EEE 485 FINAL PROJECT REPORT

Credit Card Default Estimation

Name/Surname: Alp Dursunoğlu ID: 22102196

Name/Surname: Muhammet Melih Çelik ID: 22003836

1.0 INTRODUCTION

In this project, a binary classification was made to estimate whether a person will have credit card default or not. The default status was indicated with binary 1 and non-default status was indicated with binary 0. For the project, a dataset which contains several features of customers and their payment history was used. The dataset can be accessed from the URL below:

https://www.kaggle.com/code/gpreda/default-of-credit-card-clients-predictive-models/input

Features of the dataset were shown in Table 1 as follows:

X1: LIMIT_BAL	Amount of given credit in dollars.				
X2: GENDER	Male (1)		Female (2)		
X3: EDUCATION	MSc. (1)	BSc. (2)	H.S. (3)	Others (4)	Unk. (5)
X4: MARRIAGE	Married (1)		Single (2)	Others (3)	
X5: AGE	Age in years				
X6: PAY_0	Repayment in September	(-1) on time or {1,2,3,4,5,6,7,8,9} delay			
X7: PAY_2	Repayment in August	(-1) on time or {1,2,3,4,5,6,7,8,9} delay			
X8: PAY_3	Repayment in July	(-1) on time or {1,2,3,4,5,6,7,8,9} delay			
X9: PAY_4	Repayment in June	(-1) on time or {1,2,3,4,5,6,7,8,9} delay			
X10: PAY_5	Repayment in May	(-1) on time or {1,2,3,4,5,6,7,8,9} delay			
X11: PAY_6	Repayment in April	(-1) on time or {1,2,3,4,5,6,7,8,9} delay			
X12: BILL_AMT1	Amount of bill statement in September				
X13: BILL_AMT2	Amount of bill statement in August				
X14: BILL_AMT3	Amount of bill statement in July				
X15: BILL_AMT4	Amount of bill statement in June				
X16: BILL_AMT5	Amount of bill statement in May				
X17: BILL_AMT6	Amount of bill statement in April				
X18: PAY_AMT1	Amount of previous payment in September				
X19: PAY_AMT2	Amount of previous payment in August				
X20: PAY_AMT3	Amount of previous payment in July				
X21: PAY_AMT4	Amount of previous payment in June				
X22: PAY_AMT5	Amount of previous payment in May				
X23: PAY_AMT6	Amount of previous payment in April				

Table 1: Features of the Dataset

Before making classification with the data, the correlation between predictors and the response was investigated to achieve reliable validation results. Results of the data analysis were used in the hyperparameter selection.

In the project, three different algorithms were used to make binary classification:

- Logistic Regression,
- Support Vector Machine,
- Shallow Neural Network.

The selected algorithms performances were validated by cross-validation. In the validation, k number of feature a model use was determined as the general hyperparameter in the project. In the end, the algorithms were ranked accordingly with their success rates.

2.0 DATA ANALYSIS

For the analysis, the correlation between each predictor $\underline{X}_j$ and the response vector $\underline{Y}$ was observed by the following empirical estimate formula:

$$\hat{R}_j = \frac{\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_{ij} - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{x}_j$ and $\bar{y}$ are mean of the vectors. $\hat{R}_j$ in the formula indicates the correlation coefficient. Thus, the features are ranked by the correlation coefficient score, s_j which is taken as square of the correlation coefficient $\hat{R}_j$. Simply:

$$s_j = (\hat{R}_j)^2$$

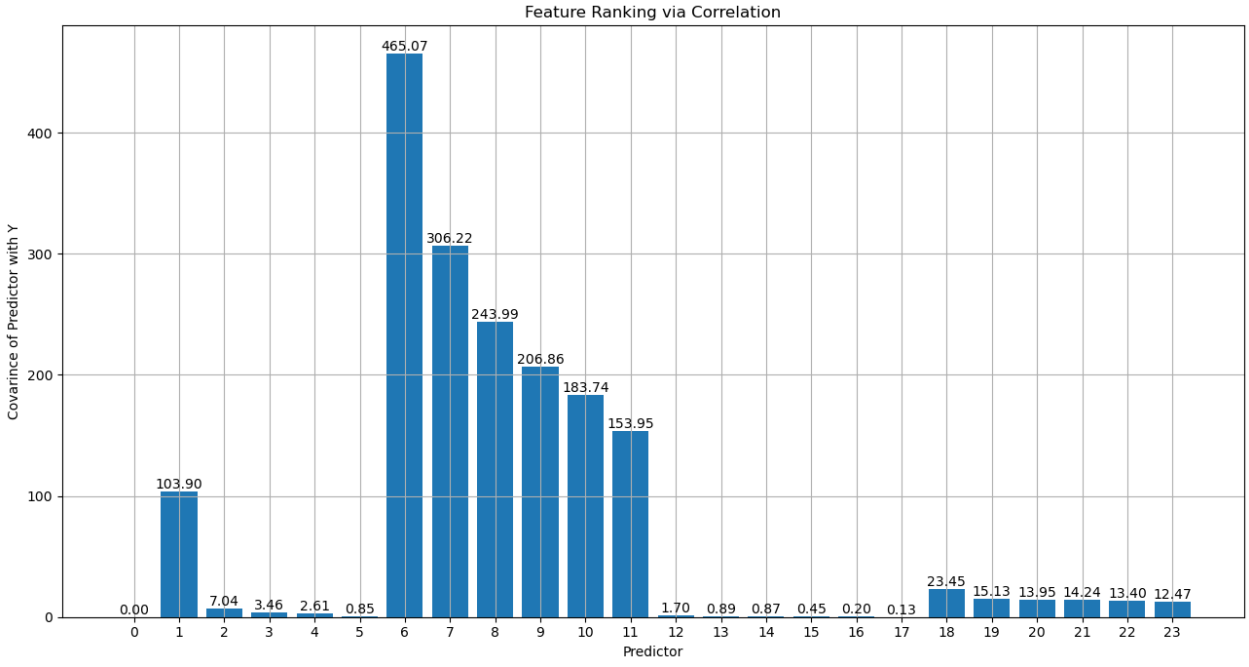

Fig.1: Correlation between Predictors and The Response

The result in Figure 1 was used during the validation of selected algorithms. The code for correlation result in Figure 1 can be seen in Appendix A.

