

University of New Haven
Tagliatela college of Engineering
Data Science, Computer Engineering and Computer Science



LINEAR REGRESSION
AND
ARTIFICIAL NEURAL NETWORKS

MATH FOR DATA SCIENTISTS

DSCI – 6601

Date: 12/16/2020

By,

Namitha Nagaraju

Supervisor,

Minkyu Kim

1. INTRODUCTION

Machine learning is an application of artificial intelligence (AI) that provides computers the ability to learn and improve from experience without being explicitly programmed. Machine Learning are broadly classified into three types -Supervised Learning, Unsupervised Learning and Reinforcement Learning. Supervised Learning is a branch of Machine Learning where past data is used to predict new data.

This project aims at developing two Supervised Machine Learning Models- Linear Regression and Artificial Neural Networks from scratch using Python and NumPy. The code developed has various parameters that can be tuned. The model accuracy has been tested by altering various parameters for both Linear Regression and Artificial Neural Networks. The best models have been selected for the given datasets. For new datasets, the project enables the users to tune the parameters and select the best model to train their datasets.

2. LINEAR REGRESSION

2.a Theory

Linear regression is a regression model that assumes a linear relationship between the input variables and the output variables. A simple linear model can be represented as follows. Here the input is a single variable x and output, y is also a single dimensional value. m and c are the parameters that can be tuned to fit the data.

$$y = mx + c(1)$$

In a multiple linear regression, the input data can have multiple features and the multiple output variables can be predicted. In this case, a simple linear model can be represented as

$$Y = A_0X_0 + A_1X_1(2)$$

In the above equation, X_i is of the size (m, n) with m data points and n features. Y is of the size (m, o) with m data points and o outputs to be predicted. So A_i is of the size (n, o) .

The Cost function or the error between the predicted and actual value can be defined as

$$J(A) = \frac{1}{2m} \sum_1^m (Y_{predicted} - Y_{actual})^2(3)$$

The aim is to determine the coefficients A , such that, the error between the actual value and the predicted value is minimum. The derivative of the cost function with respect to the coefficient A minimizes the cost function. There are two ways to reduce the error or the cost function.

1) Using Analytical Solution:

The optimal coefficients can be determined by multiplying the pseudoinverse of the dataset with the output matrix.

$$A = (X^T X)^{-1} X^T Y \quad (4)$$

2) Using Gradient Decent:

The second term in the below equation is the first order derivative of the cost function with respect to the coefficient. All the coefficients (where j is the number of features) must be updated simultaneously. Here, α is the Learning rate.

$$A_j = A_j - \alpha \frac{1}{m} \sum_i^m (Y_{predicted} - Y_{actual}) X_{ij} \quad (5)$$

Another important aspect to be considered is the model complexities. This can be increased by fitting a multinominal model. The

$$Y = A_0 X_0 + A_1 X_1 \quad (6)$$

$$Y = A_0 X_0 + A_1 X_1 + A_2 X_2 \quad (7)$$

$$Y = A_0 X_0 + A_1 X_1 + A_2 X_2 + A_3 X_3 + \dots \quad (8)$$

The model complexity increases from as the degree of the equation increases. When the model is too simple there can be a risk of under fitting and when the model is too complex, data can be overfitted.

The best model can be obtained by achieving a variance-bias trade off. Variance is defined as the error in the test data and bias is the error in the training data. The model is said to be underfitting when the bias is high and overfitting when the variance is high. An optimal solution is the one when both bias and variance are low.

Regularization is the method used to avoid overfitting by adding an additional penalty term in the error function. Regularized cost function can be defined as follows. λ is the regularization parameter.

$$J(A) = \frac{1}{2m} \sum_1^m (Y_{predicted} - Y_{actual})^2 + \lambda \sum_1^n A_j^2 \quad (9)$$

2.b Implementation

The dataset considered has 5 features and 34 data points. The output data has 3 values to be predicted and has 34 data points. A 75%-25% train-test split has been done on the input and output dataset. First 25 records are used for training the model and the rest for the testing the model.

The model parameters considered in this project for linear regressions are:

- A. **Degree of polynomial (n):** The equation (3), (4) and (5) have $n=1,2$ and 3 respectively.
- B. **Learning Rate (α):** This parameter controls the rate at which the minima is reached. When the value is too high, there are chances of missing the minima and

when this rate value is too low, the number of iterations needed to reach the minima become high and hence time consuming.

