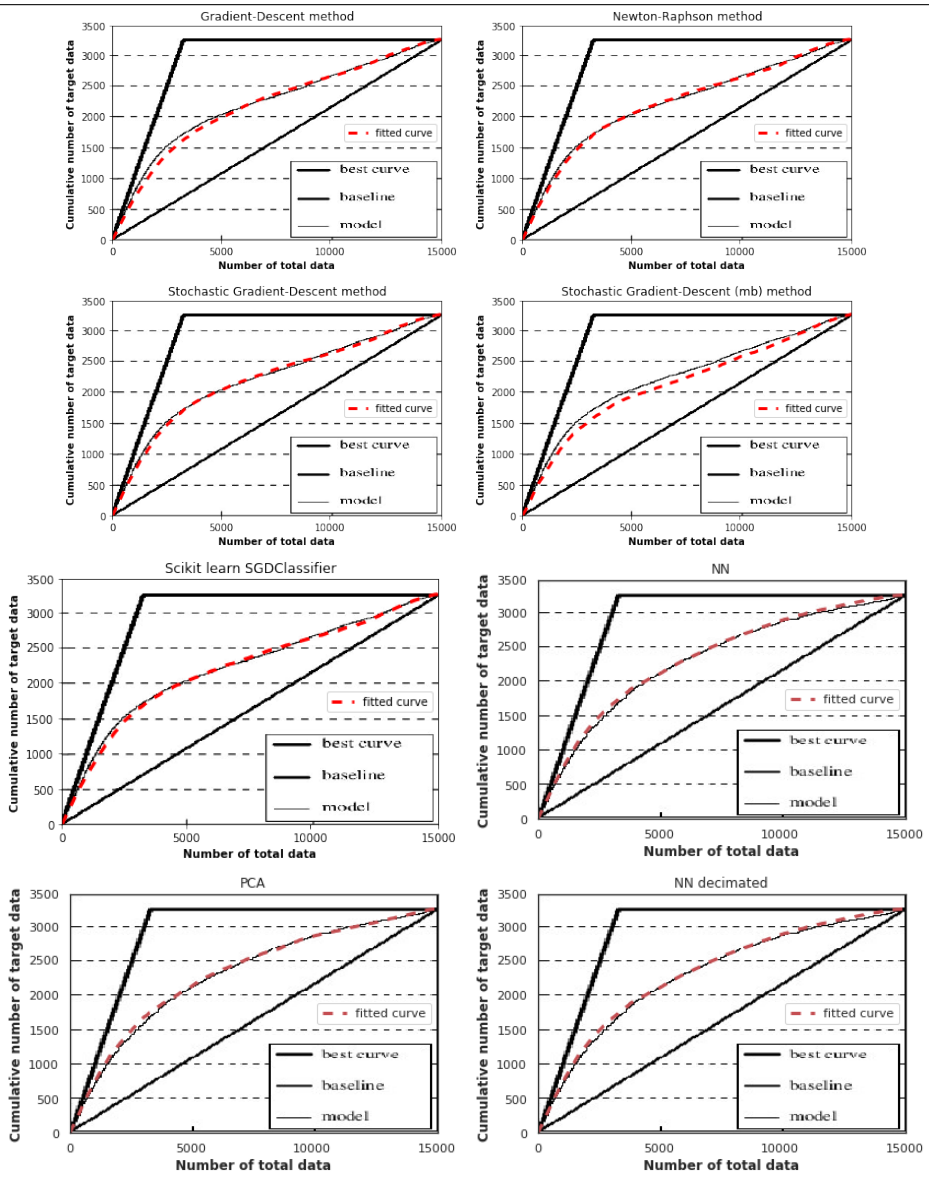

Figure 3: In this figure, as it can be seen the horizontal axis presents number of total data and vertical axis presents cumulative number of target data. The bold black line shows ideal status, the diagonal black line is the baseline, the curved black line is the results from given scientific article and red dashed line is result of our model.

4.1.1 Test the Confusion Matrix for moon data

Prior to start any classification simulation, it is tried to test the confusion matrix for moon data to see how it classifies the data. Confusion matrix of moon data is shown in figure (4). As it is presented

for category of ”Moon2”, our prediction for ”Moon1” and ”Moon2” are 15 and 439 respectively. Whereas, in ”Moon1” category, the predictions for ”Moon1” and ”Moon2” are 429 and 17. As expected the matrix looks diagonal with a small deviation.

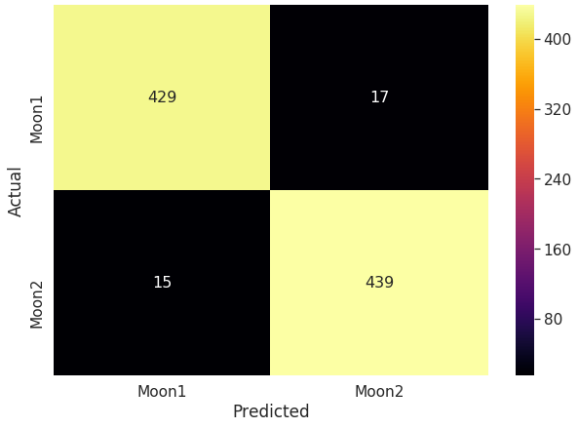

Figure 4: Illustration of confusion matrix on moon data. For category ”Moon2”, the prediction for ”Moon1” and ”Moon2” are 15 and 439 respectively. Whereas, for ”0” category, the predictions for ”Moon1” and ”Moon2” are 429 and 17.

4.1.2 ROC - Curve

A receiver operating characteristic curve (ROC curve) is a plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied [5]. This can be used to determine the cut off in the sigmoid function and determine at what level an observation is classified to a certain class. As discussed in earlier sections we want to avoid a person who would default on his credit card debt to be classified as a non defaulter. From figure it looks like 0.18 is a good place to place the cut off to avoid defaulters being miss classified.

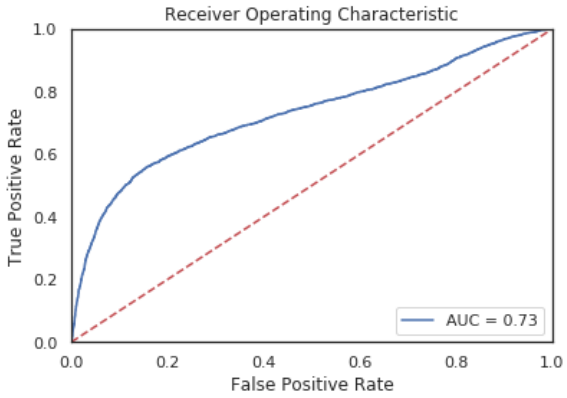

Figure 5: ROC Curve for test data in Logistic regression

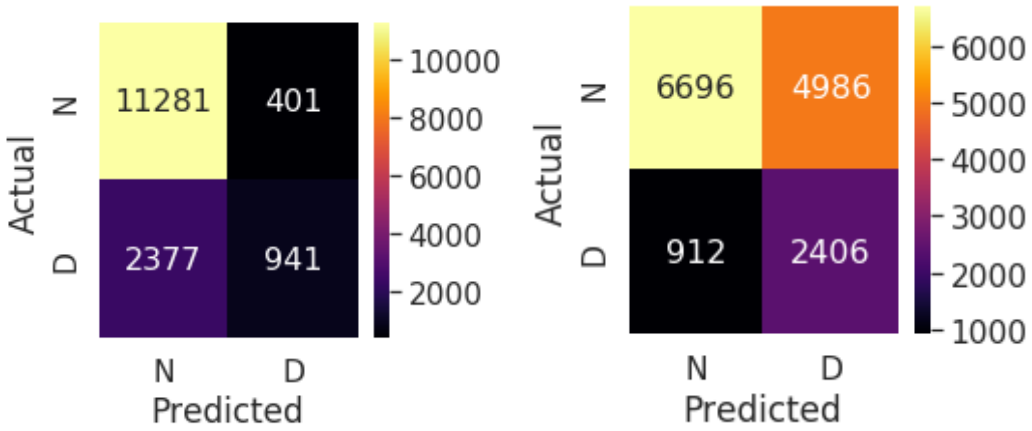

Figure 6: Picture to the left showing confusion matrix of logistic regression using Newton-Raphson minimization, picture to the right showing the result of the cut off at 0.18. as mentioned in section (4.1.2)

From the confusion matrices in figure (6) we see that there is a trade off when it comes to how many customers will be miss classified as defaulters or not. By moving the cut off in the sigmoid function we significantly lowered the amount of defaulters classified as secure customers, but on the other hand there is a large increase in secure customers classified as defaulters. It would have to be up to the client, in this case the credit card company, to decide the ratio between miss classified defaulters and miss classified secure customers.More on this in the conclusion.

