
WINTER
OF
CODE

MACHINE LEARNING BOOTCAMP

PROJECT REPORT

PROJECT DESCRIPTION

The project is to implement various machine learning algorithms in python from scratch. This project has various machine learning algorithms like linear regression, logistic regression, polynomial regression, KNN, and n layered neural network. The main motive of this project to implement the various machine learning algorithms from scratch and see how they are implemented by using basic machine learning library like NumPy, Pandas, Matplotlib in python language. To achieve this aim, we must first understand how these algorithms work and are implemented.

ALGORITHMS DESCRIPTION AND IMPLEMENTATION

LINEAR REGRESSION:

Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables. The goal of linear regression is to find the best-fitting linear equation that describes the relationship between the variables, with the objective of predicting the value of the dependent variable based on the values of the independent variables

IMPLEMENTATION OF LINEAR REGRESSION:

The Training dataset given to us has 50000 training examples and has several features and due to several features created a multiple linear regression model to fit Into linear dataset . To implement this idea I have initialized

Weights and biases with zero and then I predicted values and find the errors loss function by MSE mean squared error and then computed gradients of loss function with respect to w and b and all this we do by doing vectorization

```
class LinearRegression:
  def costPlot(cost, iters):
        #ITERS IS NO. OF ITERATIONS
        plt.plot(np.arange(iters),cost,'-b')
        plt.xlabel('Number of iterations')    #CREATING A FUNCTION TO PLOT COST VS ITERATIONS GRAPH
        plt.ylabel('Cost')
        plt.show()

  def Train_data(x,y,learning_rate,no_of_iterations):
    m=x.shape[0]
    n=x.shape[1]
    w=(np.zeros((n,1))) #CREATING A WEIGHT MATRIX OF ORDER N*1 WHERE N IS NO. OF FEATUERS
    b=0

    cost=[]     # for keeping cost data reserved USED TO PLOT COST VS ITERATIONS GRAPH
    for i in range(no_of_iterations):
      y_pred=np.dot(x,w)+b      # MODEL TO PREDICT DATA
      cost_i = (1/(2*m))*np.sum(np.square(y_pred - y))  #DEFINING MEAN SQUARED COST FUNCTION
      a=np.array(y_pred - y)
      a1 = a.transpose()    #TAKING TRANSPOSE SO THAT NEXT OPERATION CAN BE WHICH IS DOT OF M*N AND A I*M MATRIX
      derivative_w=(1/m)*np.dot(a1,x) #DERIVATIVE OF COST FUNCTION WITH RESPECT TO WEIGHTS
      derivative_w_T=derivative_w.transpose()  #TAKING TRANSPOSE TO MAKE IT IS OF SAME SHAPE AS W
      derivative_b=(1/m)*np.sum(y_pred-y)
      w=w-learning_rate*derivative_w_T  #UPDATING W AND B TO APPLY GRADIENT DESCENT
      b=b-learning_rate*derivative_b
      cost.append(cost_i) #APPENDING COST TO COST ARRAY
      if(i%math.ceil(no_of_iterations/100) == 0):
```