- C. **Iterations:** This is the number of times the coefficient matrix A_j is updated during gradient decent to minimize the error.
- D. **Regularization parameter (λ):** This is the penalty parameter that controls the fitting of the model.

The following functions have been defined to implement the Linear regression algorithm.

i) **cost_function (X, y, a):**

The below code calculates the squared differences between the actual values and the predicted values.

```
def cost_function(X,y,a):  
    m=len(y)  
    h=X.dot(a)  
    cost=(np.sum(np.square(h-y)))/(2*m)  
    return cost
```

ii) **fit_as (X_train, Y_train, n):**

This uses analytical method to compute the coefficients i.e., it computes the pseudo inverse of the input dataset and multiplies it with the output

```
def fit_as(X_train,Y_train,n):  
    ones=np.ones((len(X_train),5))  
    X_train_con=np.concatenate((ones,np.power(X_train,1)),1)  
    if n>1:  
        for i in range(2,n+1):  
            X_train_con=np.concatenate((X_train_con,np.power(X_train,n)),1)  
  
    X_inverse=np.linalg.pinv(X_train_con)  
    A=np.matmul(X_inverse,Y_train)  
    return A
```

iii) **gradient_descent (X, y, learning_rate, Iterations, a):**

The below code minimizes the cost function. The coefficients are updated “iteration” number of times. The learning_rate controls the rate at which the global minima is reached.

```
def gradient_descent(X,y, learning_rate, iterations, a):
    a_hist=[]
    cost_hist=[]
    for i in range(iterations):
        h=X.dot(a)
        sub=h-y
        gradient=sub.T.dot(X)
        a=a-(learning_rate*gradient.T)
        cost=cost_function(X,y,a)
        a_hist.append(a)
        cost_hist.append(cost)
    minimum_indx=cost_hist.index(min(cost_hist))
    minimum_a=a_hist[minimum_indx]
    return minimum_a
```

iv) **fit (X_train, Y_train, n, learning_rate, iterations):**

This function forms the equation which fits the training data, initializes the coefficients and calls the gradient decent function to minimize the cost function.

```
def fit(X_train,Y_train,n, learning_rate, iterations):
    ones=np.ones((len(X_train),5))
    X_train_con=np.concatenate((ones,np.power(X_train,1)),1)
    if n>1:
        for i in range(2,n+1):
            X_train_con=np.concatenate((X_train_con,np.power(X_train,n)),1)
    a=np.zeros(((n+1)*5,3))
    A= gradient_descent(X_train_con,Y_train, learning_rate, iterations, a)
    return A
```

v) **fit_with_reg (X_train, Y_train, n, lamb):**

This function uses a regularized cost function to compute the gradient decent using the calculus method.

```
def fit_with_reg(X_train,Y_train,n, lamb):
    ones=np.ones((len(X_train),5))
    X_train_con=np.concatenate((ones,np.power(X_train,1)),1)
    if n>1:
        for i in range(2,n+1):
            X_train_con=np.concatenate((X_train_con,np.power(X_train,n)),1)
    A=np.linalg.inv(X_train_con.transpose().dot(X_train_con) + lamb * \
        np.identity(5*n+5))\
        .dot(X_train_con.transpose()).dot(Y_train)
    return A
```

vi) **predict (X, A, n):**

Predict function predicts the output values for new data using the coefficients obtained by the fitting the dataset to an equation using the fit function.

```
def predict(X,A,n):
    ones=np.ones((len(X),5))
    X_con=np.concatenate((ones,np.power(X,1)),1)
    if n>1:
        for i in range(2,n+1):
            X_con=np.concatenate((X_con,np.power(X,n)),1)
    Y_predicted=np.matmul(X_con,A)
    return Y_predicted
```

vii) **mean_error (Y_predicted, Y):**

The function below calculates the mean squared error between the predicted and actual output values. It also computes the mean absolute percentage error.

```
def mean_error(Y_predicted,Y):
    mse=(np.square(Y - Y_predicted)).mean(axis=0)
    mape=(np.abs(Y-Y_predicted)/Y).mean(axis=0)
    print(" Error, --Y1-- , --Y2-- , --Y3-- ")
    print(" mape    %.2f    %.2f    %.2f "%(mape[0],mape[1],mape[2]))
    print(" mse     %.2f    %.2f    %.2f "%(mse[0],mse[1],mse[2]))
    return mse[0],mse[1],mse[2]
```