4.2 Initializing the weights and biases

As it is presented in figure (7), the cost shows different behaviour for two different initialization i.e. standard and Xavier method. As it is mentioned already in section (3.4), for standard method the

normal distribution of random numbers from `numpy` is used. The following plots show how cost stuck in local minimum via standard initializing and stabilizes at around 0.3. Whereas with Xavier method cost reaches to less than 0.1 which is a global minimum. Figure (7) is prepared on make moon data to present the advantage of Xavier method.

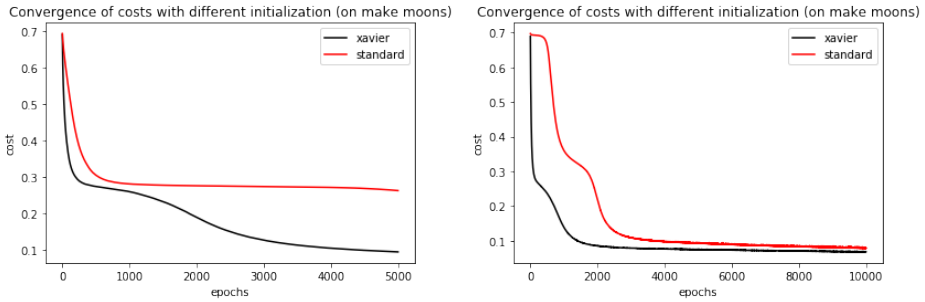

Figure 7: Comparison of cost with different initialization i.e. standard and Xavier on moon data. As it can be observed the red plot stuck in a local minimum and saturated however, using Xavier method allow to reach to a global minimum. From left to right the plots are prepared for 5000 and 10000 epochs.

4.3 Covariance and Correlation Matrix

Figure (8) and (9) illustrate correlation and covariance matrix respectively for credit card data that are calculated by using *scikitlearn* function. The application of covariance and correlation matrix helps to indicate the direction of the linear relationship between the variables. The only difference between them is the values of correlation matrix are standardized. In figure (8) and (9) large values indicate high linear dependency of relevant features which are listed in horizontal and vertical axis. From the correlation matrix, one can see that several features seem less important than others. This can be explored using PCA, as described below.

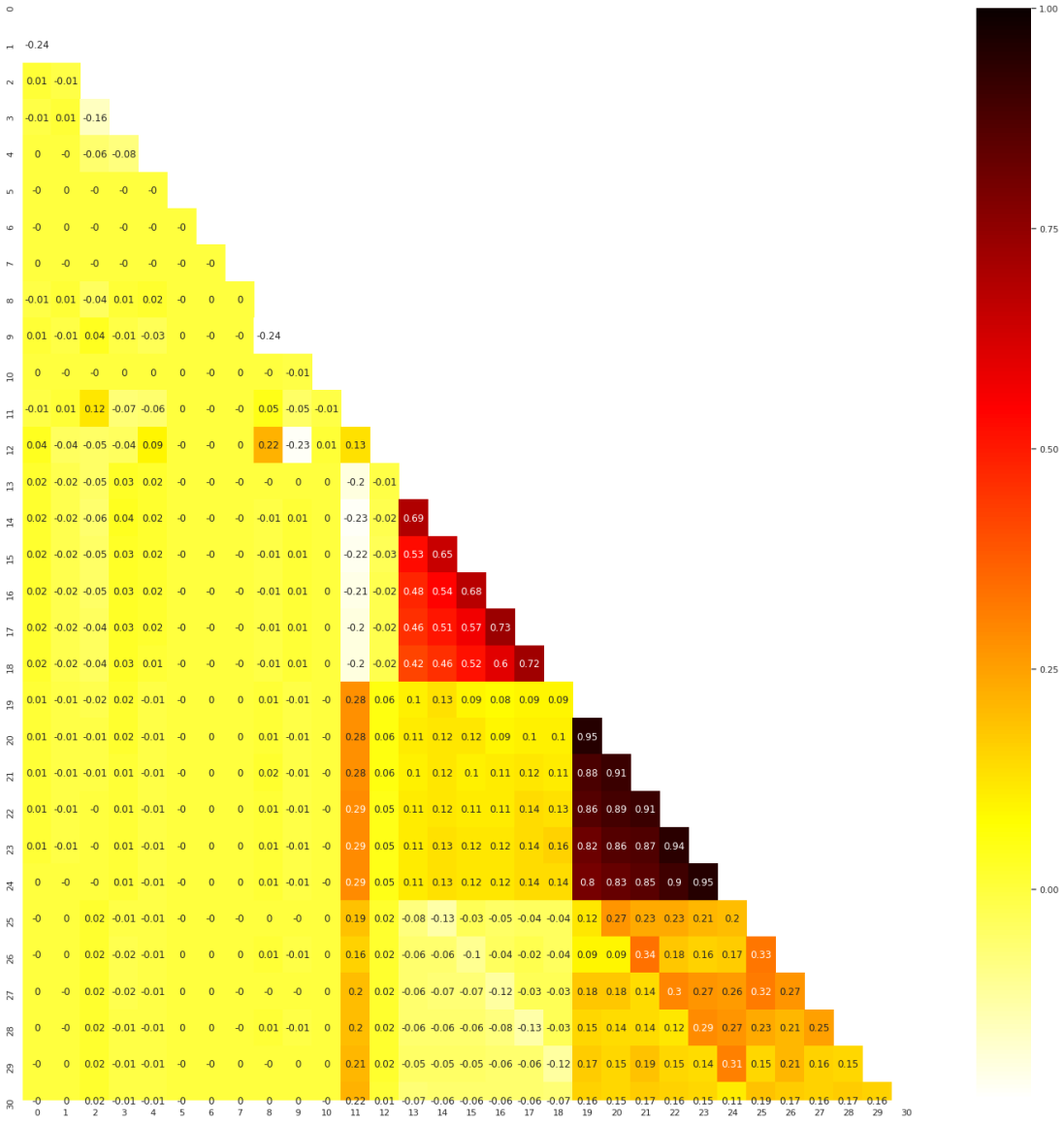

Figure 8: Illustration of covariance matrix for credit card data using *scikitlearn* package.