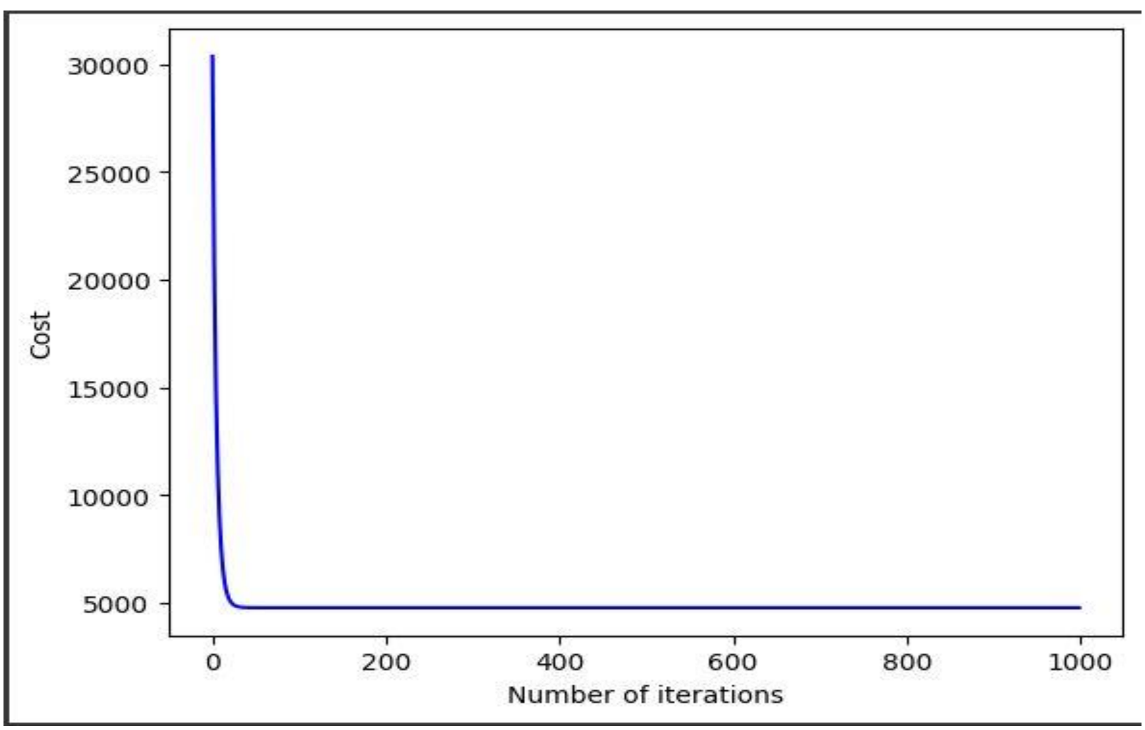
```
    w=(np.zeros((n,1))) #CREATING A WEIGHT MATRIX OF ORDER N*1 WHERE N IS NO. OF FEATUERS
    b=0

    cost=[]     # for keeping cost data reserved USED TO PLOT COST VS ITERATIONS GRAPH
    for i in range(no_of_iterations):
      y_pred=np.dot(x,w)+b      # MODEL TO PREDICT DATA
      cost_i = (1/(2*m))*np.sum(np.square(y_pred - y))  #DEFINING MEAN SQUARED COST FUNCTION
      a=np.array(y_pred - y)
      a1 = a.transpose()    #TAKING TRANSPOSE SO THAT NEXT OPERATION CAN BE WHICH IS DOT OF M*N AND A I*M MATRIX
      derivative_w=(1/m)*np.dot(a1,x) #DERIVATIVE OF COST FUNCTION WITH RESPECT TO WEIGHTS
      derivative_w_T=derivative_w.transpose()  #TAKING TRANSPOSE TO MAKE IT IS OF SAME SHAPE AS W
      derivative_b=(1/m)*np.sum(y_pred-y)
      w=w-learning_rate*derivative_w_T  #UPDATING W AND B TO APPLY GRADIENT DESCENT
      b=b-learning_rate*derivative_b
      cost.append(cost_i) #APPENDING COST TO COST ARRAY
      if(i%math.ceil(no_of_iterations/100) == 0):
            print('Cost is:',cost_i,'after ',i,'iterations') #PRINT COST VALUE VS ITERATIONS
    return w,b,cost
  def predict(X,w,b):
        return X.dot(w)+b #DEFINING A FUNCTION TO PREDICT VALUES
  def r2_score(yp,y):
        ymean=np.mean(y)
        ssr=np.sum(np.square(yp-y))
        ssm=np.sum(np.square(y-ymean))# DEFINING A FUNCTION TO CALCULATE R2 score
        r2=1-(ssr/ssm)
        return r2
```

During training several values of learning rate is taken and finally

- Learning rate as 0.1 which is the optimum for this model

```
learning_rate=0.1 #CALLING FUNCTION TO TARIN ON THIS DATA
iterations=1000
w,b,cost1=LinearRegression.Train_data(X_Train,Y_Train,learning_rate,iterations)
```

```
Cost is: 30357.23198081161 after  0 iterations
Cost is: 4769.768720624715 after  100 iterations
Cost is: 4769.76870276422 after  200 iterations
Cost is: 4769.768702764219 after  300 iterations
Cost is: 4769.768702764219 after  400 iterations
Cost is: 4769.768702764219 after  500 iterations
Cost is: 4769.768702764219 after  600 iterations
Cost is: 4769.768702764219 after  700 iterations
Cost is: 4769.768702764219 after  800 iterations
Cost is: 4769.768702764219 after  900 iterations
```



