Machine Learning Algorithm Library

M Naveen Kumar
IIT (ISM) Dhanbad

Description:

This library will consists of 5 Machine Learning Algorithm that are coded from Scratch that will obtain the most efficient and precise output for a given data set. These algorithm can be used to obtain results for common day to day queries like calculating the House selling rates,etc. These algorithms will be coded in such a way that it gives the accurate result in the most time effective manner. This library will be very user friendly and will be coded in Jupyter notebook. This project will also have a detailed Report on the status and working of the same.

This library will include the following Algorithms:

- **Linear Regression** - **Linear Regression** is a machine learning algorithm based on **supervised learning.** It shows a relationship between a dependent variable and an independent variable.
- **Polynomial Regression - Polynomial Regression** is a **Supervised learning** based machine learning algorithm. It shows a relationship between a dependent variable and an independent variable as a nth degree polynomial.
- **Logistic Regression - Logistic Regression** is also based on **Supervised Learning** based machine learning algorithm. It is used to calculate or predict the probability of a binary event occurring.
- **K-Nearest Neighbor(KNN) Algorithm - KNN Algorithm** uses the surroundings to make calculation and classifications of an individual data point.
- **n-layer Neural Networks : n-layer Neural Networks** is a n-layered computing systems inspired by the **biological neural** networks that constitute animal brains.

These algorithms will be coded from scratch along with neat report explaining the process of development and method of implementation. These algorithms will be frequently tested with the given data set for accuracy and error lessness, These codes will be ready for the hidden data set that is to be provided later.

Weekly Time Table which was to be followed:

I stuck to the timeline as much as possible and finished coding Linear,Polynomial and Logistic regression within the mid evaluation. Later I utilised the extension of time on my favour and was able to finish KNN and Single Layer Neural Network as per the extended date.

Linear Regression:

I coded the multivariate linear regression in the following manner:

I coded a basic Gradient descent function and a derivative function which updates the weights and biases to produce the best optimised values based on the given alpha(learning rate) and number of iterations.

For a learning rate of 0.01 and for 1000 iterations, I have received the following cost(error) and predicted the rmse value after the final iteration.

In [14]:
```
w_final = np.zeros(n,)
b_final = 0
losses_array = np.zeros(1001)
x_modified = z_score(x_values)
w_final,b_final,losses_array= multi_linear_regression(x_modified,y_values,m,n,w_init,b_init,0.01,1001)
```

```
The loss after 0 iterations is 29785.029539374613 and the rmse is 244.0697832152707
The loss after 50 iterations is 13216.9524770037 and the rmse is 162.5850698988299
The loss after 100 iterations is 7772.891408789139 and the rmse is 124.6827286258136
The loss after 150 iterations is 5859.370102209976 and the rmse is 108.25313022919916
The loss after 200 iterations is 5168.1892171557265 and the rmse is 101.6679813624302
The loss after 250 iterations is 4915.899646084229 and the rmse is 99.15542996814878
The loss after 300 iterations is 4823.436795898406 and the rmse is 98.21849923408935
The loss after 350 iterations is 4789.492475033252 and the rmse is 97.87228897939653
The loss after 400 iterations is 4777.020812432843 and the rmse is 97.74477799282009
The loss after 450 iterations is 4772.436194845141 and the rmse is 97.69786276930668
The loss after 500 iterations is 4770.750208605941 and the rmse is 97.6806040993394
The loss after 550 iterations is 4770.129969831078 and the rmse is 97.67425423141022
The loss after 600 iterations is 4769.901719800147 and the rmse is 97.67191735396769
The loss after 650 iterations is 4769.817695112676 and the rmse is 97.67105707539646
The loss after 700 iterations is 4769.786753311362 and the rmse is 97.67074027887126
The loss after 750 iterations is 4769.775355384144 and the rmse is 97.67062358134245
The loss after 800 iterations is 4769.771155407506 and the rmse is 97.6705805799014
The loss after 850 iterations is 4769.769607276516 and the rmse is 97.67056472936476
The loss after 900 iterations is 4769.769036445623 and the rmse is 97.67055888491294
The loss after 950 iterations is 4769.768825900481 and the rmse is 97.67055672924651
The loss after 1000 iterations is 4769.768748218553 and the rmse is 97.67055593390009
The values of w and b are [72.99307834 66.24632655 97.43785992  1.7174908  17.36250286 24.81519873
 71.23057774 30.99419528 20.34967376 93.74484167 39.36754835 37.253725
  2.82789652 40.7011222  40.7011222  46.08533099 21.51532946 33.84432661
 33.68224674 39.24677312] -0.36563510718879455
```