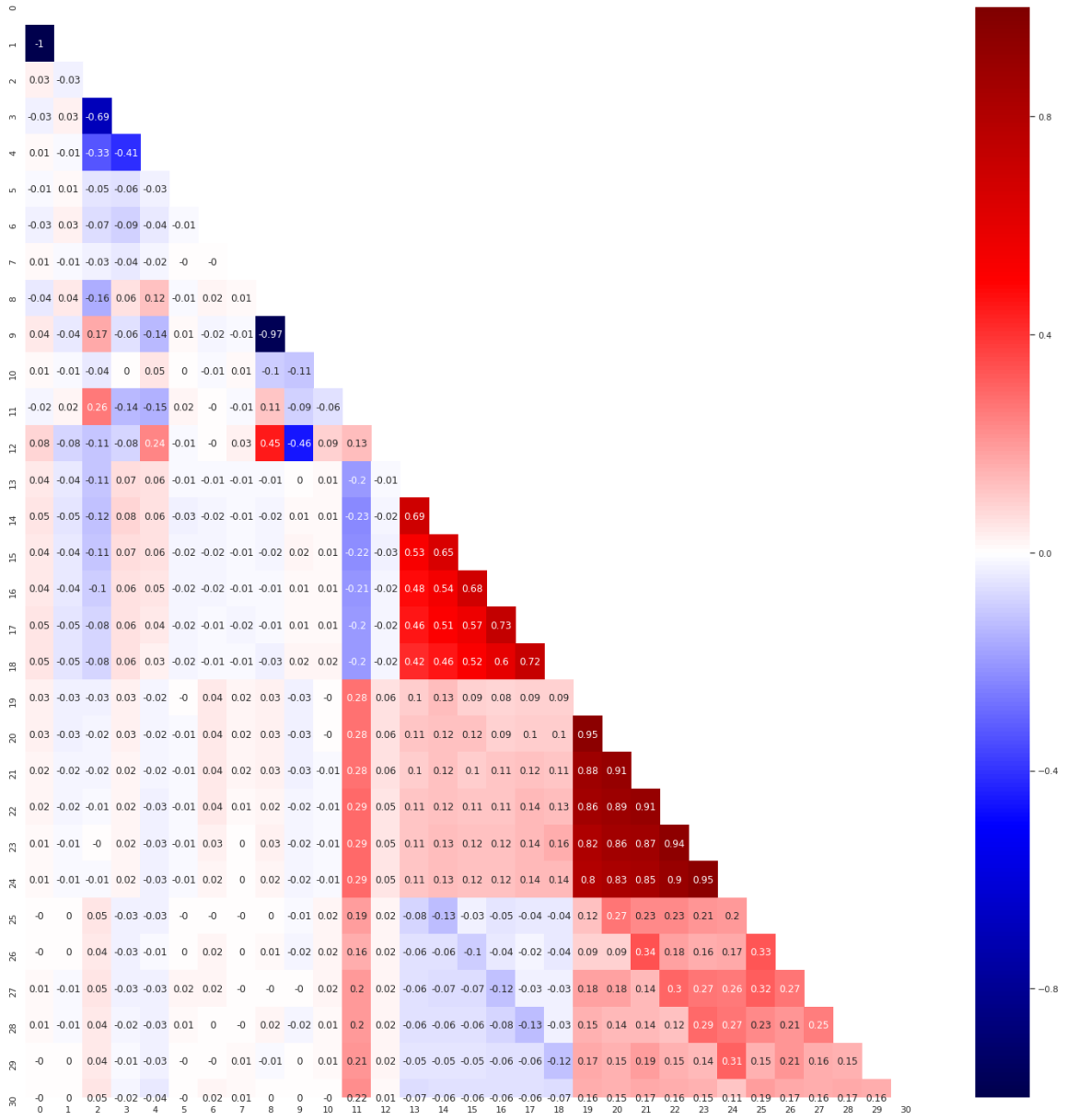

Figure 9: Illustration of correlation matrix for credit card data using *scikitlearn* package.

4.4 Principal component analysis (PCA)

First we run the neural network for varying numbers of features cut by the PCA, as shown in figure (10). We see that the accuracy stays the same until we reduce to 18 PCA features, and after that there is a drastic drop. Rerunning the NN using only these 18 features lead to the results listed as

NN-PCA in table (1).

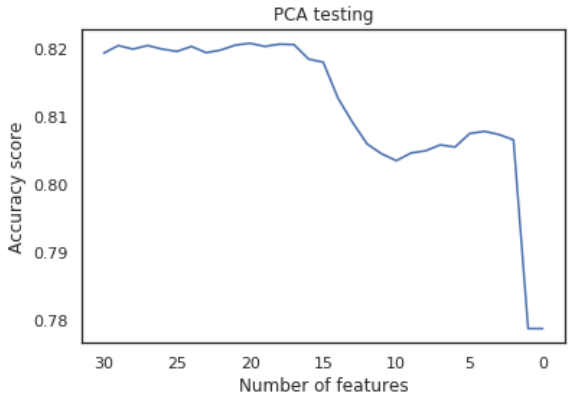

Figure 10: Accuracy scores on test data for NN on PCA transformed data with decreasing number of features.

We have now seen that we can reduce the input features to 18 ”PCA features”, but can we translate this back to the features in our design matrix? An attempt at this was made by first plotting the coefficients of the PCA (18) transform, as seen in figure (11).

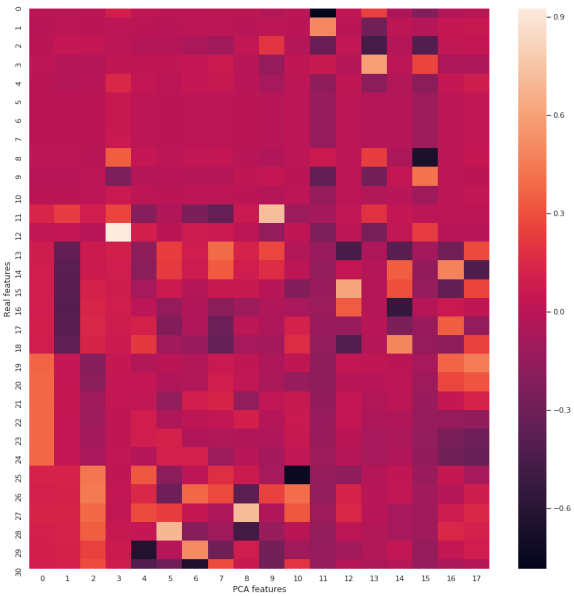

Figure 11: Illustration of PCA conversion coefficients.

This plot seems unhelpful, so we found the mean of the absolute values of coefficients along the x-axis, to see which real features had the highest impact on the PCA features, then sorting them and cutting the 13 features deemed least significant, see figure (12). This leaves us with the 18 features potentially most significant. These were used as the input in the NN, giving the results seen in table (1), dubbed *NN 'decimated'*. Running with the 18 least desired features gives worse values, near the baseline score of 0.22. The method seems promising, so a ranked table of the features follows in Appendix A.9 .

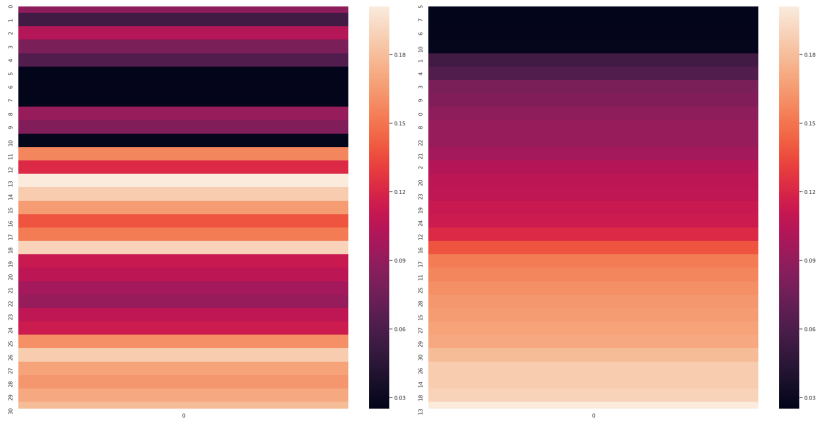

Figure 12: Values of mean impact of real data features on PCA features, sorted to the right.

4.5 Regularization Parameters

To investigate the effect of the regularization parameter (λ) and learning rate (η) on accuracy score, we used the tensorflow code from lecture notes and developed it for the credit card data set. The relevant plots are illustrated in figure (13) and (14) for training and test data. In figure (13) the plot in top is printed for biased data namely the classes are non-identically distributed. Whereas in the bottom one the biases are removed via up-sampling technique.
Figure (14) illustrates results of tensorflow application on test data. In figure (14), outcomes of tensorflow on biased and un-biased data are printed in top and bottom respectively. As we can see up-sampling technique works well for training data such that for $\lambda = 0$, accuracy of 1 is obtained. Whereas, for test data the highest accuracy belongs to biased data set. It means up-sampling leads to over fitting on train data. In fact, application of tensorflow on biased data allow to understand that the highest accuracy is obtained at $\lambda = 0$ which means it is pointless to add regularization parameter to neural network and study its effect.