2.c Results

Input:

Input Learning rate: 0.01
 Number of iterations: 1000
 Till which degree polynomial do you want: 8

Partial Output: The output of only two equations has been shown.

```
EUQATION DEGREE( 1 ) --> A 0.X 0 + A 1 .X 1
***** CO-EFFICIENTS *****
USING GRADIENT DECENT
[[ 0.6761379  0.52756567  5.05583434]
 [ 0.6761379  0.52756567  5.05583434]
 [ 0.6761379  0.52756567  5.05583434]
 [ 0.6761379  0.52756567  5.05583434]
 [ 0.6761379  0.52756567  5.05583434]
 [ 0.86130376  0.2927377  -1.64694605]
 [ -0.70996073 -0.44065541 -7.98838978]
 [ 0.62619229 -0.39654088 -11.57507391]
 [ 0.52618398  0.1488556  -0.64171265]
 [ 0.20225477  1.22810394 -4.44044028]]
USING ANALYTICAL SOLUTION
[[ 0.67621567  0.52767998  5.05665484]
 [ 0.67621567  0.52767998  5.05665484]
 [ 0.67621567  0.52767998  5.05665484]
 [ 0.67621567  0.52767998  5.05665484]
 [ 0.67621567  0.52767998  5.05665484]
 [ 0.86593606  0.30249458 -1.569886 ]
 [ -0.71253506 -0.44307835 -8.00260502]
 [ 0.6245034  -0.41006035 -11.69851063]
 [ 0.52759183  0.15280787 -0.60899933]
 [ 0.20352553  1.23310461 -4.39718712]]
```

It can be observed that both the methods have given almost similar coefficient values. Gradient decent was executed with learning rate = 0.01 and iterations=100. Lower values of learning rate took many iterations to compute the coefficients that gives minimum error. Higher values of learning rate and iterations were tried and it was observed that the minima was being missed due to high learning rate.

```
@@@ Mean error @@@
***** BIAS *****
Error, --Y1-- , --Y2-- , --Y3--
mape    0.12    0.16    0.25
mse     0.23    0.29    76.46
***** VARIANCE *****
Error, --Y1-- , --Y2-- , --Y3--
mape    0.16    0.13    0.25
mse     0.53    0.21    69.83

EUQATION DEGREE( 2 ) --> A 0.X 0 + A 1 .X 1 + A 2 .X 2
```

```

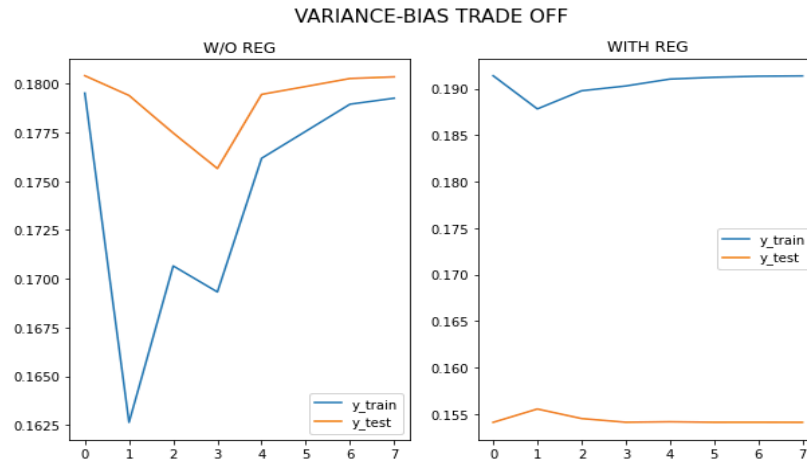
***** CO-EFFICIENTS *****
USING GRADIENT DESCENT
[[ 0.62139294 0.48683134 4.73017694]
[ 0.62139294 0.48683134 4.73017694]
[ 0.62139294 0.48683134 4.73017694]
[ 0.62139294 0.48683134 4.73017694]
[ 0.62139294 0.48683134 4.73017694]
[ 1.08588611 0.38238725 3.5312879 ]
[ -0.89870043 -0.24976554 -10.23751454]
[ 0.61048948 -0.5473254 -8.62279784]
[ 0.71205001 0.48306433 -0.04143743]
[ 0.32224892 1.39674742 -2.24133563]
[ 0.7818327 0.53260483 -6.50042126]
[ 0.84671741 -0.82658805 -0.79449215]
[ -0.49668196 0.61935389 12.55039936]
[ 1.49067068 2.11547739 -8.77519201]
[ 1.82434103 0.86082989 40.39472439]]
USING ANALYTICAL SOLUTION
[[ 0.61586331 0.45185252 4.05810637]
[ 0.61586331 0.45185252 4.05810637]
[ 0.61586331 0.45185252 4.05810637]
[ 0.61586331 0.45185252 4.05810637]
[ 0.61586331 0.45185252 4.05810637]
[ 1.51756137 0.67471864 7.27141041]
[ -1.15232451 0.05209438 -11.98177411]
[ 0.66402362 -0.62290592 -6.11952523]
[ 0.78535856 0.73389171 0.49336216]
[ 0.38031291 1.44352366 -1.74140944]
[ 1.40841856 2.08630779 2.36091747]
[ 1.57915161 -2.41325471 -3.39325752]
[ -4.89540208 6.10396885 113.05440321]
[ 1.70195167 2.56027885 -15.49872106]
[ 3.09172009 1.3809232 63.67901225]]

@@@ Mean error @@@
***** BIAS *****
Error, --Y1-- , --Y2-- , --Y3--
mape 0.11 0.15 0.23
mse 0.16 0.22 59.88
***** VARIANCE *****
Error, --Y1-- , --Y2-- , --Y3--
mape 0.17 0.16 0.21
mse 0.66 0.30 49.47