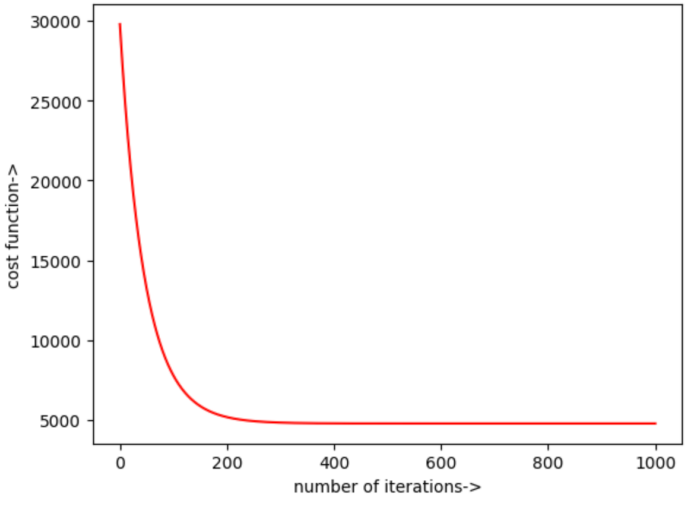
I received a **R2 score of 0.8428783212232978** upon running the code.

In [16]:
```
r2(x_modified,y_values,w_final,b_final,m,n)
```

Out[16]: 0.8428783212232978

I plotted the cost vs no. of iterations graph and received a **rectangular hyperbola** as expected

In [19]:
```
#plot the graph of iterations vs cost function
plt.plot(np.arange(1001),losses_array,c='r')
plt.xlabel('number of iterations->')
plt.ylabel('cost function->')
plt.show()
```



Polynomial Regression:

I coded polynomial regression in the following manner:

I have coded a part which can add all the permutation and combinations of the features to the existing features based on the given degree 'n' :

In [3]:
```
#enter the degree of polynomial to find the features
degree = int(input("Enter value of degree : "))
```

```
Enter value of degree : 5
```

In [4]:
```
#code to add additional features based on degree
tmp = np.zeros((m,1))
for i in range(degree+1):
    for j in range(degree+1):
        for k in range(degree+1):
            if (i+j+k)>degree or (i+j+k)<=1:
                continue
            else:
                tmp = ((x_values[:,0]**i)*(x_values[:,1]**j)*(x_values[:,2]**k))
                tmp = np.reshape(tmp,(m,1))
                x_values = np.hstack((x_values,tmp))

#assinging value of n
n = x_values.shape[1]

print(x_values.shape)
```

```
(50000, 55)
```

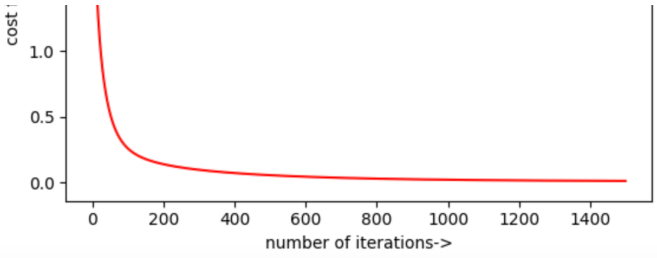
The Gradient descent and the Derivative function is similar to that of the Linear regression and I managed to obtain the following cost and rmse after 1500 iterations and a learning rate of 0.01 .