3.0 ALGORITMS, IMPLEMENTATIONS AND RESULTS

2.1 Logistic Regression

The goal of the logistic regression is to model the probability. An input $\underline{x_i}$ belongs to one of the two classes based on one or more independent variables. The estimation of binary outcomes was done by fitting the data to a logistic curve, also known as the sigmoid function. The logistic function and the response vector $\underline{Y}$ were defined as:

$$\pi\left(\underline{x_i}\,;\underline{\beta}\right) = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_p x_p}}{1+e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_p x_p}}, \qquad \underline{y_i} \sim Bernoulli\left(\pi\left(\underline{x_i}\,;\underline{\beta}\right)\right)$$

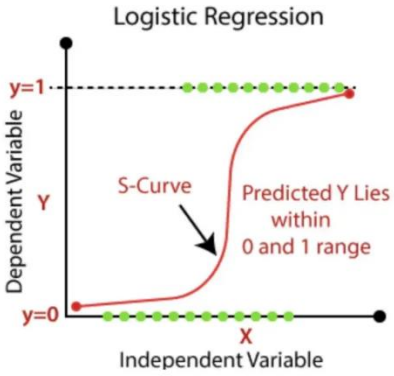

Fig.2: Logistic Regression

There are 23 number of features in the project. Thus, a multiple logistic model was used to obtain Beta coefficients. To obtain the Beta coefficients from the log-likelihood function Newton-Raphson and Gradient-Descent methods were used.

2.1.1 Newton-Raphson Method

Newton-Raphson method is a powerful iterative technique used to find successively better approximations to the roots (or zeroes) of a real-valued function. The method's procedure starts with an initial guess x_0 for the root of the function $f(x)$. This guess can be any value, but it's often chosen to be close to the actual root for faster convergence. After computing the tangent line to the function $f(x)$ for initial guess, the guess is updated with the point on which the x-axis and the tangent line intersects. Equation of this tangent line is given by:

$$y - f(x_0) = f'(x_0)(x - x_0)$$

where $f'(x_0)$ represents the derivative of $f(x)$ evaluated at x_0. The updated guess can be indicated with x_1. Same procedure is applied for the x_1. The iterative process continues until convergence. Therefore, the method can be summarized by the general formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

2.1.2 Implementation of Logistic Regression with Newton-Raphson Method

For the multiple logistic regression model, first, the key matrices were generated as:

$$\mathbf{X} = \begin{pmatrix} 1 & \cdots & \mathrm{X_{1p}} \\ \vdots & \ddots & \vdots \\ 1 & \cdots & \mathrm{X_{np}} \end{pmatrix}, \quad \underline{\mathrm{Y}} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_n \end{bmatrix} \quad \underline{\pi} = \begin{bmatrix} \pi(\underline{x}_1; \underline{\beta}_{old}) \\ \pi(\underline{x}_2; \underline{\beta}_{old}) \\ \vdots \\ \vdots \\ \pi(\underline{x}_n; \underline{\beta}_{old}) \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} \pi(\underline{x}_1; \underline{\beta}_{old})\left(1 - \pi(\underline{x}_1; \underline{\beta}_{old})\right) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \pi(\underline{x}_1; \underline{\beta}_{old})\left(1 - \pi(\underline{x}_n; \underline{\beta}_{old})\right) \end{bmatrix}.$$

In the model, number of samples (n) is equal to 30000 and number of features (p) is equal to 24. Initialization of the matrices were given in the Appendix B.

To use the Newton-Raphson formula the log-likelihood function and its derivatives were used in one single Newton-Step. The log-likelihood function, its first and second gradients were defined as [2]:

$$l\left(\underline{\beta}\right) = \sum_{i=1}^{N} \left\{ y_i \log\left(\pi\left(\underline{x}_i; \underline{\beta}\right)\right) + (1 - y_i \log\left(1 - \pi\left(\underline{x}_i; \underline{\beta}\right)\right)\right\}$$

$$= \sum_{i=1}^{n} \left\{ y_i \underline{\beta}^T \cdot \underline{x}_i - \log\left(1 + e^{\underline{\beta}^T \cdot \underline{x}_i}\right)\right\}$$

$$\nabla l\left(\underline{\beta}\right) = \sum_{i=1}^{n} \underline{x}_i (y_i - \pi\left(\underline{x}_i; \underline{\beta}\right) = \mathbf{X}^T(\underline{\mathrm{Y}} - \underline{\pi})$$

$$\nabla^2 l\left(\underline{\beta}\right) = -\sum_{i=1}^{n} \underline{x}_i \underline{x}_i^T \pi\left(\underline{x}_i; \underline{\beta}\right)\left(1 - \pi\left(\underline{x}_i; \underline{\beta}\right)\right) = -\mathbf{X}^T\mathbf{W}\mathbf{X}$$

Hence, one single step of Newton method is:

$$\underline{\beta_{\mathrm{new}}} = \underline{\beta_{\mathrm{old}}} + (\mathbf{X^T W X})^{-1}\mathbf{X^T}(\underline{\mathrm{Y}} - \underline{\pi})$$

Code for Newton-Raphson was given in the Appendix C.

2.1.2 Implementation Results of Logistic Regression

F1_Train	F1_Test	Threshold Value
0.356994194	0.367544623	0.1
0.356994194	0.367544623	0.2
0.357013748	0.367544623	0.3
0.357013748	0.367544623	0.4
0.357033304	0.36758463	0.5
0.357033304	0.367684686	0.55
0.357033304	0.36771838	0.6
0.357307313	0.37543559	0.7
0.384777456	0.431334245	0.8
0.050993181	0.011733646	0.9