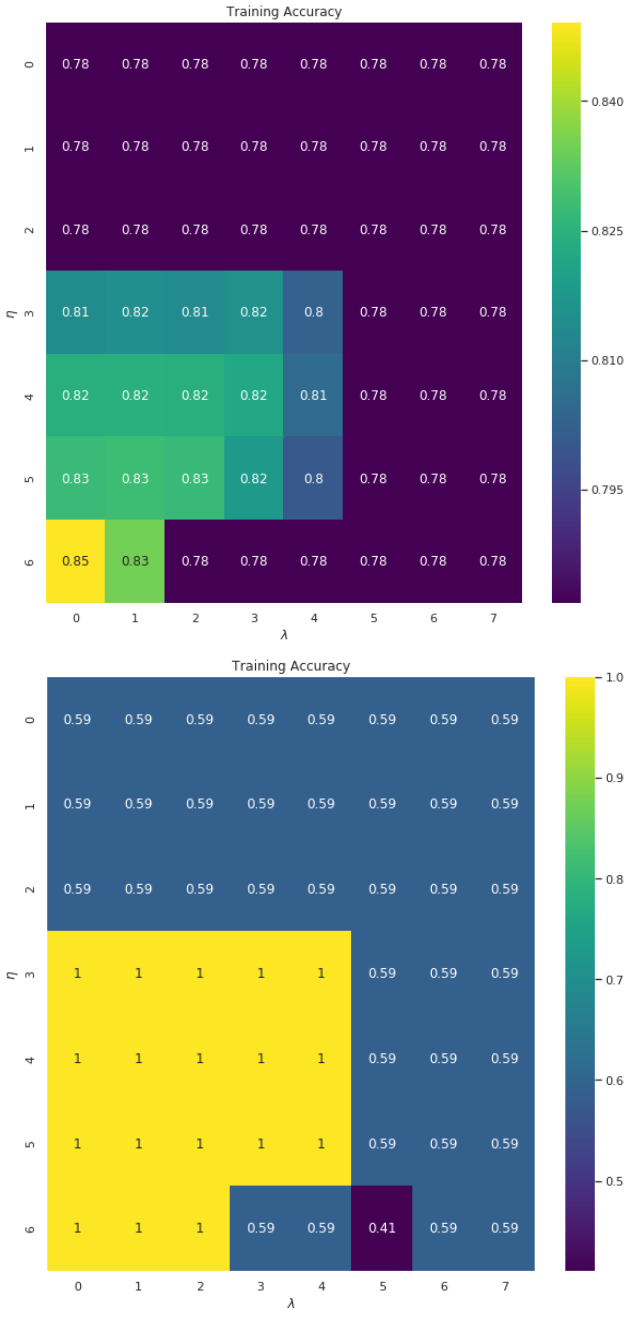

Figure 13: Application of tensorflow to identify the effect of regularization parameter (λ) and learning rate (η) on accuracy score for training data (credit card data set). Here Tensorflow uses two layers including 100 and 50 neurons respectively. The top plot is for biased data (classes are non identically distributed) and in bottom plot we removed the biases via up-sampling technique.

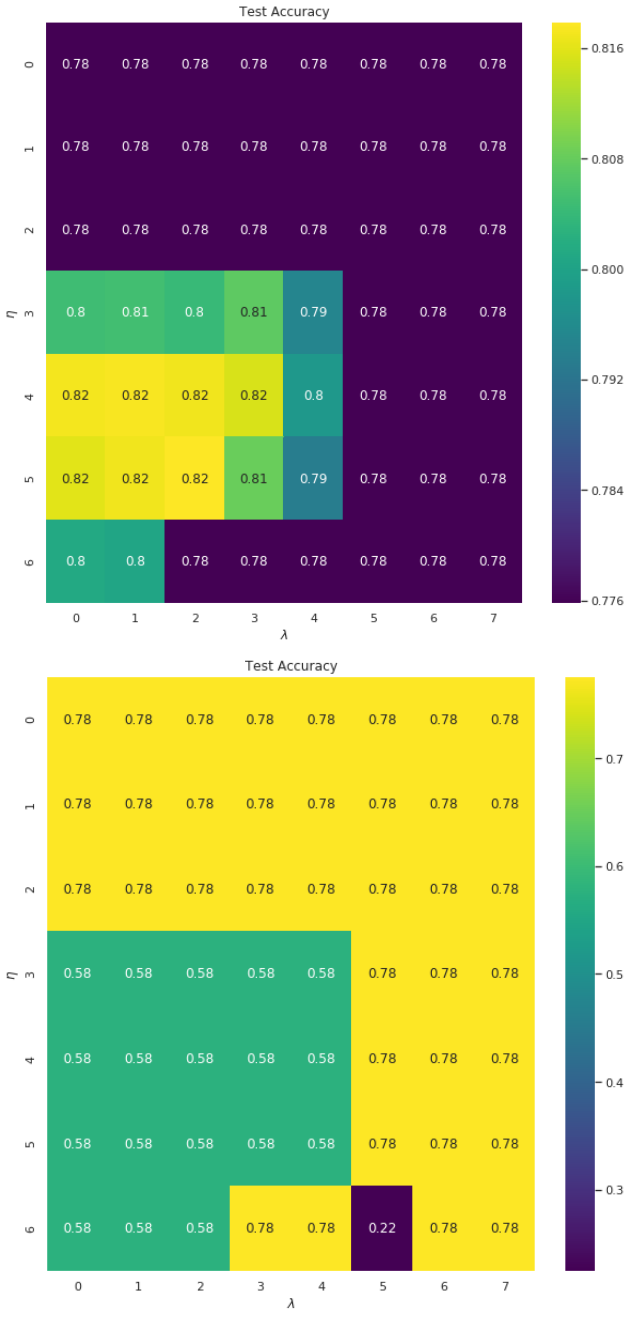

Figure 14: Application of tensorflow to identify the effect of regularization parameter (η) and learning rate (η) on accuracy score for test data (credit card data set). Tensorflow uses two layers including 100 and 50 neurons respectively. The top plot is for biased data (classes are non identically distributed) and in bottom plot we removed the biases via up-sampling technique.

4.6 Neural Network for Classification

A neural network system with flexible number of layers and neurons is coded. Although many popular activation functions exist we have used *ReLU*. The NN is executed for variety of architecture and the results are plotted in figure (15). As it is observed NN quickly reaches to high accuracy and the higher learning rate (λ) the more flattened curve. Namely, for the smaller learning rates it takes longer time to reach to desired accuracy. We tried running a loop over different architectures and activation functions, but it was time consuming to run and we didn't achieve any impressing results. By more random trial and error we found that the best result was for 10x10 system (2 layers and 10 neurons in each).

Our NN uses a SGD with minibatches, with 100 batches in this case. As so few customers default, this would result in many minibatches containing only non-defaulters. One would think this would result in worse learning, so we should consider upsamling or adding a weight parameter to the cost function. Surprisingly the results were good with this current batch size, and we did not have time to explore this further.

Over fitting is always something to keep in mind with any supervised learning method and it has been a topic on our mind for this project. The stochasticity from stochastic gradient descent applied to minimize the cost function will help avoid over fitting. Other strategies to avoid over fitting is to add a regularizer to the cost function or to do a random drop out of nodes in the layers. For this project one can see that over fitting is not occurring fast enough to be problematic.