In [16]:
```
w_final = np.zeros(n,)
b_final =  0
loss_array = np.zeros(num_iterations,)
x_modified = z_score(x_values,m,n)
w_final,b_final,loss_array = polynomial_regression(x_modified,y_values,m,n,w_init,b_init,0.01,num_iterations)
```

```
The loss after 1250 iterations is 16729387319.623083 and the rmse is 182917.39840497996
The loss after 1300 iterations is 15934014796.875135 and the rmse is 178516.18860414388
The loss after 1350 iterations is 15229282189.190792 and the rmse is 174523.82180774515
The loss after 1400 iterations is 14604698421.965803 and the rmse is 170907.56812947636
The loss after 1450 iterations is 14051288644.170746 and the rmse is 167638.2333727646
The values of w and b are [-1.37100027e+05  1.02218315e+03  3.36621791e+02  2.78387876e+03
 -1.81636140e+03  6.97522375e+01  2.48753874e+03 -1.01652551e+03
  9.53261982e+02  1.94363163e+01  4.10858297e+02  4.39425551e+03
  1.76158795e+03 -8.78204544e+02 -1.75240611e+03 -1.11429937e+03
  4.14015036e+02  2.86834244e+03 -2.81094872e+03  3.03483608e+03
 -2.50936264e+03  1.21039357e+05 -4.69062114e+04 -1.01891597e+04
  5.50290796e+04 -8.87361016e+03 -6.84405306e+03  1.04040187e+04
 -4.06061873e+03 -2.05183281e+04  1.30684049e+02  2.43859905e+04
 -5.29110845e+03 -1.72669998e+02  1.43495132e+04  7.78931055e+03
 -2.64129620e+03  6.42636233e+03  8.37821522e+03 -2.58314235e+03
  1.99171925e+04  6.48093917e+03  3.26609887e+03  3.67597959e+03
  4.81490373e+03  5.52570488e+05 -1.18235132e+05 -4.92952415e+04
  4.24277466e+04  5.32807609e+04  7.33838749e+03 -1.39764157e+04
  6.37702519e+04  9.27234986e+04  2.11259041e+06] 71413.11207132586
```

I plotted the graph of cost vs no.of iterations and received the expected graph of **Rectangular hyperbola** as expected similar to Linear Regression.

In [24]:
```
#plot the graph of iterations vs cost function
plt.plot(np.arange(1500),loss_array,c='r')
plt.xlabel('number of iterations->')
plt.ylabel('cost function->')
plt.show()
```



I also received an R2 score of **0.9959344357973254** after the final iteration.

```
In [17]: r2(x_modified,y_values,w_final,b_final,m,n)
Out[17]: 0.9959344357973254
```

I also received an R2 score of **0.8944251767004523** when the degree of polynomial was set to 3 and a learning rate of 0.01 and number of iterations 1500.

```
In [17]: r2(x_modified,y_values,w_final,b_final,m,n)
Out[17]: 0.8944251767004523
```

Logistic Regression:

I coded Logistic Regression in the following manner:

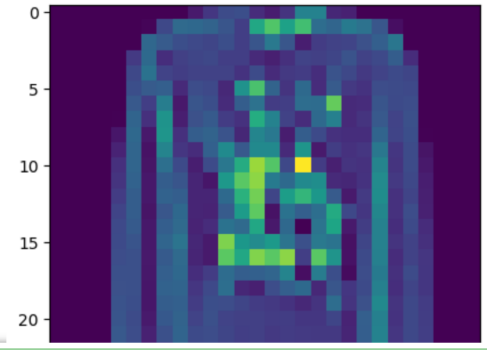
I code the Logistic Regression using the **one vs all coding (One Hot Encoding)** method and used the **sigmoid function** for the classification.

```
In [4]: num_unique = np.unique(y_values)
        print((num_unique))
        y_classes = np.zeros((m,len(num_unique)))
        for j in range(len(num_unique)):
            for i in range(m):
                if y_values[i] == num_unique[j]:
                    y_classes[i,j] = 1
        y_classes = y_classes.T
        print(y_classes.shape)

        [0 1 2 3 4 5 6 7 8 9]
        (10, 30000)
```

I first plotted a sample image using the **plt.imshow** function to get a deeper understanding of the given **classification_train.csv** file.

```
In [3]: plt.imshow(x_values[2998].reshape(28,28))
Out[3]: <matplotlib.image.AxesImage at 0x1221abe50>
```



For the gradient descent I had trained the weights and bias for every individual label and applied the z_score feature scaling for the given x train variables. In order to implement the

z_score I had coded the mean and standard deviation from scratch.

```
In [8]: #to find mean
        def mean(x_values,m,n):
            mean = np.zeros(n,)
            for j in range(n):
                summation = 0
                for i in range(m):
                    summation += x_values[i,j]
                mean[j] = summation/m
            return mean
```

```
In [9]: #to find standard deviation
        def std_deviation(x_values,mean,m,n):
            summation = np.zeros(n,)
            standard_deviation = np.zeros(n,)
            for j in range(n):
                for i in range(m):
                    summation[j] += ((x_values[i,j]-mean[j])**2)
                summation[j] = (summation[j]/m)
                standard_deviation[j] = math.sqrt(summation[j])
            return standard_deviation
```

```
In [10]: #implementing the z score normalization
         def z_score(x_values,m,n):
             avg = mean(x_values,m,n)
             standard_deviation = std_deviation(x_values,avg,m,n)
             x_modified = np.zeros_like(x_values)
             for j in range(n):
                 for i in range(m):
                     x_modified[i,j]= ((x_values[i,j]-avg[j])/standard_deviation[j])
             return x_modified
```