Table 2: Logistic Regression Results

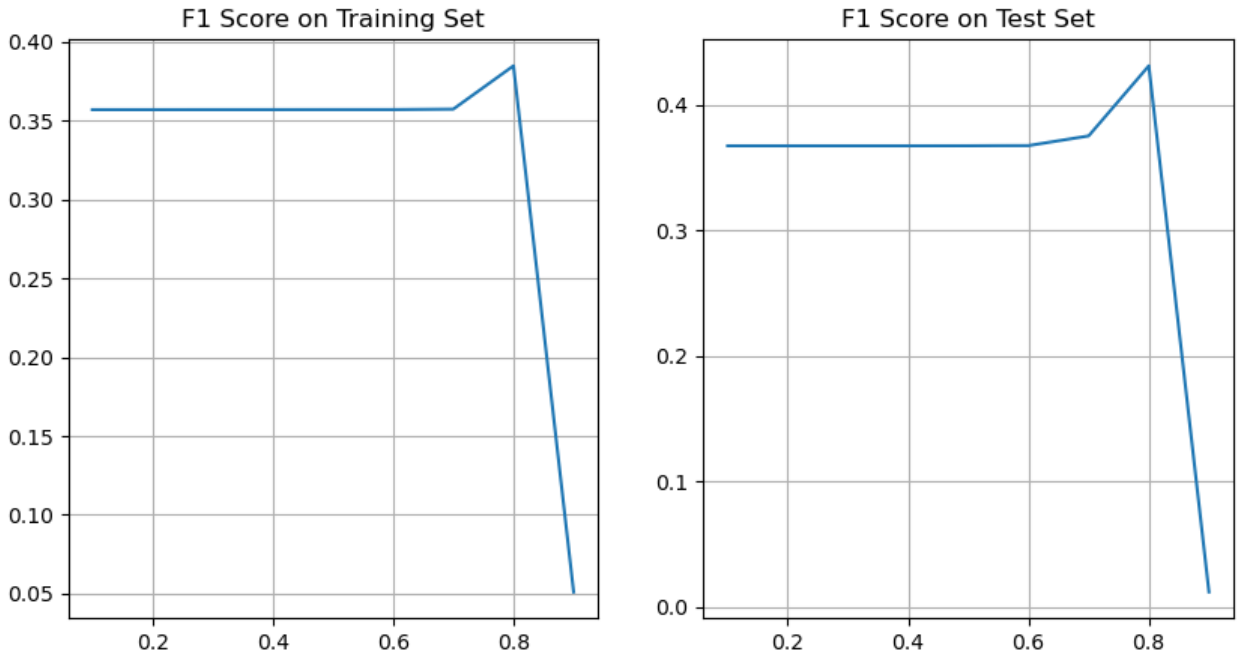

Fig.3: Logistic Result Graph

As can be seen from the validation, threshold value for binary classification was selected as 0.8 due to the maximum success rate at the value. It was observed that although there are little difference between test and train score, f1 scores on the test set are generally lower than the training set.

2.2 Support Vector Machine

Support Vector Machine (SVM in short) algorithm is a supervised learning method that can be used for regression and classification problems. In this project, we used SVM for classification purposes. When dealing with classification problems, SVM's aim is to find the best (optimal) hyperplane that separates the data points linearly regarding their classes. The hyperplane is positioned between the support vectors, the nearest data points of each class to the hyperplane. Then SVM tries to maximize this defined margin and achieves a better classification on the test data.

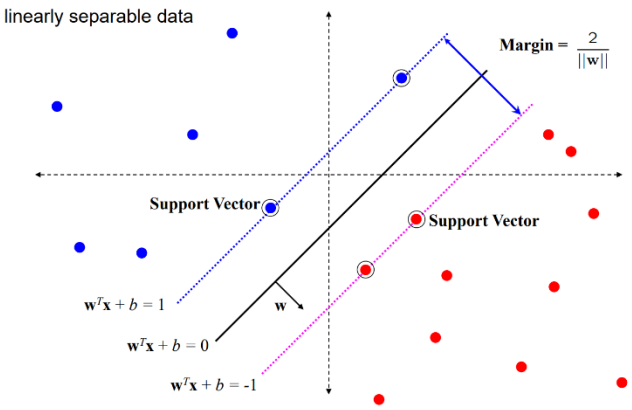

Fig.4: Visualization of the Support Vector Machine Model

The logic behind the SVM is using a linear line equation:

$$y = ax + b$$

Rearrangement gives:

$$ax + b - y = 0.$$

By defining vectors $\underline{X} = \begin{bmatrix} x \\ y \end{bmatrix}$ and $\underline{W} = \begin{bmatrix} a \\ -1 \end{bmatrix}$, we can write the equation $ax + b - y = 0$ as:

$$\underline{X} \cdot \underline{W} + b = 0 \ (*)$$

In the Figure 4, the optimal hyperplane lies on $(*)$. For given vectors $\underline{X}_i$ and $\underline{W}_j$, if the result of $(*)$ is bigger than zero, it is classified as blue class; otherwise, the datapoint is classified as red class. For estimating purposes, we need the matrices **X** and **W**. We already have the main design matrix X. So, we need to find **W**. In this project, the matrix **W** was found using the gradient descent technique.

2.2.1 Gradient Ascent/Descent in Support Vector Machine

Gradient ascent/descent is a recursive algorithm that aims to optimize the parameters of a machine learning model by updating itself until it converges. This algorithm is used to optimize parameters of many models such as Neural Networks and Support Vector Machine, which are selected models for the project [1].

The gradient of a continuous function *f* at point P is defined as the vector that contains the partial derivatives of the function computed at that point P [1]. The gradient descent algorithm aims to minimize the loss function with respect to weights by simply updating weights in every single iteration. The weights are updated in the direction of the gradient. When the minimum of loss function is achieved (which also means that optimal value for weight hyperparameter is achieved) the algorithm converges and stops working. In the project we have successfully achieved the optimal hyperparameter for our model. The mathematical formula for gradient descent is:

$$W_{n+1} = W_n - \gamma \frac{\delta L}{\delta W}$$

where γ denotes the learning rate.

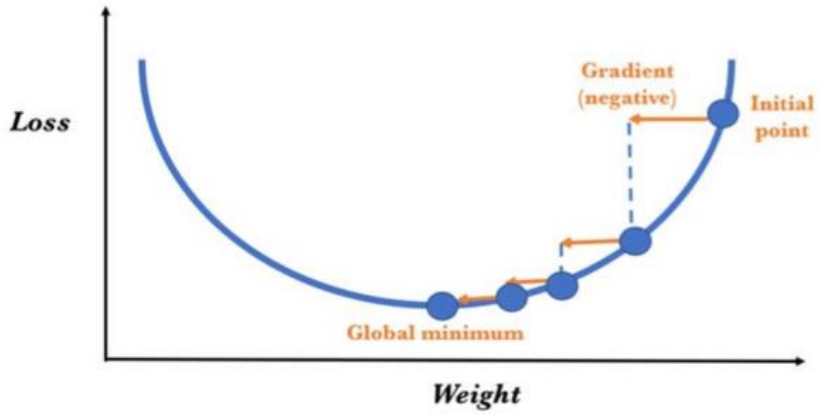

Fig.5: Visualization of the gradient descent method

The gradient ascent algorithm works very similarly to the gradient descent algorithm with only one difference. In gradient ascent, the aim is to maximize a function with respect to a hyperparameter, instead of minimizing it. The mathematical expression for the gradient ascent algorithm is:

$$W_{n+1} = W_n - \gamma \frac{\delta L}{\delta W}$$

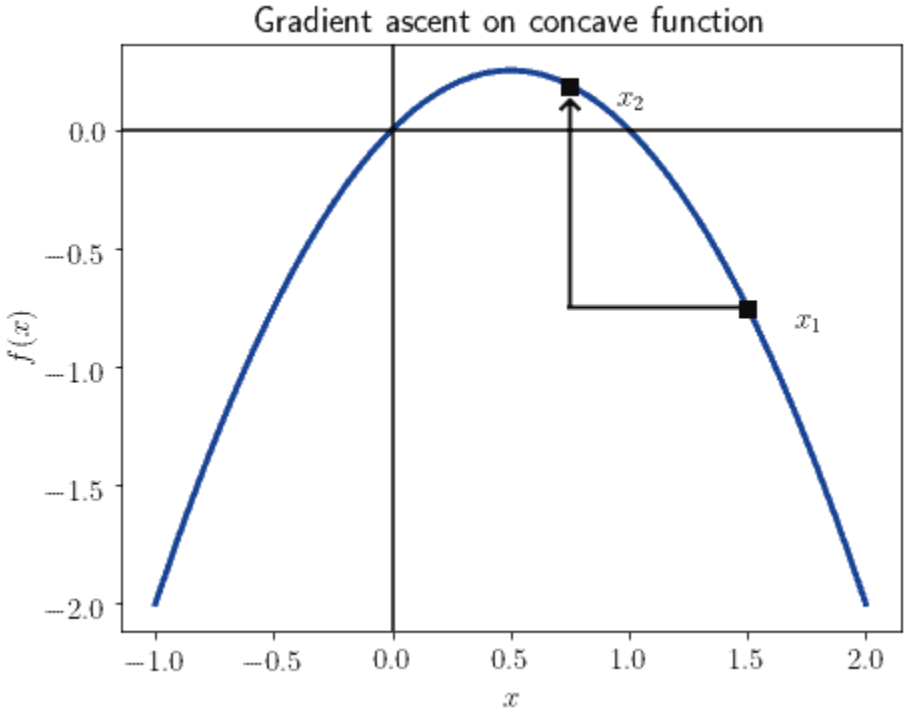

Fig.6: Visualization of the gradient ascent method

2.2.2 Gradient Descent for Support Vector Machine

The margin between support vectors can be expressed with $\frac{2}{w}$. Our goal is to maximize the margin, namely, we need to maximize $\frac{1}{||2w||}$ that is subject to constraints

$$y_i(w \cdot x_i + b) \geq 1 \; \forall \; i$$

To do that, this problem can be transformed to an unconstrained optimization problem using the Lagrange Multipliers method. The Lagrangian equation for this problem is:

$$L(w, b, \alpha) = \frac{1}{2} \parallel w \parallel^2 - \sum_{i=1}^{N} \alpha_i [y_i(w \cdot x_i + b) - 1]$$

Taking the derivative with respect to lambda and solving for (w) gives us the gradient:

$$\nabla_w L = \frac{\delta L}{\delta w} = \sum_{i=1}^{N} \alpha_i \, y_i x_i$$

And the mathematical expression for the gradient ascent algorithm is:

$$w_{n+1} = w_n - \gamma \sum_{i=1}^{N} \alpha_i \, y_i x_i$$

2.2.3 Implementation Results for Support Vector Machine

The Support Vector Machine Algorithm was implemented in Python3 programming language by using the gradient descent optimization technique. After the hyperparameter tuning C in the model and obtained feedback from the first demo, the accuracy of the SVM model achieves %81. The accuracy versus hyperparameter can be seen in the Figure 7.

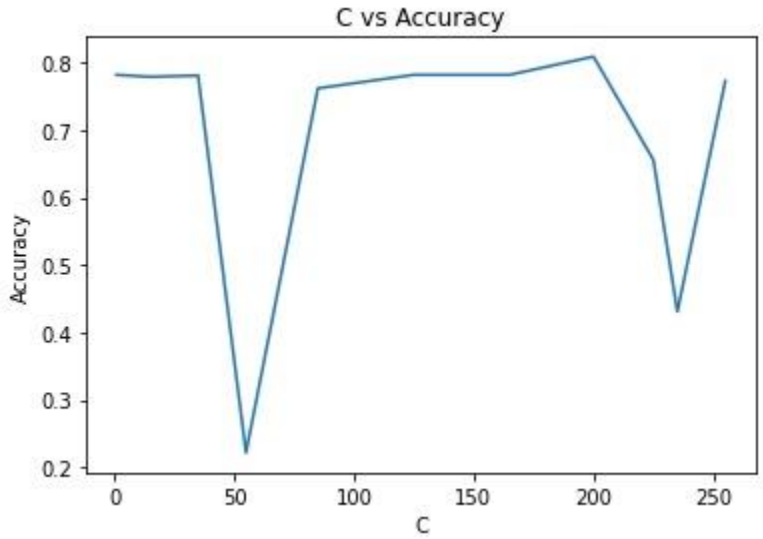

Fig.7: SVM Accuracy Result

Code for the SVM model can be seen in the Appendix D.

2.3 Shallow Neural Network

Shallow neural network is a type of neural network which only consist of input layer, output layer and one single hidden layer [towards]. The input layer receives the data, the hidden layer processes it, and finally, the output layer produces the output. Shallow neural networks are simpler implementations compared to deep neural networks. Additionally, they are easily trained and they work more efficiently. Shallow networks are mostly used for simpler tasks such as linear regression, binary classification, or low-dimensional feature extraction.

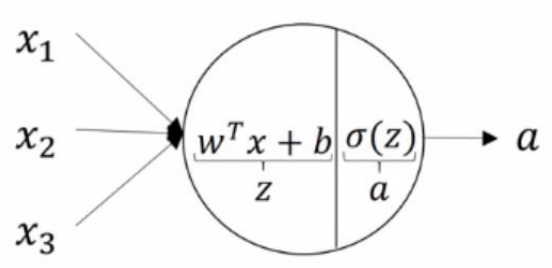

Fig.8: Visualization of a Neuron

As the name stands, neural networks are built by neurons. Given an input, the neurons calculate the output and passes it to the next layer in process. In the figure below, it is seen that the first part, Z, is the calculated output in the form of $w^T x + b$ where w denotes weights, and the second part, α, is the activation part to pass the activated output of the neuron. The activated output is calculated by with the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$. So, the output, $A = \sigma(z) = \frac{1}{1+e^{-(w^T x+b)}}$.

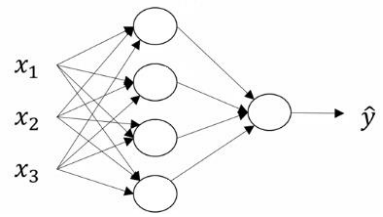

Fig.9: Visualization of a Shallow Neural Network with a Single Hidden Layer

The weights of the neural network are initialized randomly. If they all are to be assigned the same value, their activations would be the same. This would result with their derivatives being the same and afterwards, the neurons would modify the weights in the same way. This would make no sense to have more than a neuron in a layer. So, the weights are picked randomly from a zero mean, unit variance normal distribution. And the bias term, *b,* should be initialized as zero.