```

The below graph shows that both the test and train error are comparatively lower when $y=3$. When $y=3$ the degree of the polynomial is 4. The equation $Y = A_0X_0 + A_1X_1 + A_2X_2 + A_3X_3 + A_4X_4$ has fit the dataset better than the other equations.

It can also be observed that, without regularization, the train error was lesser than the test error. This implies that the model is overfitting. With regularization ($\lambda = 1$), the test error has reduced from approximately 0.18 to 0.15 and the train error has increased evidently.



Graph 1: Variance and bias trade off with and without regularization

The below screenshot shows the prediction for new values of input data using a regularized cost function with lambda value = 1 and degree = 2.

```
x1=float(input("Enter feature 1: "))
x2=float(input("Enter feature 2: "))
x3=float(input("Enter feature 3: "))
x4=float(input("Enter feature 4: "))
x5=float(input("Enter feature 5: "))
```

```
Enter feature 1: 0.1
Enter feature 2: -0.56
Enter feature 3: 0.78
Enter feature 4: 1.23
Enter feature 5: 0.56
```

```
# NEW DATA PREDICTION
new_x= np.array([[x1] , [x2] , [x3] , [x4] , [x5] ] )
coefs=fit_with_reg(X_train,Y_train,2,lamb=1)
y_new_predicted=predict(new_x.T,coefs,2)
print("New X values are: ", new_x.T)
print("Predicted y values are: ",y_new_predicted)
```

```
New X values are: [[ 0.1 -0.56  0.78  1.23  0.56]]
Predicted y values are: [[ 4.59345864  3.60582085 23.16325411]]
```

3. ARTIFICIAL NEURAL NETWORK

3.a Theory

The main idea behind Artificial Neural networks is to make the computers to be able to make decisions like the human brain. An Artificial Neural Network works like interconnected brain cells.

Consider,

n: Number of features in the dataset

m: Number of records in the dataset

X : Input dataset of size (m, n)

h : Number of hidden layers

nh_i : Number of nodes in each hidden layer

W_{ij} : Weight from i^{th} node of the previous layer to j^{th} node of the next layer.

Every ANN is made up of three parts.

- 1) **Input layer:** This layer consists of n nodes. Data is first fed at the input layer.
- 2) **Hidden layers:** Data from the input layer travels to the h hidden layers. The connections from each input node to hidden layer node is weighted. The data is multiplied with the weights and a bias is added. There can be more than one hidden layer and each layer can consist of thousands of nodes.
- 3) **Output layers:** After the data passes through multiple hidden layers that performs calculations on the input data, the output layer has the final decisions.

Every node consists of two parts:

- 1) The dataset multiplied with the weights are summarized from a vector to scalar function using summation

$$y = \sum X_i W_{ij} \quad (10)$$

- 2) A transfer function/ activation function produces input values to next layer. These functions are listed in the below figure.