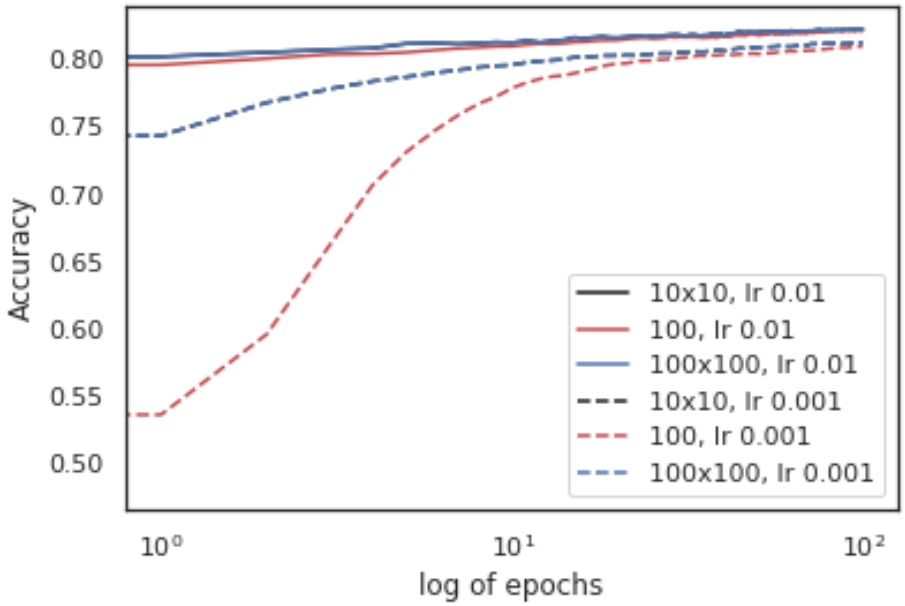

Figure 15: Accuracy on test data as a function of the number of epochs for some neural network architectures. Here two different learning rates i.e. $\eta = 0.01, 0.001$ with batch size of 10x10 and 100 are shown. As it can be seen the neural networks converges to perfect accuracy with even one hidden layer including 100 neurons

4.7 Neural Network for Regression

In table (2), one can compare the performance of training and test data which are sampled from Franke's function based on OLS, Ridge, Lasso and NN. The results for OLS, Ridge and Lasso are copied from project 1 but relevant results for NN have computed recently in project 2. Although in train data the calculated MSE by NN is higher but R^2 is larger than other methods whether for train and test which convince us to say NN performs the best fit for Franke's.

Method	MSE(train)	R^2(train)	MSE(test)	R^2(test)
OLS	0.000585	0.9888580	0.0021412	0.9592810
Ridge	0.0002058	0.9960936	0.0011889	0.9778371
Lasso	0.0006745	0.9869749	0.002109	0.9501375
NN	0.0007536	0.9910987	0.0008016	0.9905277

Table 2: Comparison the performance of OLS (A), Ridge (A.2), Lasso (A.3) and NN on Franke's function. Neural network shows the best fit [Note: an update of numpy reduced the performance of regression NN, we did not have time to explore what happened.]

The bias-variance trade off for OLS, Ridge, Lasso and NN are reported in table (3). The results for

OLS, Ridge and Lasso are copied from project 1 but relevant results for NN have computed recently in project 2. As it can be seen the bias-variance works well for NN.

Method	MSE	Bias+Variance	Bias	Variance
OLS	0.000124982	0.000124982	0.000119858	5.12393420e-06
Ridge	0.005489025	0.005489025	0.005431349	5.76760195e-05
Lasso	0.031897187	0.031897187	0.031727567	0.000169620
NN	0.010373027	0.010373027	0.010371483	1.544e-06

Table 3: Illustration of bias-variance (A.7) trade off for Franke's function using Bootstrap (A.6.2) method. For Ridge and Lasso the $\lambda = 0.01$, and polynomial degree is 10.

5 Conclusion

In this study logistic regression and neural networks techniques have been applied to analyze the credit card data. At first step the analysis results of Yeh and Lien [1] are reproduced using GD, Newton, SGD, SGD-mini batches and scikitlearn which Newton and SGD showed the best fit.

Two initialization method are used namely random initialization and Xavier where the latter was more efficient, because Xavier escapes local minimum and converges toward a global minimum.

Neural network with the flexibility in number of layer and number of neurons with well-known activation functions is coded and its robustness for credit card data and Franke's function is studied. According to the reported results in table 1, 2 and 3 it is realized that NN give better accuracy with the architecture of 10x10 (two layers of 10 neurons each).

We used tensorflow to check whether regularization parameter λ effect and it is realized that in our case regularization parameter is not required.

PCA technique is used to identify less significant real features, and remove them from the training data. It is observed that for NN the results differ only in the last significant digit.

The results found for the credit card data are useful for the client, the bank, when deciding on accepting or declining a new customer. As mentioned earlier the cut off for the sigmoid function can be adjusted to make sure the client doesn't accept new customers who would default on their loan. A graph to select the correct cut off dependent on how much a False Positive (defaulting customer) costs compared to a False Negative (not accepted loan applicant who could handle her loan) is shown in figure (16). To use this plot, assume you can say that 1 "bad" customer is worth x "good" customers. Follow the x-axis to the correct point, and read off the cut-off for the sigmoid on the y-axis. The results can also be used for calculating expected annual revenue, where you would likely have a different ratio between False positives and false negatives than above, depending on how optimistic (or risky) the credit card company are willing to be in their estimates of income. When used in this way the credit card company can adjust expected revenue according to how many defaulted loans they can afford.

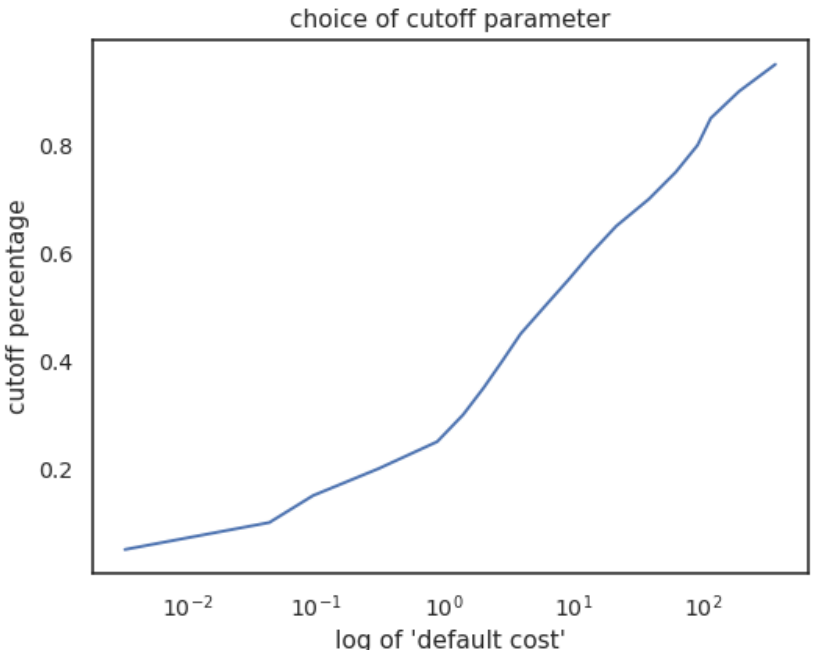

Figure 16: Graph to help decide sigmoid cutoff. Find the ratio of ”good” customers per ”bad” customer along the x-axis to read the cut off from the y-axis. Note that the x-axis is logarithmic.

Lastly the regression problem where we used the results from project 1 to compare with the performance of a NN to fit the Franke function. The NN has the highest R^2 score on the test data out of the four methods, but has lower MSE than Lasso.