2.3.1 Gradient Descent for Shallow Neural Network

In order to acquire accurate predictions, the initialized weights of the neural network have to be modified. The weights can be modified with the use of Gradient Descent algorithm, just like in Support Vector Machine model. The loss function is calculated as:

$$L(\hat{y}, y) = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})]$$

From the loss function, the derivatives of all parameters can be calculated to be used in the gradient descent [3]:

$$dA^{[2]} = \frac{\delta L\left(A^{[2]},Y\right)}{\delta A^{[2]}} = \frac{-Y}{A^{[2]}} + \frac{1-Y}{1-A^{[2]}}$$

$$dZ^{[2]} = \frac{\delta L\left(A^{[2]},y\right)}{\delta Z^{[2]}} = \frac{\delta L\left(A^{[2]},y\right)}{\delta A^{[2]}} * \frac{\delta A^{[2]}}{\delta Z^{[2]}} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{\delta L\left(A^{[2]},y\right)}{\delta W^{[2]}} = \frac{\delta L\left(A^{[2]},y\right)}{\delta Z^{[2]}} * \frac{\delta Z^{[2]}}{\delta W^{[2]}} = dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{\delta L\left(A^{[2]},y\right)}{\delta b^{[2]}} = \frac{\delta L\left(A^{[2]},y\right)}{\delta Z^{[2]}} * \frac{\delta Z^{[2]}}{\delta b^{[2]}} = dZ^{[2]}$$

$$dA^{[1]} = \frac{\delta L\left(A^{[2]},Y\right)}{\delta A^{[1]}} = \frac{\delta L\left(A^{[2]},Y\right)}{\delta Z^{[2]}} * \frac{\delta Z^{[2]}}{\delta A^{[1]}} = dZ^{[2]} W^{[2]}$$

$$dZ^{[1]} = \frac{\delta L\left(A^{[2]},y\right)}{\delta Z^{[1]}} = \frac{\delta L\left(A^{[2]},y\right)}{\delta A^{[1]}} * \frac{\delta A^{[1]}}{\delta Z^{[1]}} = W^{[2]T} dZ^{[2]} * \sigma'\left(Z^{[1]}\right)$$

$$dW^{[1]} = \frac{\delta L\left(A^{[2]},y\right)}{\delta W^{[1]}} = \frac{\delta L\left(A^{[2]},y\right)}{\delta Z^{[1]}} * \frac{\delta Z^{[1]}}{\delta W^{[1]}} = dZ^{[1]} X^T$$

$$db^{[1]} = \frac{\delta L\left(A^{[2]},y\right)}{\delta b^{[1]}} = \frac{\delta L\left(A^{[2]},y\right)}{\delta Z^{[1]}} * \frac{\delta Z^{[1]}}{\delta b^{[1]}} = dZ^{[1]}$$

Fig.10: Derivative Table for Parameters [3]

Weights should be updated in order to acquire minimum loss. The weights will be updated with respect to the classical gradient descent formula:

$$W_{new} = W_{old} - \gamma \frac{dL}{dW}$$

2.3.2 Implementation Result Shallow Neural Network

Implementation results were given in table 1 for Shallow Neural Network.

Accuracy	0.8389333
Number of Negative	11741
Number of Positive	3259
True Negatives	10938
True Positives	1646
False Negatives	1613
False Positives	797
precision	0.673761768
recall	0.505062903
F1_Score	1.33

Table 3:Shallow Nueral Network Results

As can be seen from the Table 3, Neural Network implementation has the highest accuracy and F1 score among the selected algorithms. Code for Neural Network implementation can be seen in Appendix E.

4.0 CONCLUSION

In this project, a binary classification were made via selected three algorithms: Logistic Regression, Support Vector Machine and Shallow Neural Network. It can be seen from the implementation results; Neural Network implementation was the most successful one with the %83 accuracy and 1.33 F1 score. While SVM model gave results as good as Neural Network, Logistic Regression is the worst model among the algorithms with 0.38 and 0.43 F1 scores.

APPENDICES

Appendix A: Python-Jupyter Cell for Correlation between Predictors $\underline{X}_j$ and $\underline{Y}$.

```
#Feature Ranking Via Correlation

feature_scores = np.empty((24))


for i in range(0,24):

    column = X[:,i]
    column = np.reshape(column,(X.shape[0],1))
    response = np.reshape(Y,(X.shape[0],1))

    mean_of_response = np.mean(response)
    mean_of_column = np.mean(column)

    var_of_response = np.var(response)
    var_of_column = np.var(column)

    tilda_column = column - mean_of_column
    tilda_response = response - mean_of_response

    tilda_column_T = np.transpose(column)
    tilda_response_T = np.transpose(response)

    correlation_coeff = (np.dot(tilda_column_T,tilda_response))/\
        (np.sqrt(np.dot(tilda_column_T,tilda_column) +
np.dot(tilda_response_T,tilda_response)))

    correlation_coeff_scalar = correlation_coeff[0,0]

    feature_scores[i] = np.square(correlation_coeff_scalar)

index_array = np.arange(0,24)

plt.figure(figsize=(13, 7))
plt.bar(index_array, feature_scores)
plt.xlabel('Predictor')
plt.ylabel('Covarince of Predictor with Y')
plt.title('Feature Ranking via Correlation')

plt.xticks(index_array)

for i, value in enumerate(feature_scores):
    if value > 0:
        plt.text(i, value, f'{value:.2f}', ha='center', va='bottom')
    elif value < 0:
        plt.text(i, value, f'{value:.2f}', ha='center', va='top')

plt.grid(True)
plt.tight_layout()
plt.show()
```

Appendix B:

Python-Jupyter Cells for Initialization of **X**, $\underline{Y}$ and $\underline{\beta}$ Matrices in Logistic Regression

```
#EEE 485: Statsitical Leaning and Data Analytics Term Project#Logistic Regression Algorithm
Implementation

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
import time
file_path = "UCI_Credit_Card.csv"
data = pd.read_csv(file_path)
print(data)
column_names = data.columns
print(column_names)
```

```
ones = np.ones(30000)

X0 = np.reshape(ones,(30000,1))
X1 =  data['LIMIT_BAL'].values #
X2 =  data['SEX'].values
X3 =  data['EDUCATION'].values
X4 =  data['MARRIAGE'].values
X5 =  data['AGE'].values       #
X6 =  data['PAY_0'].values
X7 =  data['PAY_2'].values
X8 =  data['PAY_3'].values
X9 =  data['PAY_4'].values
X10 = data['PAY_5'].values
X11 = data['PAY_6'].values
X12 = data['BILL_AMT1'].values #
X13 = data['BILL_AMT2'].values #
X14 = data['BILL_AMT3'].values #
X15 = data['BILL_AMT4'].values #
X16 = data['BILL_AMT5'].values #
X17 = data['BILL_AMT6'].values #
X18 = data['PAY_AMT1'].values  #
X19 = data['PAY_AMT2'].values  #
X20 = data['PAY_AMT3'].values  #
X21 = data['PAY_AMT4'].values  #
X22 = data['PAY_AMT5'].values  #
X23 = data['PAY_AMT6'].values  #

main_design_matrix =
np.column_stack((X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23)
)
main_response_vector = data['default.payment.next.month'].values
```

```
def normalize(predictor):

    mean_value = np.mean(predictor)
    std_variance = np.std(predictor)
    normalized = (predictor - mean_value)/std_variance
    return normalized
```

```
def matrix_generator(ones,main_design_matrix,main_response_vector):

    #number of row
    n = int(input("Enter row size: "))
    #whether to take whole main matrix or not
    matrix_bool = int(input("All features?(Yes:1/No:0): "))
    ones = np.ones(n)
    X0 = np.reshape(ones,(n,1))
    X = X0
    if matrix_bool != 1:
        p = input("Enter predictor numbers: ")
        index_lst = p.split(",")
        int_lst  = [int(num) for num in index_lst]
        print(int_lst)
        Beta = np.zeros((len(int_lst)+1,1))

        for i in range(len(int_lst)):

            normalized_colum = normalize(main_design_matrix[0:n,int_lst[i]])
            X = np.column_stack((X,normalized_colum))
    else:
        int_lst = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]
        print(int_lst)
        Beta = np.zeros((len(int_lst)+1,1))

        for i in range(len(int_lst)):

            normalized_colum = normalize(main_design_matrix[0:n,int_lst[i]])
            X = np.column_stack((X,normalized_colum))

    Y = main_response_vector[0:n]
    Y = np.reshape(Y,(n,1))



    print(Y.shape)
    print(X.shape)
    print(Beta.shape)

    return Y,X,Beta
```

Appendix C: Python-Jupyter Cell for Newton Raphson Method

```
def newton_raphson(Y_vector,X_matrix,Beta_vector):

    error = 0.001
    ones = np.ones(X_matrix.shape[0])
    ones = np.reshape(ones,(X_matrix.shape[0],1))
    Beta_new = Beta_vector
    Beta_old = Beta_vector

    max_iteration = 20
    count = 0

    while count != max_iteration:

        # Calculate Pi_vector
        Pi_vector = 1/ (1 + np.exp(-np.dot(X_matrix, Beta_old)))
        Pi_complement = ones - Pi_vector

        # Update W_matrix
        W_matrix = np.eye(X_matrix.shape[0])
        diagonal_indices = np.diag_indices_from(W_matrix)
        diagonal_values = np.multiply(Pi_vector, Pi_complement)
        diagonal_values = diagonal_values.flatten()
        W_matrix[diagonal_indices] = diagonal_values

        # Update Beta_new using Newton-Raphson update rule
        X_t = np.transpose(X_matrix)
        A = np.dot(np.dot(X_t, W_matrix), X_matrix)
        A_inv = np.linalg.inv(A)

        Beta_new = Beta_old + np.dot(np.dot(A_inv, X_t), (Y_vector -
Pi_vector))

        if np.linalg.norm(Beta_new - Beta_old) < error:
            break

        Beta_old = Beta_new
        count += 1

    print(count)
    return Beta_new
```