R2_SCORE: 0.84567

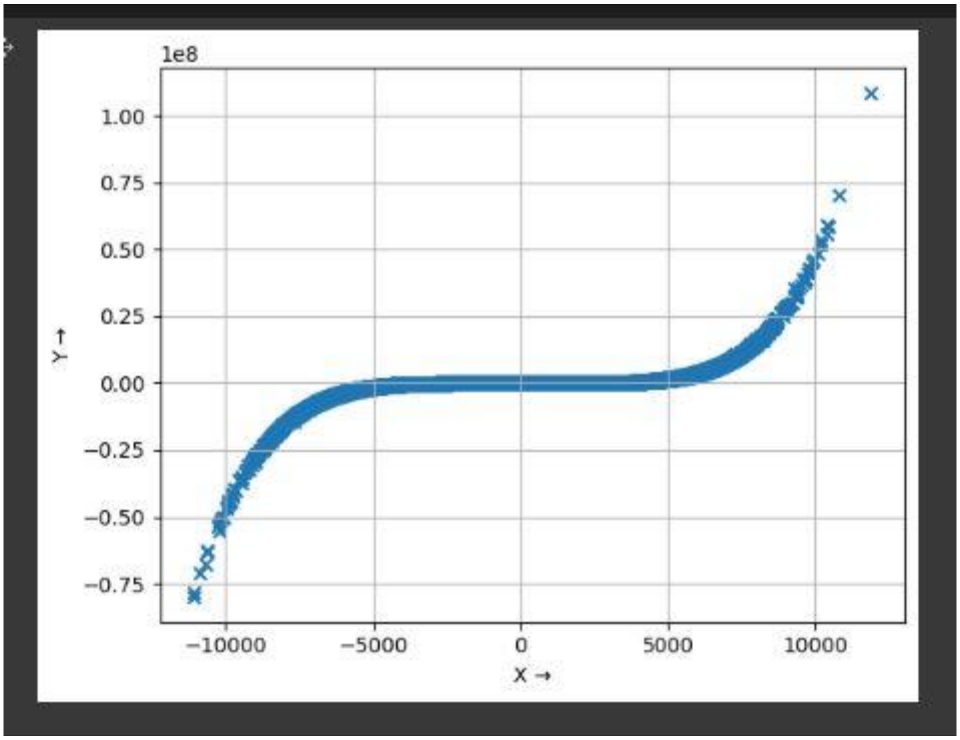
RMSE score: 97.66

POLYNOMIAL REGRESSION:

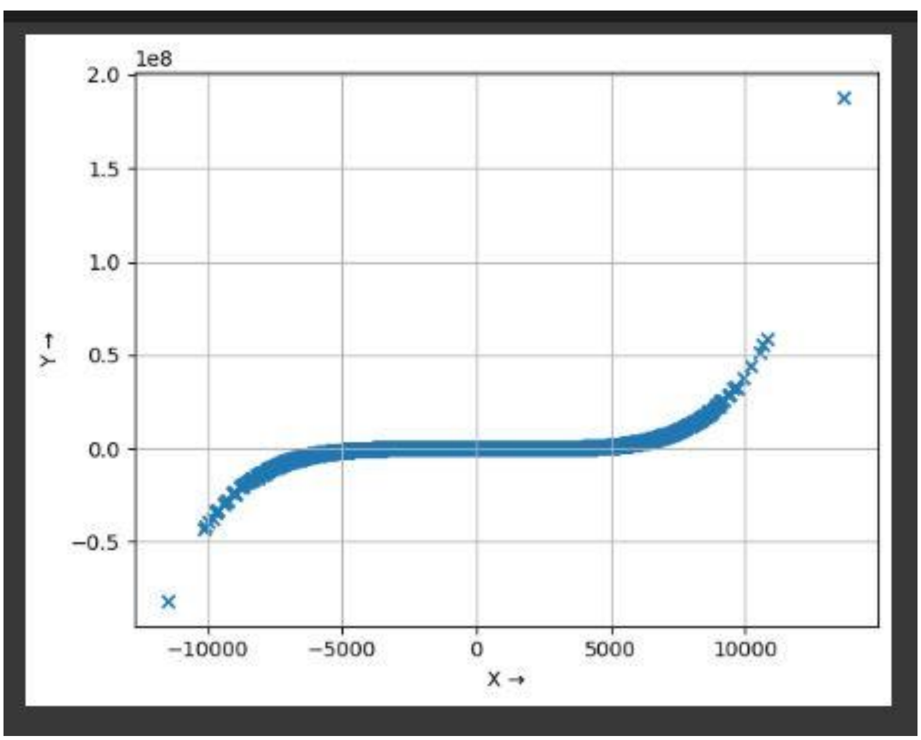
The Training dataset given to us has 50000 training examples and has 3 features by these three features we have to create polynomial terms and to create these polynomials by a create _polynomial function

```
class PolynomialRegression:
  def create_polynomial(X):
    m=X.shape[0]
    A=X[:,0]
    A=np.reshape(A,(m,1))
    B=X[:,1]
    B=np.reshape(B,(m,1))
    C=X[:,2]
    C=np.reshape(C,(m,1))
    X_Poly=np.zeros((m,1))
    n=int(input("Enter the degree of polynomial you want "))
    a=0
    # X_Poly_empty=np.empty((m,a))
    for i in range(n+1):
      for j in range(n+1-i):
        for k in range(n+1-j-i):
          if  i==0 and j==0 and k==0:
            a+=1
            # print(X_Poly.shape)
            continue
          else:
            a+=1
            X_Poly=np.append(X_Poly,((A)**i)*((B)**j)*((C)**k),axis=1)
            # print(X_Poly.shape)
    X_Poly = np.delete(X_Poly, 0, axis=1)
    return X_Poly
```