A Ordinary Least Squares

Ordinary least squares (OLS) is the simplest and more popular method and it is defined as summation of the squared differences between predicted and measured values for each data point:

$$Q = \sum_{i=0}^{n-1} \epsilon_i^2 = \hat{\epsilon}^T \hat{\epsilon} = (\hat{y} - \hat{X}\hat{\beta})^T (\hat{y} - \hat{X}\hat{\beta})$$

Q is called cost function and as it is shown above, summation of squared differences i.e. $\hat{\epsilon}^2$ can be considered as inner product of the vector with itself. We are going to use this equation to find a value for $\hat{\beta}$ which minimizes Q as cost function. To achieve this we differentiate Q with respect to $\hat{\beta}$ which gives:

$$\frac{\partial Q(\hat{\beta})}{\partial \hat{\beta}} = 0 = \hat{X}^T(\hat{y} - \hat{X}\hat{\beta})$$
$$\hat{X}^T\hat{y} = \hat{X}^T\hat{X}\beta$$

in cases where we have an invertible $\hat{X}$, the analytical solution is:

$$\hat{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\hat{y} \tag{16}$$

Furthermore, the relevant variance could be analytically determined by:

$$\text{Var}\left(\hat{\beta}\right) = \sigma^2 \left(X^T X\right)^{-1} \tag{17}$$

A.1 Geometric Aspect of Ordinary Least Squares (OLS)

For ordinary least square we need to find a linear combination among the $\hat{X}$ columns such that gives the closest value to $\hat{y}$. This is fulfilled when $\hat{X}\hat{\beta}$ equals to the projection of $\hat{y}$ onto column space of $\hat{X}$ and in case the $\hat{X}$ has linear independent columns the solution will be unique. Here, we can say $\hat{X}\hat{\beta}$ is the projection of $\hat{y}$ onto the column space of $\hat{X}$, therefore the error vector namely, $\hat{y} - \hat{X}\hat{\beta}$, is orthogonal to the rows of $\hat{X}^T$, namely:

$$X^T(\hat{y} - \hat{X}\hat{\beta}) = \hat{0} \implies \hat{X}^T\hat{X}\hat{\beta} = \hat{X}^T\hat{y} \tag{18}$$

if we solve the equation with respect to $\hat{\beta}$ gives:

$$\hat{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\hat{y}. \tag{19}$$

A.2 Ridge

One of the disadvantages of the OLS method is that the matrix is not necessarily invertible. We know it is required to get an inverse of $(\hat{X}^T\hat{X})^{-1}$ and when matrix is not invertible, it is impossible to model the data by this method [4]. It is clear that for cases where $\hat{X}$ is not a square matrix, and

have many columns it is most likely that we get a singular matrix (zero determinant). A simple solution to this problem is to add $\lambda\hat{I}$ term, where $\lambda \geq 0$, and make the matrix invertible. In this way by superimposing a variable to the solution space we will have an invertible matrix and subsequently a unique solution [3]. The cost function of the Ridge method is:

$$Q(\hat{\beta}) = \left\| \left(\hat{y} - \hat{X}\hat{\beta}\right) \right\|_2^2 + \lambda \left\| \hat{\beta} \right\|_2^2 = \sum_{i=1}^{N} (y_i - x_{ij}\beta_j)^2 + \lambda \sum_{i=1}^{p} \beta_i^2 \tag{20}$$

In Ridge regression, we needs to have $\lambda \neq 0$. By setting $\lambda = 0$ we have ordinary least squares and as small as possible errors. But non-zero values of λ provides β_j results more reasonable predictors for those values of x which are not in training data set[2].

$$\frac{\partial Q(\hat{\beta})}{\partial \hat{\beta}} = 0 = \hat{X}^T \left(\hat{y} - \hat{X}\hat{\beta}\right) - \lambda\hat{\beta} \tag{21}$$

and then gives:

$$\left(\hat{X}^T\hat{X} + \lambda I\right)^{-1} \hat{\beta} = \hat{X}^T\hat{y} \tag{22}$$

finally, closed form of $\hat{\beta}_{\text{Ridge}}$ is written as:

$$\hat{\beta}_{\text{Ridge}} = \left(\hat{X}^T\hat{X} + \lambda I\right)^{-1} \hat{X}^T\hat{y}. \tag{23}$$

Furthermore, the relevant variance could be analytically determined by:

$$\text{Var}\left[\hat{\beta}^{Ridge}\right] = \sigma^2 \left[X^TX + \lambda I\right]^{-1} X^TX \left(\left[X^TX + \lambda I\right]^{-1}\right)^T \tag{24}$$

A.3 Lasso

Generally, the cost function of Lasso (Least Absolute Shrinkage and Selection Operator) is written as:

$$Q(\hat{\beta}) = \left\| \left(\hat{y} - \hat{X}\hat{\beta}\right) \right\|_2^2 + \lambda \left\| \hat{\beta} \right\|_1 = \sum_{i=1}^{N} (y_i - x_{ij}\beta_j)^2 + \lambda \sum_{i=1}^{p} |\beta_i| \tag{25}$$

The only difference between Lasso and Ridge is, we take L_1-norm instead of L_2-norm in Lasso regression. This brings an advantage for Lasso regression which is, certain values of λ gives sparse solution, namely $\beta_j = 0$. As we did for other methods, it is expected to get a gradient of the cost function and set it to zero. However, as it could be seen here, the absolute function is not differentiable at zero. The derivative of absolute function is defined as:

$$\frac{d|x|}{x} = \text{sgn}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} \tag{26}$$

then, the gradient of the cost function which is mentioned above is written as:

$$\nabla Q = -2\hat{X}^T(\hat{y} - \hat{X}\hat{\beta}) + \lambda \, \text{sgn}(\hat{\beta}), \tag{27}$$

The mentioned constraint results in nonlinear solutions and there is no closed form expression of Lasso [3]. Thus, to find the minimum of cost function a dedicated minimum algorithm should be applied to find proper β_j. It seemss, the gradient descent algorithm with a proper step is convenient and Newton's method which includes **Hessian** of the cost function is another alternative. Actually, Hessian matrix for cost function of Lasso, is the second derivative of the OLS cost function, namely:

$$H_{kl} = H_{lk} = \frac{\partial^2 Q_\lambda}{\partial\beta_l\beta_k} = \frac{\partial}{\partial\beta_l}(\nabla_k Q) = \frac{\partial}{\partial\beta_l}(-2x_{ik}(y_i - x_{ij}\beta_j)) = 2x_{ik}x_{il}, \tag{28}$$

in matrix form, Hessian is written as:

$$H = 2\hat{X}^T\hat{X}. \tag{29}$$

then, for gradient we will have:

$$\nabla Q_\lambda = -2\hat{x}^T\hat{y} + H\hat{\beta} + \lambda \, \text{sgn}(\hat{\beta}). \tag{30}$$

According to the theory, it is obvious that large values of λ will force some of the β_j parameters to zero which causes non-differentiable cost function. Here, for minimizing Lasso cost function we used **scikit-learn**. Nevertheless, as a computational discussion we know evaluation of the ∇Q_λ incorporates with matrix-vector computation with a FLOPS of $\mathcal{O}(p^2)$. Due to using the second derivative, the cost function is independent of $\hat{\beta}$. Thus, we can use Cholesky method for decomposition of Hessian. Cholesky method has FLOPS of $\mathcal{O}(p^3)$. Therefore, after using Cholesky only $\mathcal{O}(p^2)$ FLOPS per iteration is required for solving set of linear equations.

A.4 Minimization

Minimization of the cost function is a crucial part of any regression method. For OLS and Ridge regression there are analytical solutions to find β_j by minimizing the formula for $Q(\hat{\beta})$. But, some methods, such as Lasso regression, need to use other minimization algorithms.

A.4.1 Newton's Method

We use Taylor expansion of ∇Q around the point of $\hat{\beta}_0$. If we write Taylor expansion at a point $\hat{\beta}_0 + \delta\hat{\beta}$ then we have:

$$\nabla Q(\hat{\beta}_0 + \delta\hat{\beta}) = \nabla Q(\hat{\beta}_0) + H(\hat{\beta}_0)\delta\hat{\beta}, \tag{31}$$

here $H(\hat{\beta}_0)$ is the derivative of ∇Q, namely the Hessian of Q, evaluated at β_0. therefore, we have:

$$H(\hat{\beta}_0)\delta\hat{\beta} = -\nabla Q(\hat{\beta}_0), \tag{32}$$

and, in general, solving

$$H(\hat{\beta}_i)\delta\hat{\beta}_i = -\nabla Q(\hat{\beta}_i), \tag{33}$$

by defining $\hat{\beta}_{i+1} = \hat{\beta}_i + \delta\hat{\beta}_i$, the process can be continued until convergence.

A.5 Steepest Descent

The algorithms that are used for nonlinear optimizations are in the category of iterative methods. For a function such as f(x) that should be minimized, these methods generate a sequence of points that gradually move toward smaller values. There are two main types of algorithms in nonlinear optimizations:

- *Line search method*: in this algorithm, by using the information of nonlinear function f(x), a search direction such as d_k is chosen from the current points of $x = \{x_1, ..., x_n\}$. Then choosing the step length of α_k gives a new point as following:

  $$x_{k+1} = x_k + \alpha_k d_k$$

  x_{k+1} has a small, or maybe smallest possible value.