Appendix D: Python Code for SVM MODEL

```
"""
SUPPORT VECTOR MACHINE
WORK IN PROGRESS.............
"""
import numpy as np
import random
import math
import Logistic_Regression_Preprocess as LRP
import pandas as pd

def data_to_matrix_function():
    #this function will take the raw data and column size
    #and will return the main X decision matrix as an output.

    raw_data = open("UCI_Credit_Card.csv",'r')
    #the raw data obtained from kaggle.com as below:
    #https://www.kaggle.com/datasets/uciml/default-of-credit-card-clients-
dataset?resource=download

    column_size = int(input("Specified column size to be imported: "))
    #column size will be input for model training with X which will have variying
column size
    #there are 23 number of fixed features.
    #X1,X2,X3,X4,...,X23.
    #decision_matrix_function(raw_data, column_size)

    predictor_matrix = []
    response_vector = []

    for i in range(1,column_size+2):                        #first two line are
predictor names.

        column_vector_str = raw_data.readline()
        """
        try:
            first = column_vector_str[0] #if a line starts with a number it will
            first = int(first)           #it will be added as column to the
matrix.

            column_vector_lst = column_vector_str.split(',') #splitting the line
                                                             # to add it to a
column vector.

            column_vector_lst[-1] = column_vector_lst[0:-2]  #last element is \n
```

```
                                                                # which is removed.
            column_vector_int_lst =
[]
            for i2 in range(0,len(column_vector_lst)-1):
                 column_vector_int_lst.append(int(column_vector_lst[i2]))

            X.append(column_vector_int_lst)

        except ValueError:
            continue
        """

        #due to the integer-float-string conversion the code above generates
        #matrix with column dimension less than the input value.

        if column_vector_str[0].isdigit():
            column_vector_lst = column_vector_str.split(',')
            column_vector_lst[-1] = column_vector_lst[-1][0:1]
            response_vector.append(int(column_vector_lst[-1]))

            column_vector_float_lst =
[1]
            for i2 in range(1,len(column_vector_lst)-1): # last value is the y
value of the regression.
                 column_vector_float_lst.append(float(column_vector_lst[i2]))

            predictor_matrix.append(column_vector_float_lst)

        else:
            continue


    #NORMALIZATION

    #print(np.mean(response_vector))
    # response_vector=random.shuffle(response_vector)
    # predictor_matrix=random.shuffle(predictor_matrix)
    trainset_length=math.floor(len(predictor_matrix)*0.7)

    X_test=predictor_matrix[trainset_length:]
    X_train=predictor_matrix[0:trainset_length]
    Y_test=response_vector[trainset_length:]
    Y_train=response_vector[:trainset_length]
    return predictor_matrix, response_vector, X_test, X_train, Y_test, Y_train
```

```
# def gradient_descent(X,Y,C,learning_rate,tolerance,iterations):

#     W=np.zeros(len(X[0]))

#     for i in range (0,iterations):
#         gradient=np.zeros(len(X[0]))

#         for k in range(0,len(X)):
#             if Y[k] * np.dot(W, X[k])<1:

#                 gradient+=C*Y[k]*X[k]


#         W=W-learning_rate*gradient
#         if np.all(np.abs(gradient*learning_rate) <= tolerance):
#             break


#     return W



# def
Support_Vector_Machine(X_Train,X_Test,Y_Train,iterations,tolerance,learning_rate,
C):

#     Y_Train2=convert_Y(Y_Train)

#     print(np.mean(X_Train))
#     X_Train = np.insert(X_Train,0,2,axis = 1)
#     X_Test = np.insert(X_Test,0,2,axis = 1)


#     W=gradient_descent(X_Train,Y_Train2,C,learning_rate,tolerance,iterations)

#     training_results=[]
#     test_results=[]




#  #9000000000000000
#     for X1 in X_Train:
```

```
#         if np.dot(W,X1)>20000:
#             #print(np.dot(W,X1))
#             training_results.append(1)
#         else:
#             training_results.append(0)

#     for X2 in X_Test:
#         if np.dot(W,X2)>20000:
#             test_results.append(1)
#         else:
#             test_results.append(0)


#     return training_results,test_results

# iterations=200
# tolerance=5e-06
# learning_rate=0.0002
# C=45

def convert_Y(Y_Train):
    for k in range (len(Y_Train)):
        if Y_Train[k]==0:
            Y_Train[k]=-1
    return Y_Train



# stochastic gradient descent
def gradient_descent(iterations, learning_rate, X, Y, C, tolerance):

    weights = np.zeros(24)
    for k in range(0, iterations):


        gradient = 0
        for i in range(0, len(X)):
            if np.dot(weights, X[i])*Y[i] < 1.0:
                gradient=gradient+(C*Y[i]*X[i])
        gradient=weights-gradient

        if np.any(np.abs(gradient * learning_rate) <= tolerance):
            return weights

        else:
```

```
            weights = weights - (gradient * learning_rate)
    return weights

# def svm_predict(X,W):

#     return results


iterations = 1000
learning_rate = 0.0002
tolerance = 5e-07


def Support_Vector_Machine(X_train, Y_train, X_test, C, iterations,
learning_rate, tolerance):
    for yi in range(len(Y_train)):
        if Y_train[yi] == 0:
            Y_train[yi] = -1

    X_train = np.insert(X_train,0,1,axis = 1)
    X_test = np.insert(X_test,0,1,axis = 1)

    weights = gradient_descent(iterations, learning_rate, X_train, Y_train, C,
tolerance)


    # train_results=-1*np.ones(len(X_train))
    # for Xi_train in range(len(X_train)):

    #     if  np.dot(weights,X_train[Xi_train]) >0:
    #         train_results[Xi_train]=1


    results = -1*np.ones(len(X_test))
    for Xi in range(len(X_test)):

        if  np.dot(weights,X_test[Xi]) >0:
            results[Xi]=1
    return results
    return train_results, results



dt = pd.read_csv('UCI_Credit_Card.csv')
```

```
dt.columns =
["1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","17","18"
,"19","20","21","22","23","24","y"]
X = np.array(dt)[:,1:24]
Y = np.array(dt)[:,24]

Y=convert_Y(Y)
columns=[0,4,11,12,13,14,15,16,17,18,19,20,21,22]
for xi in columns:

    mean=np.mean(X[:,xi])
    std=np.std(X[:,xi])
    X[:,xi]=(X[:,xi]-mean)/std


X_Train=X[0:15000]
X_Test=X[15000:30000]
Y_Train=Y[15000:30000]
Y_Test=Y[15000:30000]

#C=150

#c_list=[1,15,35,55,85,125,165,200,225,235,255]
c_list=[200]
accuracylist=[]


for C in c_list:
    test_results=Support_Vector_Machine(X_Train,Y_Train,X_Test,C,iterations,learn
ing_rate,tolerance)



    summm=0
    summm2=0

    for k in range(len(test_results)):
        if test_results[k]==Y_Test[k]:
            summm2+=1
            if Y_Test[k]==1:
                summm+=1

    false_positive = 0
    false_negative = 0
    true_positive = 0
```

```
    true_negative = 0

    for i in range(0,len(test_results)):
        if (test_results[i] == Y_Test[i]) and (Y_Test[i] == 1):
            true_positive += 1
        elif (test_results[i] == Y_Test[i]) and (Y_Test[i] == -1):
            true_negative += 1
        elif (test_results[i] != Y_Test[i]) and (Y_Test[i] == 1):
            false_positive += 1
        elif (test_results[i] != Y_Test[i]) and (Y_Test[i] == -1):
            false_negative += 1


    #print(training_results)

    #print(test_results)
    #print(Y_Test)
    #print(summm2/len(test_results))
    #print(summm2/summm)

    accuracy = ((true_positive+true_negative)/len(test_results))

    # print("False Negatives:",false_negative)
    # print("False Positives:",false_positive)

    # print("True Negatives:",true_negative)
    # print("True Positives:",true_positive)

    print("Accuracy for c:",C,"   ",accuracy,"    ","True
Positives:",true_positive,"True Negatives:",true_negative)
```