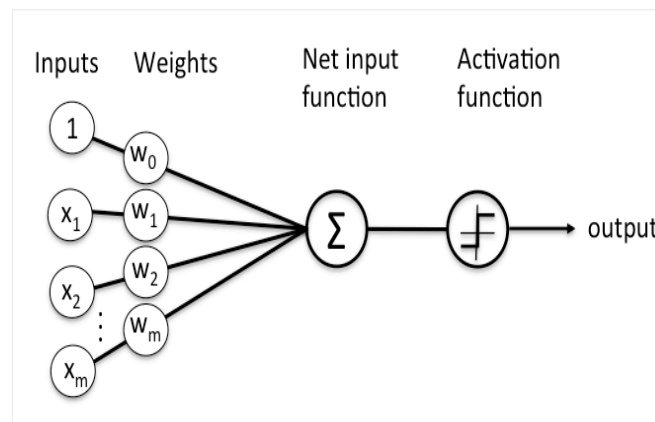


Figure 1: Two steps in in each node

The process of finding out the optimal weights that has to be multiplied with the input values is called the learning process of neural networks. This takes several iterations of computation of the weights, computation of the error between the actual and predicted values and updating the weights to reduce the error. Every iteration is called as an **epoch**.

Each epoch consists of the steps:

- 1) **Forward propagation:** In the forward propagation, the weights are multiplied with the input values at each layer to predict the output at the final layer. A bias term is also added. Every hidden layer can have different activation functions defined to compute the input values for the next layer.

- 2) **Error computation:** The final output values are compared with the predicted values at the output layer. The error is calculated.
- 3) **Backward propagation:** The goal of this step is to reduce the error calculated in the previous step by using gradient decent. Chain rule differentiation of the cost function with respect to weights and bias is the heart of backward propagation step.

The below figure shows the different activation functions that can be used and its derivatives.










Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 2: Activation functions and its derivatives

3.b Implementation

In this project, 1 input layer, 2 hidden layers and 1 output layer has been considered. tanh function has been used in the hidden layer 1 as the activation function. Sigmoid function has been used in the hidden layer 2 and tanh again in the last output layer.

The below code shows the first step in the iterative algorithm. Z1, Z2, Z3 are computed as shown in equation (10). Activation functions are applied on each of the outputs.

```
# FORWARD PROPOGATION
z1=a0.dot(W1)+b1
a1=np.tanh(z1) # Activation function at hidden layer 1 is tanh

z2=a1.dot(W2)+b2
a2=1/(1 + np.exp(-z2)) # Activation function at hidden layer 2 is sigmoid

z3=a2.dot(W3)+b3 # Activation function at output layer is tanh
a3=np.tanh(z3)
```

The next step is to compute the cost function. This is the error between the predicted and the actual values. The m is the number of records in the datasets. The equation to compute the cost function is:

$$J = \frac{1}{2m} \sum (y - y_{\text{predicted}})^2 \quad (11)$$

The implementation of computing the error can be seen in the below code.

```
# COST FUNCTION
h = copy.deepcopy(a3)
j = ((1/(2*m))*((y_data.iloc[:,0]-h.iloc[:,0])**2).sum())
```

The most crucial step is the back propagation. Weights and bias at each layer can be computed using the below general equations

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial W_{ij}} \quad (12)$$

$$\frac{\partial J}{\partial b_{ij}} = \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial b_{ij}} \quad (13)$$

```
# BACKWARD PROPOGATION
da3=(-1/m) * (y_data.to_numpy()-a3)
dz3= da3*(1-(np.square(a3))) #tanh derivative wrt a3
dW3= a2.T.dot(dz3)
db3= np.sum(dz3,axis=0)

da2= dz3.dot(W3.T)
dz2= da2*(a2*(1-a2)) # sigmoid derivative wrt a2
dW2= a1.T.dot(dz2)
db2= np.sum(dz2,axis=0)

da1= dz2.dot(W2.T)
dz1= da1*(1-(np.square(a1))) # tanh derivative wrt a1
dW1= a0.T.dot(dz1)
db1= np.sum(dz1,axis=0)
```

Finally, the below code updates the weights and bias for the next iteration. “rp” in the below code is the regularization parameter which has the same effect as explained in the linear regression section.

```
# UPDATING WEIGHTS and BIAS
W1=W1-((lr/m)*(dW1 + (rp*W1)))
W2=W2-((lr/m)*(dW2 + (rp*W2)))
W3=W3-((lr/m)*(dW3 + (rp*W3)))

b1=b1-((lr/m)*(db1.to_numpy()))
b2=b2-((lr/m)*(db2.to_numpy()))
b3=b3-((lr/m)*(db3.to_numpy()))
```