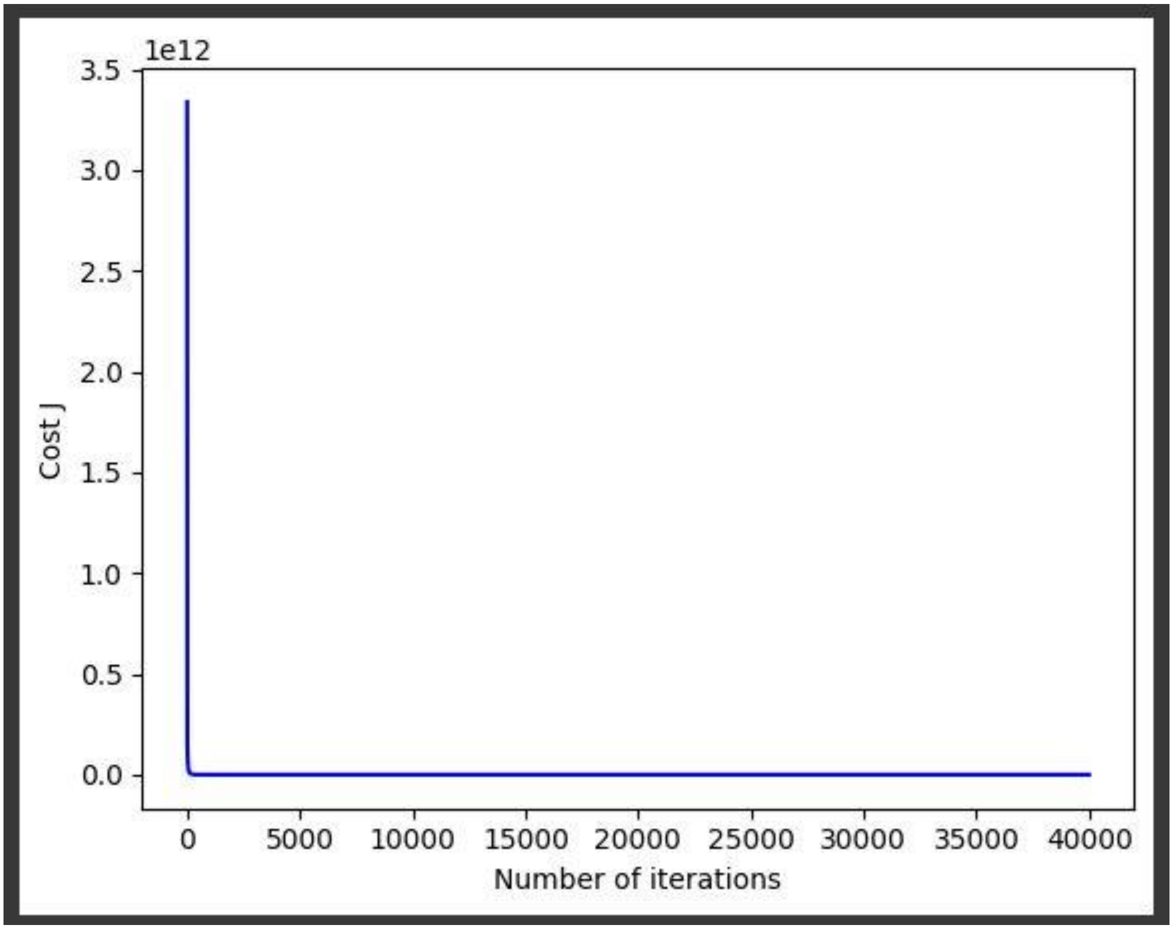
```
def Train_data(x,y,learning_rate,no_of_iterations,L):  #L is regularization constant
  m=x.shape[0]
  n=x.shape[1]
  w=(np.zeros((n,1)))
  b=0
  # for keeping cost data reserved
  cost=[]
  for i in range(no_of_iterations):
    y_pred=np.dot(x,w)+b
    cost_i = (1/(2*m))*np.sum(np.square(y_pred - y))+ (L/2*m)*np.sum(np.square(w))
    a=np.array(y_pred - y)
    a1 = a.transpose()
    derivative_w=(1/m)*np.dot(a1,x)
    derivative_w_T=derivative_w.transpose()
    derivative_b=(1/m)*np.sum(y_pred-y)
    w=w*(1-(learning_rate*L)/m)-learning_rate*derivative_w_T
    b=b-learning_rate*derivative_b
    cost.append(cost_i)
    if(i%math.ceil(no_of_iterations/400) == 0):
         print('Cost is:',cost_i,'after ',i,'iterations')
```