- *Trust region method*: in TRM algorithm, a region is defined in neighborhood of x_k in which a certain model can approximate the original objective function by using the information of function. Then, according to the depiction of the model within the region, TRM takes a step forward. The difference between TRM and line search method is, TRM is able to determine the step size prior to improve the direction.

These two methods are based on quadratic approximation of f(x) with two different approaches, but the aim of this study is the line search methods. In fact, in this method, an iterative procedure is constructed to solve $Ax = b$. Writing a quadratic function gives:

$$f(x) = \frac{1}{2}x^T Ax - x^T b, x \in R^n$$

rewriting the function for multiple variable function gives:

$$f(x_1, ..., x_n) = \frac{1}{2}\sum_{i,j=1}^{n} A_{i,j}x_i x_j - \sum_{i=1}^{n} b_i x_i$$

The extremum of above multivariable function happens at the point where the derivatives in each of the directions x_i becomes zero. The vector of derivatives which are called **gradient** is shown by $\nabla f(x)$. Namely, by vanishing the gradient ($\nabla f(x) = 0$) we have extremum point:

$$\begin{aligned}\frac{d}{dx_k}f(x_1, ..., x_n) &= \frac{1}{2}(\sum_{i=1}^{n} A_{ik}x_i + \sum_{j=1}^{n} A_{kj}x_j) - b_k \\ &= \sum_{j=1}^{n} A_{kj}x_j - b_k\end{aligned}$$

Obviously, for x is the extremum for $Ax - b$ if make it zero. thus

$$\nabla f(x) = Ax - b = 0$$

In the other hand, since A is symmetric and positive definite, the extremum is minimum. Now, the question is how α is calculated. Regarding to calculate α, we have:

$$\begin{aligned} f(x_{k+1}) = f(x_k + \alpha_k d_k) &= \frac{1}{2}(x_k + \alpha_k d_k)^T A(x_k + \alpha_k d_k) - (x_k + \alpha_k d_k)^T b \\ &= \frac{1}{2}x_k^T A x_k + \frac{1}{2}\alpha d_k^T A x_k + \frac{1}{2}x_k^T A \alpha d_k + \frac{1}{2}\alpha^2 d_k^T A d_k - x_k^T b - \alpha d_k^T b \\ &= f(x_k) + \alpha d_k^T (A x_k - b) + \frac{1}{2}\alpha^2 d_k^T A d_k \end{aligned}$$

let suppose

$$g_k = A x_k - b = \nabla f(x_k)$$

then, $f(x_{k+1})$ is rewritten as:

$$f(x_k + \alpha_k d_k) = f(x_k) + \alpha d_k^T g_k + \frac{1}{2}\alpha^2 d_k^T A d_k$$

for minimizing the $f(x_{k+1})$, it is required to have:

$$\frac{d}{d\alpha_k} f(x_k + \alpha_k d_k) = d_k^T g_k + \alpha_k d_k^T A d_k = 0$$

which results:

$$\alpha = \frac{-d_k^T g_k}{d_k^T A d_k}$$

Moreover, it is required to know which d_k leads us to the minimum. In this regard, it is tried to investigate it by Taylor's theorem:

$$f(x + h) = f(x) + \nabla f(x).h + h^T \nabla^2 f(x + h)h$$

By substitution of small steps of h in Taylor expansion, the term with first order will dominant. Then, Cauchy-Schwarz inequality gives:

$$\nabla f(x).h \geq - \parallel \nabla f(x) \parallel \parallel h \parallel$$

In above mentioned relation, for consistency with other equations we replace h with d_k. It is trivial that the equality just holds for $d_k = -\beta \nabla f(x)$ where $\beta \geq 0$. Here, d_k is called **descent direction** if satisfies the condition of $\nabla f(x).d_k < 0$. Therefore, moving toward descent direction and taking small steps of d_k gives [6]:

$$f(x_{k+1}) < f(x_k)$$

A.5.1 Gradient Descent

Some times it is not efficient to evaluate Hessian matrix. To circomvent this problem, we replace Hessian with a number which is shown as $1/\alpha$. Therefore, equation (33) is rewritten as

$$\hat{\beta}_{i+1} = \hat{\beta}_i - \alpha \nabla Q(\hat{\beta}_i), \tag{34}$$

To interpret this coefficient geometrically, we say, moving a small step in the opposite direction of the gradient gives reduction in value of cost function $\hat{\beta}$ will approach to minimum value. If we have a convex function function, small steps results convergence but slowly while larger steps may not give convergence. It is worth to note that there is no analytical solution to calculate the variance of Lasso β as we had for OLS and Ridge. Therefore, we have to run the program hundreds of time and then find the mean value and relevant variance.

A.6 Resampling Methods

In this study Franke's function is used as a data set and it is modeled via OLS, Ridge and Lasso. Resampling methods are techniques that improves the accuracy of prediction and let us find estimation for variance of β via dividing the data set to **training data** and **test data**. Often you would also split off a part of the data set into **validation data**. This validation data is then used to evaluate the final model, found by training and testing on the train and test data sets. The reason for this is that evaluation becomes more biased when skill on the validation dataset is incorporated into the model configuration [7]. Note that which set is called the validation set and which is the test set varies from source to source. These techniques are mostly applied in two cases, first when data collection is difficult and it is expensive to gather new data. Second, our model could be overfitted which means the training data is very close to the data set and starts fitting noise in the data, such that predicting any new data might fail. The idea of resampling is very simple which is required to split the data set into training and testing data. For this purpose, we can use K-fold cross-validation and bootstrapping methods.

A.6.1 K-fold

The approach for K-fold is straight forward, such that instead of perform a regression on the entire data set, first we divide the data set into K number of subsets with equal size(the last set may be smaller if the data set doesn't divide with the number of folds). It is important to note that the should be shuffled (randomized) before distributing them to the subsets. Now we have a subset as 'validation' set while the rest are the 'training' sets.

A.6.2 Bootstrap

In Bootstrapping, the random selection of data set is carried out with replacement. Random selection with replacement means that each value may appear multiple times in the one sample.

A.7 Bias and Variance Trade Off

In data analysis using supervised learning methods, such as the models studied here, OLS, Ridge and Lasso, will have a conflict between minimizing two of the sources of error namely bias and variance. Minimizing one means maximizing the other. Therefore, the best performance happens when we find a balance between the bias and variance, where the mean squared error (or cost function) is at its minimum. Below is the equation for error estimation, mean squared error:

$$\begin{aligned} E[(y-\hat{f}(x))^2] &= \Big(E[f(x)-\hat{f}(x)]^2\Big) + \Big(E[\hat{f}(x)^2] - E[\hat{f}(x)]^2\Big) + \sigma^2 \\ &= \text{Bias}(\hat{f}(x))^2 + \text{Var}(\hat{f}(x)) + \sigma^2 \end{aligned} \tag{35}$$