I defined a function for obtaining the accuracy and received an accuracy of **83.42** after iterating every label for a learning rate of 0.1.

```
In [59]: y_classes_cap = np.zeros_like(y_classes)
         y_cap = np.zeros_like(y_values)
         for i in range(len(num_unique)):
             y_classes_cap[i] = main_func(x_modified,w_final[i],b_final[i])
         for i in range(30000):
             temp = np.argmax(y_classes_cap.T[i])
             y_cap[i] = num_unique[temp]
         print(w_final.shape)
         print(x_modified.T.shape)
         print(y_cap)
         #for checking accuracy
         equal = np.sum(y_cap==y_values)
         accuracy = (equal/m)*100
         print(accuracy)
```

```
(10, 784)
(784, 30000)
[8 4 3 ... 9 4 0]
83.42
```

K-Nearest Neighbours Classification:

KNN Classification was coded in the following manner:

I first figured out the euclidean distance to find the nearest neighbours of a given particular dataset.

```
In [4]: def nearest_values(data_point,x_values,m,n):
            dist_array = []
            for i in range(m):
                distance = euclidean_distance(data_point,x_values[i])
                dist_array.append(distance)
            #making data frame of top k nearest neighbours
            distance_array = pd.DataFrame(dist_array)
            k_nearest_neighbours = distance_array.sort_values(0)
            return k_nearest_neighbours
```

```
In [5]: def KNN_classification(k_nearest_neighbours,k):
            #k_nearest_neighbours = nearest_values(data_point,x_values,m,n,k)
            k_nearest_neighbours_array = y_values_train[k_nearest_neighbours[:k].index]
            #to find the highest occuring label among the k nearest neighbours
            unique,c = np.unique(k_nearest_neighbours_array,return_counts=True)
            i = np.argmax(c)
            return unique[i]
```

```
In [3]: def euclidean_distance(x_values_1,x_values_2):
            distance = np.sqrt(np.sum((x_values_1-x_values_2)**2))
            return distance
```

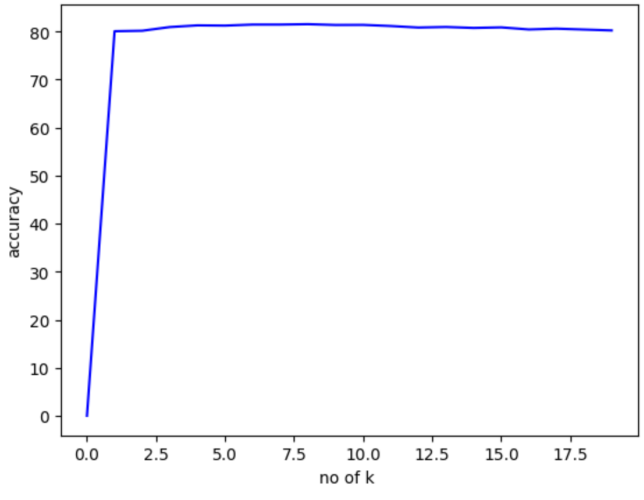
I then figured out the first K nearest neighbours to obtain the label of the data set. And upon setting the value of **K to 8**, I obtained the accuracy of the dataset to be **81.53333333333333**

```
In [8]: print(accuracy(y_predicted(8)))

        81.53333333333333
```

I also plotted the accuracy vs no. of k graph to figure out the best value of K and found the best value to be for K to be 8.

```
In [16]: plt.plot(range(20),accur[:20],c='blue')
         plt.xlabel("no of k ")
         plt.ylabel("accuracy")
         plt.show()
```




```
1 80.06666666666666
2 80.16666666666666
3 80.93333333333334
4 81.28333333333333
5 81.23333333333333
6 81.45
7 81.45
8 81.53333333333333
9 81.36666666666666
10 81.38333333333333
11 81.15
12 80.83333333333333
13 80.95
14 80.75
15 80.86666666666666
16 80.43333333333334
17 80.60000000000001
18 80.41666666666667
19 80.23333333333333
```