3.c Results

The results obtained by tuning the various parameters have been tabulated.

lr: learning rate

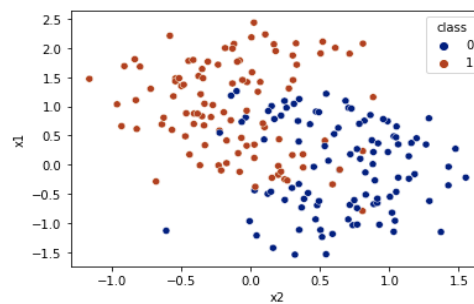
rp: regularization parameter

n_iter: number of iterations

n_h1_dim: number of layers in hidden layer- 1

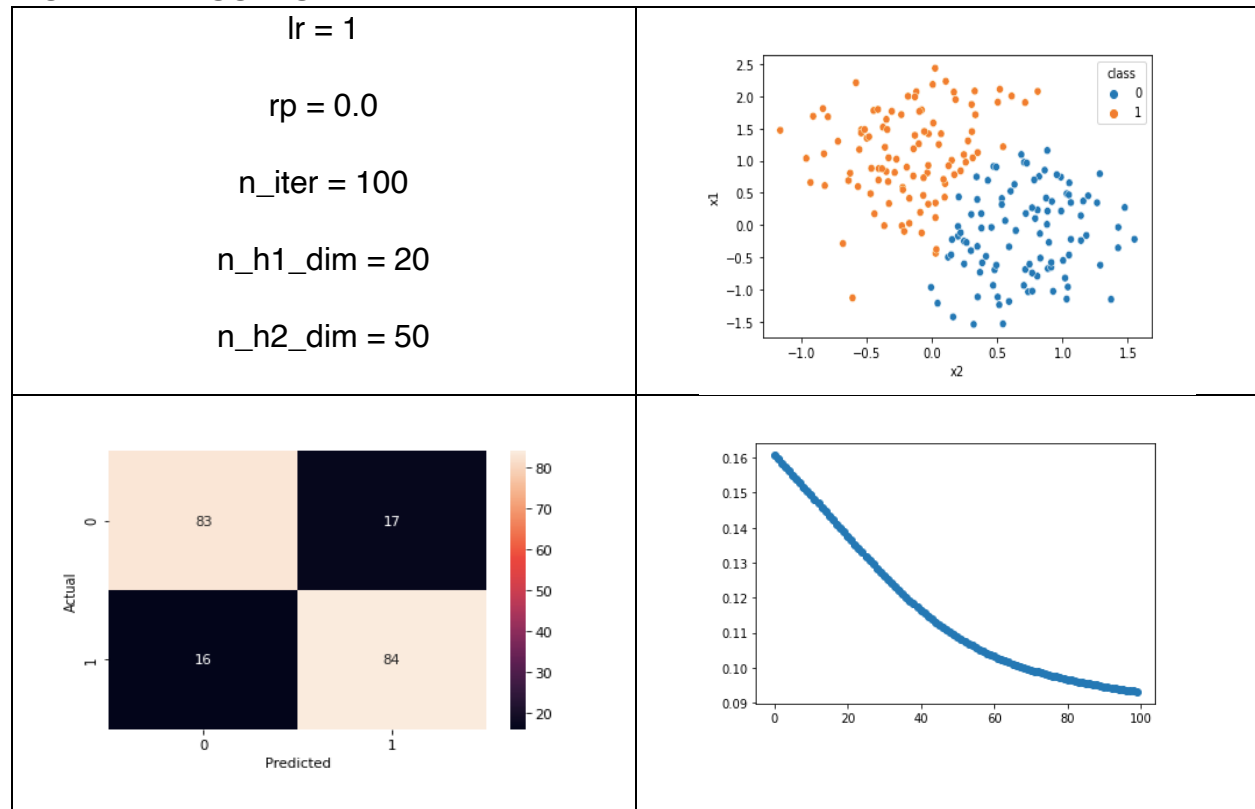
n_h2_dim: number of layers in hidden layer- 2

The graph 2 shows the scatter plot of the output classes on the dataset. These are the true classes. The model predictions and confusion matrix for three of the many models that were created using different values for the parameters have been shown in Table 1.