Appendix E: Code Shallow Neural Network Model

```
import numpy as np
import math
import random
import pandas as pd


dt = pd.read_csv('UCI_Credit_Card.csv')

dt.columns =
["1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","17","18"
,"19","20","21","22","23","24","y"]
X = np.array(dt)[:,1:24]
Y = np.array(dt)[:,24]


columns=[0,4,11,12,13,14,15,16,17,18,19,20,21,22]
for xi in columns:

    mean=np.mean(X[:,xi])
    std=np.std(X[:,xi])
    X[:,xi]=(X[:,xi]-mean)/std


X_Train=X[0:50]
X_Test=X[50:100]
Y_Train=Y[0:50]
Y_Test=Y[50:100]

# dt = pd.read_csv('CKD_Preprocessed.csv')

# #shuffle all rows
# dt = dt.sample(frac = 1)

# dt.columns =
["1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","17","18"
,"19","20","21","22","23","24","y"]
# matrix = np.array(dt)
# X=matrix[:,0:24]
# Y= matrix[:,24]
"""
shuffle_list =  list(zip(X,Y))
random.shuffle(shuffle_list)
X,Y=zip(*shuffle_list)
Y=np.array(Y)
```

```
X=np.array(X)
"""
# columns=[0,1,2,5,6,7,8,9,10,11,12,13]
# for column in columns:
#     X[:,column]=X[:,column]-np.mean(X[:,column])
#     X[:,column]=X[:,column]/np.std(X[:,column])
#     #print("okan bok kokan")


# X_Train = (X[0:200])
# X_Test = (X[200:400])
# Y_Train = Y[0:200]
# Y_Test = Y[200:400]



def sigmoid(z):
    return 1/(1+np.exp(-1*z))

def sigmoid_derivative(z):
    return (1-sigmoid(z))*sigmoid(z)

def pred(X,input_hidden_weights,hidden_output_weights):
    return 1 if
forward_propagation(X,input_hidden_weights,hidden_output_weights)[0][0][0]>=0.5
else 0

def mse_loss(y_true, y_pred):
    return np.mean((y_true-y_pred)**2)

def init_weights(sizeof_input, sizeof_output):

    #Xavier initialization for weights of a neural network layer.
    weights = np.random.normal(0, math.sqrt(2/(sizeof_input+sizeof_output)),
(sizeof_input, sizeof_output))
    return weights

input_size = 23
hidden_layer_size = 15
output_size = 1

# Initialize weights
input_hidden_weights = init_weights(input_size, hidden_layer_size)
hidden_output_weights = init_weights(hidden_layer_size, output_size)
```

```
def forward_propagation(X,input_hidden_weights,hidden_output_weights):
    results_hidden_layer=[]
    final_out=[]

    for k in range (len(X)):
        input_hidden_layer=0
        for i in range(len(X[0])):
            input_hidden_layer+=np.dot(X[k][i],(input_hidden_weights[i]))
        result_hidden_layer=(sigmoid(input_hidden_layer))
        #sigmoid is for activation
        results_hidden_layer.append(result_hidden_layer)
        input_output=np.dot(result_hidden_layer,hidden_output_weights)

        final_out.append(sigmoid(input_output))
    #return final_out,result_hidden_layer
    final_out=np.array(final_out)

    results_hidden_layer=np.array(results_hidden_layer)

    return final_out,results_hidden_layer

def average_of_lists(*lists):
    # Initialize a list to store the averages
    averages = []

    # Iterate through the lists
    for values in zip(*lists):
        # Calculate the average of corresponding elements
        avg = sum(values) / len(values)
        averages.append(avg)

    return averages


def backpropagation(X, y_true, input_hidden_weights, hidden_output_weights,
learning_rate, epochs):

    losses=[]
    for epoch in range(epochs):
        # Calculate output
        (y_pred,hidden_layer_output) = forward_propagation(X,
input_hidden_weights,hidden_output_weights)
```

```
        # Calculate loss
        loss = mse_loss(y_true, y_pred)
        losses.append(loss)
        if len(losses)>2:
            if losses[-1]>losses[-2]:
                return hidden_output_weights,input_hidden_weights
        # Backrpropagation
        output_error = y_true - y_pred
        output_delta = output_error * sigmoid_derivative(y_pred)
        #transpose may be needed

        hidden_error = np.dot(output_delta,hidden_output_weights.T)

        hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)
        #print(hidden_layer_output)
        # Update weights (transpose may be needed)

        hidden_output_weights=np.add(hidden_output_weights,
np.dot(hidden_layer_output.T,output_delta) * learning_rate)

        input_hidden_weights=np.add(input_hidden_weights,
np.dot(X.T,hidden_delta) * learning_rate)
        newloss=loss

        # if epoch % 100 == 0:
        #     print(f"Epoch {epoch}, Loss: {loss}")

    return hidden_output_weights,input_hidden_weights

#find weights avrg
learning_rate=0.002
epochs=1000
hidden_output_weights_list=[]
input_hidden_weights_list=[]

for k in range (len(X_Train)):
    X_Train_1=np.array([X_Train[k]])
    Y_Train_1=np.array([Y_Train[k]])
    hidden_output_weights,input_hidden_weights=backpropagation(X_Train_1,
Y_Train_1, input_hidden_weights, hidden_output_weights, learning_rate, epochs)
    hidden_output_weights_list.append(hidden_output_weights)
    input_hidden_weights_list.append(input_hidden_weights)

def sum_and_average_arrays(*arrays):
```

```
    # Convert the list of arrays into a numpy array
    stacked_arrays = np.stack(arrays, axis=0)
    # Sum along the first axis
    summed_array = np.sum(stacked_arrays, axis=0)
    # Calculate the average
    average_array = summed_array / len(arrays)

    return average_array



def Shallow_Neural_Network(X_Test,input_hidden_weights,hidden_output_weights):

    test_predictions = []
    for r in range(len(X_Test)):
        prediction =
forward_propagation(X_Test,input_hidden_weights,hidden_output_weights)[0][0][0]
        print(prediction)
        if prediction >0.3:
            test_predictions.append(1)
        else:
            test_predictions.append(0)
    return test_predictions







resultlist=[]
for k in range (len(X_Test)):
    X_Train_1=np.array([X_Train[k]])
    #X_Train_2=np.array([X_Train[1]])
    X_Test_1=np.array([X_Test[k]])
    Y_Train_1=np.array([Y_Train[k]])
    results=Shallow_Neural_Network(X_Train_1,sum_and_average_arrays(*input_hidden
_weights_list),sum_and_average_arrays(*hidden_output_weights_list))
    #results=Shallow_Neural_Network(X_Test_1,input_hidden_weights_list[k],hidden_
output_weights_list[k])
    resultlist.append(results)


correct=0
true_negative=0
true_positive=0
false_negative=0
```

```
false_positive=0
negative=0
positive=0


for e in range(0,len(Y_Train)):
    if Y_Train[e]==1:
        positive+=1
    else:
        negative+=1

for i in range(0,len(resultlist)):
        #print(resultlist[i][0],Y_Train[i])
        if (resultlist[i][0] == Y_Train[i]) and (resultlist[i][0] == 1):
            true_positive += 1
        elif (resultlist[i][0] == Y_Train[i]) and (resultlist[i][0] == 0):
            true_negative += 1
        elif (resultlist[i][0] != Y_Train[i]) and (resultlist[i][0] == 1):
            false_positive += 1
        elif (resultlist[i][0] != Y_Train[i]) and (resultlist[i][0] == 0):
            false_negative += 1


print(resultlist)

accuracy=(true_negative+true_positive)/len(Y_Test)

print(accuracy)
print("Negative:",negative,"Positive:",positive,"True
Negatives:",true_negative,"True Positives:",true_positive,"False
Negatives:",false_negative,"False Positives:",false_positive)
#print(hidden_output_weights)
#print(forward_propagation(X_Train,input_hidden_weights,hidden_output_weights)[0]
[0][0])
```