σ^2 is the first term representing the noise and it is indisputable. The second term is the deviation of the average prediction from the true value, noise-free model, which is the (squared) bias. The last term is the so-called variance and shows how much the predictions vary Mathematically, we can say if we have $D = (\vec{x}_i, y_i)_{i=1}^N$ as the data set, which is produced by a function such as Franke's function, then we have:

$$y_i = f(\hat{x}_i) + \varepsilon_i = f_i + \varepsilon_i$$

where the variables ε_i are independent and normally distributed with variance σ^2 around zero. By using any regression method it is possible to predict $\tilde{y}_i$ from the data set then, the expected mean square error will be:

$$\begin{aligned} E_{D,\varepsilon}\left[\left\|\hat{y}-\hat{\tilde{y}}_D\right\|_2^2\right] &= E_{D,\varepsilon}\left[\left\|\hat{y}-\hat{f}+\hat{f}-\hat{\tilde{y}}_D\right\|_2^2\right] \\ &= E_{D,\varepsilon}\left[\left\|\hat{y}-\hat{f}\right\|_2^2 + \left\|\hat{f}-\hat{\tilde{y}}_D\right\|_2^2 + 2(\hat{y}-\hat{f})\cdot(\hat{f}-\hat{\tilde{y}}_D)\right] \\ &= E_{D,\varepsilon}\left[\left\|\hat{y}-\hat{f}\right\|_2^2\right] + E_{D,\varepsilon}\left[\left\|f-\hat{\tilde{y}}_D\right\|_2^2\right] + 2E_{D,\varepsilon}\left[(\hat{y}-\hat{f})\cdot(\hat{f}-\hat{\tilde{y}}_D)\right] \\ &= E_{D,\varepsilon}\left[\|\hat{\varepsilon}\|_2^2\right] + E_{D,\varepsilon}\left[\left\|\hat{f}-\hat{\tilde{y}}_D\right\|_2^2\right] + 2E_{D,\varepsilon}\left[\hat{\varepsilon}\cdot(\hat{f}-\hat{\tilde{y}}_D)\right] \\ &= \sigma^2 + E_{D,\varepsilon}\left[\left\|\hat{f}-\hat{\tilde{y}}_D\right\|_2^2\right] + 0\cdot E_D[(\hat{f}-\hat{\tilde{y}}_D)] \\ &= \sigma^2 + E_D\left[\left\|\hat{f}-\hat{\tilde{y}}_D\right\|_2^2\right] \end{aligned}$$

adding and subtracting the average prediction, $E_D\left[\hat{\tilde{y}}_D\right]$ gives:

$$
\begin{aligned}
E_D\left[\left\|\hat{y}-\hat{\tilde{y}}_D\right\|_2^2\right] &= \sigma^2 + E_D\left[\left\|\hat{f}-E_D\left[\hat{\tilde{y}}_D\right]+E_D\left[\hat{\tilde{y}}_D\right]-\hat{\tilde{y}}_D\right\|_2^2\right] \\
&= \sigma^2 + \left\|\hat{f}-E_D[\hat{\tilde{y}}_D]\right\|_2^2 + E_D\left[\left\|E_D\left[\hat{\tilde{y}}_D\right]-\hat{\tilde{y}}_D\right\|_2^2\right] \\
&\quad + 2E_D\left[(\hat{f}-E_D\left[\hat{\tilde{y}}_D\right]\right)\cdot 0 \\
&= \sigma^2 + \left\|\hat{f}-E_D\left[\hat{\tilde{y}}_D\right]\right\|_2^2 + E_D\left[\left\|E_D\left[\hat{\tilde{y}}_D\right]-\hat{\tilde{y}}_D\right\|_2^2\right] \qquad (36)
\end{aligned}
$$

As it is seen above, the same result as equation (35) is given mathematically. The complex model minimizes the second term due to better fitting the true model (f). Lastly, the third term increases with model complexity. Because, the fitting procedure is more sensitive to noise in the different data sets. The equation (36) that shows bias-variance decomposition needs to have the underlying model namely f. More mathematical computation including adding and subtracting $E_D\left[\vec{\tilde{y}}_D\right]$ gives the expression as:

$$
E_D\left[\left\|\hat{y}-\hat{\tilde{y}}_D\right\|_2^2\right] = \left\|\hat{y}-E_D\left[\hat{\tilde{y}}_D\right]\right\|_2^2 + E_D\left[\left\|E_D\left[\hat{\tilde{y}}_D\right]-\hat{\tilde{y}}_D\right\|_2^2\right] \qquad (37)
$$

This is another form of the bias-variance formula which does not need any information about the underlying model. Division by N gives MSE on the left-hand side.
The bias – variance tradeoff is also affected by model complexity, number of data points, training set and testing set. When we fit a function to our data and start by exploring a function with low complexity p, we might find that the bias is high and the variance relatively low (underfitting). This leads to the problem of the model ignoring important features of the training data. Conversely if p becomes too large, our function will follow random fluctuations in the data (overfitting), leading to an increase in variance and no important gains in bias. If we increase the complexity too much we'll reduce the bias and increase the variance [5]. A model with higher complexity is not able to predict the reasonable values for test data. Therefore, we strive to find the best balance between bias and variance. Number of data points will also affect this balance, too small data set and you're in the danger of underfitting. If this is the case, resampling methods can be applied. If we have a large data set we can apply other strategies to avoid the opposite problem, overfitting. One strategy is to split our data in training and test sets (sometimes training, test, and validation sets). A randomly selected portion of our data is used as training set, is used to fit the various candidate models. The remaining portion is used as test set to evaluate the performance of the available models and choose the most accurate one. This can be used to find the best value of p.

A.8 Mean Squared Error (MSE) and R^2 Score

In all regression methods, the value of the cost function is minimized, but its value is not necessarily meaningful. To be specific, the cost functions in Ridge and LASSO regression depend on the hyperparameter λ,and to determine the optimal λ one runs the evaluation process over a range of λ

values. There are other functions, which are used to interpret the performance of a given regression method.

Mean square error (MSE) which should be as small as possible and it is defined as:

$$\text{MSE}(\hat{\beta}) = \frac{1}{N}\left\|\hat{y} - \hat{\tilde{y}}\right\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2, \tag{38}$$

The value of **MSE** presents the quality of our prediction. As it is presented in equation (38), this value prospects the average of the square of the residuals. Apparently, from definition in equation (38), it could be realized that MSE can never be negative and lower MSE value means a better prediction. For regression evaluation the R^2 is considered as a golden standard to assess quality of fitting. R^2 score which is defined as:

$$\text{R}^2(\hat{\beta}) = 1 - \frac{\left\|\hat{y} - \hat{\tilde{y}}\right\|_2^2}{\|\hat{y} - \bar{y}\|_2^2} = 1 - \frac{\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}, \tag{39}$$

Here, y_i are the data we want to fit (indexed response variables), $\bar{y}$ is the mean of the measured values and $\tilde{y}_i$ is the predictor variables (from model). In addition, we define ϵ_i as $\epsilon_i = y_i - \tilde{y}_i$. As it is shown in equation (39), to calculate R^2, a fraction of **MSE** over **Variance** should be deduced from one. The average of the response variables is denoted $\bar{y}_i$. Let's interpret the formula step by step: if the residual sum of squares becomes low so, we have a good fit. But, it should be compared with the spread of response variables. After all, if the response variables are all nicely distributed close to the mean then getting a low value for residual sum of squares could not be impressive. In the other hand, if the model fits well, then residual sum of squares would be zero and the R^2 becomes one. In this sense we have a span of possible R^2 scores between zero and one. Therefore, the R^2 value shows us how good our model is at predicting future samples.

A.9 Features

Table of dataset features, sorted from most to least significant on the test accuracy.

PAY_0
PAY_6
PAY_2
PAY_AMT2
PAY_AMT6
PAY_AMT5
PAY_AMT3
PAY_3
PAY_AMT4
PAY_AMT1
LIMIT_B
PAY_5
PAY_4
AGE
BILL_AMT6
BILL_AMT1
BILL_AMT5
BILL_AMT2
GRAD SCHOOL
BILL_AMT3
BILL_AMT4
MARRIED
MALE
SINGLE
UNIVERSITY
HIGH SCHOOL
FEMALE
OTHER MARRIAGE STATUS
UNKNOWN EDUCATION 1
UNKNOWN EDUCATION 2
OTHER EDUCATIONS

References

[1] I.-C. Yeh, C.-h. Lien, *Expert Systems with Applications* **2009**, *36*, 2473–2480.

[2] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, D. J. Schwab, *Physics reports* **2019**.

[3] J. Friedman, T. Hastie, R. Tibshirani, *The elements of statistical learning*, *Vol. 1*, Springer series in statistics New York, **2001**.

[4] M. Hjorth-Jensen, Project 2, Department of Physics, University of Oslo, Norway, **2019**.

[5] A. Azzalini, B. Scarpa, *Data analysis and data mining: An introduction*, OUP USA, **2012**.

[6] A. C. Faul, A Concise Introduction to Numerical Analysis, University of Cambridge, UK, **2016**.

[7] J. Brownlee, *Machine Learning Mastery* **2017**.