Single Layer Neural Classification:

I coded Single layer Neural Network Classification in the following manner:

I coded the single layer Neural Network in such a way that the user can decide the number of neurons present in the single hidden layer and the code will predict the output based on it.

```
In [13]: #for getting the number of nodes
         num_nodes = np.array([0])
         num_nodes[0] = int(input(" Enter the number of neurons for hidden layer "+str(1)+" : "))

          Enter the number of neurons for hidden layer 1 : 28
```

```
In [14]: w = [0]*2
         b = [0]*2
```

```
input_layer = n
output_layer = len(num_unique)
w[0] = np.random.randn(input_layer,num_nodes[0])*0.01
b[0] = np.zeros((num_nodes[0],))
input_layer = num_nodes[0]
w[1] = np.random.randn(input_layer,output_layer)*0.01
b[1] = np.zeros((output_layer,))
```

I used the **Sigmoid function** as the **activation function** in this single layer and the **softmax function** for obtaining the output.

In [6]:
```
def sigmoid(z):
    answer = 1/(1+np.exp(-z))
    return answer
```

In [7]:
```
def sigmoid_der(z):
    answer = sigmoid(z)*(1-sigmoid(z))
    return answer
```

I also used the **Cross-Entropy Loss** method instead of the **Square Sum Residue (SSR)** method in order to retrieve the loss as this method will have a larger curve as compared to the later which will Enhance the jump of the weights and bias.

In [9]:
```
def cross_entropy_loss(y,a):
    answer = (1/m)*np.sum(y*np.log(a))
    return -answer
```

I programmed the **Front propagation**,**Back propagation** and **updating the parameters** in the following manner

```
def neural_network(alpha,num_iters):
    for i in range(num_iters):
        #forward propogation
        #phase 1
        z_hidden[0] = np.dot(a,w[0])+b[0]
        a_hidden[0] = sigmoid(z_hidden[0])
        #phase 2
        z_out = np.dot(a_hidden[0],w[1])+b[1]
        a_out = softmax(z_out)
        #backward propogation
        #phase 1
        dJ_dzo = a_out - y_classes_train.T
        dzo_dwo = a_hidden[0]
        dJ_dwo = (1/m)*np.dot(dzo_dwo.T,dJ_dzo)
        dJ_dbo = (1/m)*dJ_dzo
        #phase 2
        dzo_dah1 = w[1]
        dJ_dah1 = np.dot(dJ_dzo,dzo_dah1.T)
        dah1_dzh1 = sigmoid_der(z_hidden[0])
        dzh1_dwh1 = x_values_train
        dJ_dw1 = (1/m)*np.dot(dzh1_dwh1.T,dah1_dzh1*dJ_dah1)
        dJ_db1 = (1/m)*dJ_dah1*dah1_dzh1
        #updating weights and biases
        w[0] = w[0] - alpha*dJ_dw1
        b[0] = b[0] - alpha*dJ_db1.sum(axis=0)

        w[1] = w[1] - alpha*dJ_dwo
        b[1] = b[1] - alpha*dJ_dbo.sum(axis=0)
```

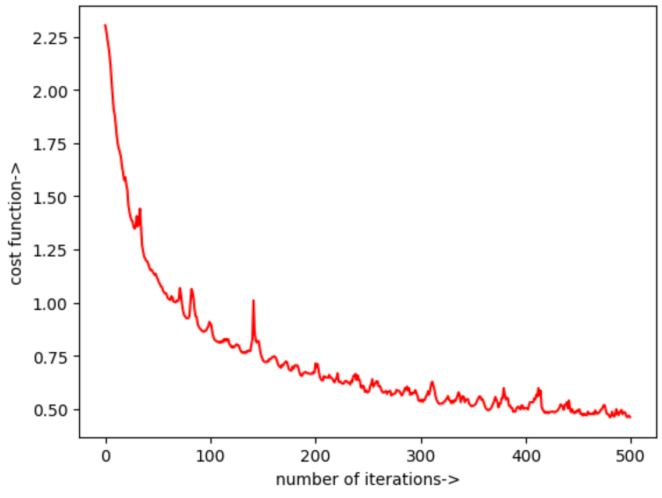
I obtained an **accuracy of 82.93333333333334** after running the code for 500 iterations with a learning rate of 0.3

In [19]:
```
print(accuracy(index))
```
```
82.93333333333334
```

I also plotted the graph of number of iterations vs cross entropy error and received the following graph

In [21]:
```
#plot the graph of iterations vs cost function
plt.plot(np.arange(500),loss_array,c='r')
plt.xlabel('number of iterations->')
```

```
plt.ylabel('cost function->')
plt.show()
```



N-Layer Neural Network Classification:

N layer Neural Network classification was coded in the following manner:

In the N layer Neural Network, the number of hidden layers and the number of neurons in each hidden layer has to be inputted by the user only.