Graph 2: Classes of the dataset

MODEL 1 RESULTS



MODEL 2 RESULTS

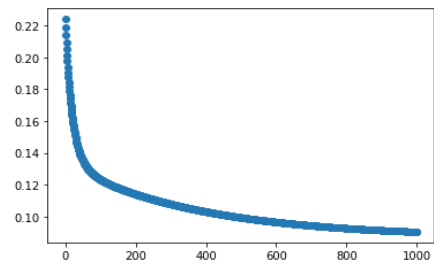
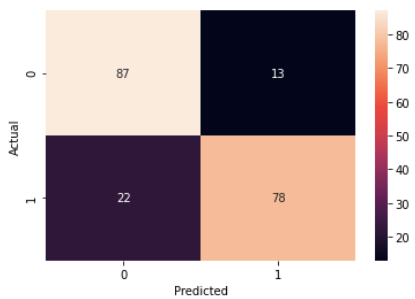
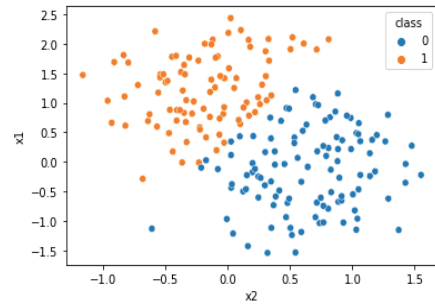
lr = 0.5

rp = 0.1

n_iter = 1000

n_h1_dim = 30

n_h2_dim=40



MODEL 3 RESULTS

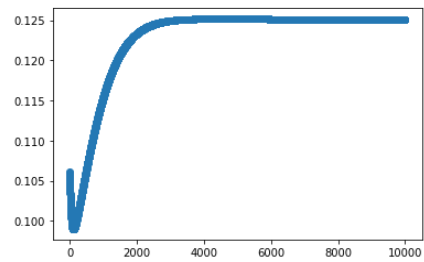
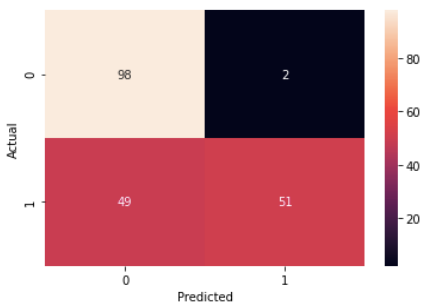
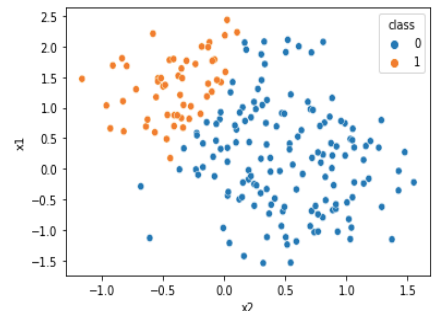
lr = 0.5

rp = 0.3

n_iter = 10000

n_h1_dim=10

n_h2_dim=20



It can be observed that the first model has performed well compared to the other two models. About 85% of the classes have been predicted correctly. This model is also not overfitting. This can be inferred by comparing the graph (2) and the predictions column graph of model 1. The decision boundary is similar to graph (2), but not the exact. The second model does not perform as well as the first model. It only classifies 70% of the data properly. The third model has least accuracy out of the three.

```
# prediction with new x values
x_to_predict_y = np.array([-2.34,0.5])
y_predicted=predict(x_to_predict_y,w1,w2,w3,b1,b2,b3)
y_class= 0 if y_predicted < 0.5 else 1
print("New prediction:\n")
print("For x1= ",x_to_predict_y[0],"\nFor x2= ",\
      x_to_predict_y[1],"\ny predicted= ",\
      y_predicted,"\nclass of y= ",y_class)
```

New prediction:

```
For x1= -2.34
For x2= 0.5
y predicted= [[0.36756385]]
class of y= 0
```

Model 1 has been chosen to predict values for new input. When $x_1 = -2.34$ and $x_2 = 0.5$, the model has built predicted a y value of 0.36 which belongs to class 0.

4. CONCLUSION

Model selection and parameter tuning are few of the most important steps in Machine Learning. While developing a Linear regression model and Artificial Neural Network, which works best for the given dataset, many parameters were tweaked to see the changes in the model performance. The influence of these parameters on the model performance was analysed. The parameters that gave minimum error have been used to predict data for a new example input.