In [4]:
```
num_layer = int(input("enter the number of hidden layers : "))
```
```
enter the number of hidden layers : 1
```

In [15]:
```
#for getting the number of nodes in every hidden layer
num_nodes = [0]*(num_layer)
for i in range(num_layer):
    num_nodes[i] = int(input(" Enter the number of neurons for hidden layer "+str(i+1)+" : "))
```
```
 Enter the number of neurons for hidden layer 1 : 28
```

I also wanted to provide an option for the user to choose the activation function from the list of the given activation functions and coded a bunch of them in an attempt to do the same.

In [5]:
```
def tanh(z):
    return np.tanh(z)
```

In [6]:
```
def derivative_tanh(z):
    return (1-(np.tanh(z))**2)
```

In [7]:
```
def relu(z):
    return (np.maximum(0,z))
```

In [8]:
```
def derivative_relu(z):
    return (z>0)
```

In [9]:
```
def sigmoid(z):
    answer = 1/(1+np.exp(-z))
    return answer
```

In [10]:
```
def derivative_sigmoid(z):
    answer = sigmoid(z)*(1-sigmoid(z))
    return answer
```

In [13]:
```
def softmax(z):
    num=np.exp(z)
    den=np.sum(num,axis=1)
    den=np.expand_dims(den,axis=1)
    soft=num/den
    return soft
```

The forward and backward propagation along with the updating parameters was coded within the Neural network method only.

```
#forward propogation================================
for j in range(num_layer):
    z_hidden[j] = np.dot(tmp,w[j])+b[j]
    a_hidden[j] = tanh(z_hidden[j])
    if j==num_layer-1:
        tmp=x_values
        break
    tmp = a_hidden[j]
z_output = np.dot(a_hidden[-1],w[-1])+b[-1]
a_output = softmax(z_output)
#back propogation===================================
#this is for the output layer only
dJ_dzo = a_output - y_classes.T
dzo_dwo = a_hidden[-1]
dJ_dwo = np.dot(dzo_dwo.T,dJ_dzo)
dJ_dbo = dJ_dzo
temp = dJ_dzo
dJ_dw = [0]*(num_layer)
dJ_db = [0]*(num_layer)
for k in range(num_layer,0,-1):
    #for dj_dw
    dJ_dw[k-1] =(1/m)*np.dot((derivative_tanh(z_hidden[k-1])*a_hidden[k-1]).T,temp)*w[k]
    #for temp
    temp = np.dot(temp,w[k].T)*(derivative_tanh(z_hidden[k-1]))
    dJ_db[k-1] =(1/m)*temp
for l in range(num_layer):
    w[l+1] = w[l+1] - alpha*dJ_dw[l]
    b[l] = b[l] - alpha*dJ_db[l].sum(axis=0)
```

I received an optimal **accuracy of 49.8** after iterating the code for 500 iterations.

```
In [21]: print(accuracy(index))
         49.8
```

Conclusion:

I had a lot of learning in the month-long Coding Competition, I believe I had given my best and have precisely programmed all the algorithms as expected by the mentors and fulfilled all the requirements as given in the WOC site.

I believe that I could improvise the library in the following ways upon given some extra time and space:

- Implementation of Classes and objectifying the code.
- Optimising the value of number of layers and neurons for getting the best accuracy for N layer.

I would also like to Thank my mentors **Mr. Vishak Bhat Sir, Mr. M Thirulok Sundar Sir and Mr. Gyanendra Permaar Sir** along with other Respected seniors for helping me with the project and guiding me throughout the competition.

About Me:

Name : M Naveen Kumar

Place : Bangalore, Karnataka.

Branch : Engineering Physics (2022-2026)

University : Indian Institute Of Technology (Indian School Of Mines) Dhanbad

Year Of Study : 1st year

Contact no : 9740978147

Email Id : 22je0529@iitism.ac.in