1e8
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Y →
−10000
−5000
0
5000
10000
X →



1e8
2.0
1.5
1.0
0.5
0.0
−0.5
Y →
−10000
−5000
0
5000
10000
X →

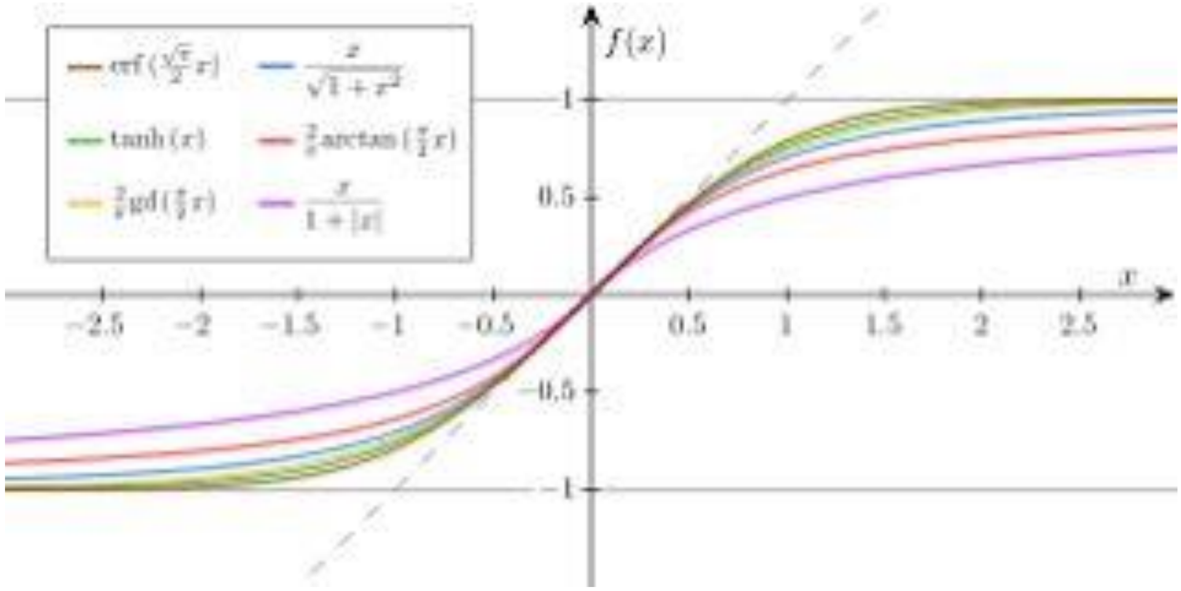


R2_SCORE: 0.99999999934

RMSE score: 5.65

LOGISTIC REGRESSION:

I have implemented logistic regression by one vs all approach by sigmoid activation function and trained each label separately by 1000 iterations and there I got accuracy of 83.6666778%



K NEAREST NEIGHBOUR:

K-Nearest Neighbors (KNN) is a non-parametric supervised learning algorithm used for classification and regression tasks in machine learning. In KNN, the classification of a new sample is based on the majority class among its k-nearest neighbors in the feature space.

Given a set of labeled training data, the KNN algorithm builds a model by memorizing the features and classes of the training samples. To classify a new sample, KNN finds the k closest training samples to the new sample in the feature space based on some distance metric (e.g., Euclidean distance), and then assigns the class label that is most common among the k neighbors to the new sample.

The value of k is a hyperparameter that determines the number of neighbors to consider. The KNN algorithm is simple and intuitive, but can be computationally expensive, especially for large datasets, because it requires calculating the distance between the new sample and all training samples for each prediction.

ACCURACY: 84.444448%

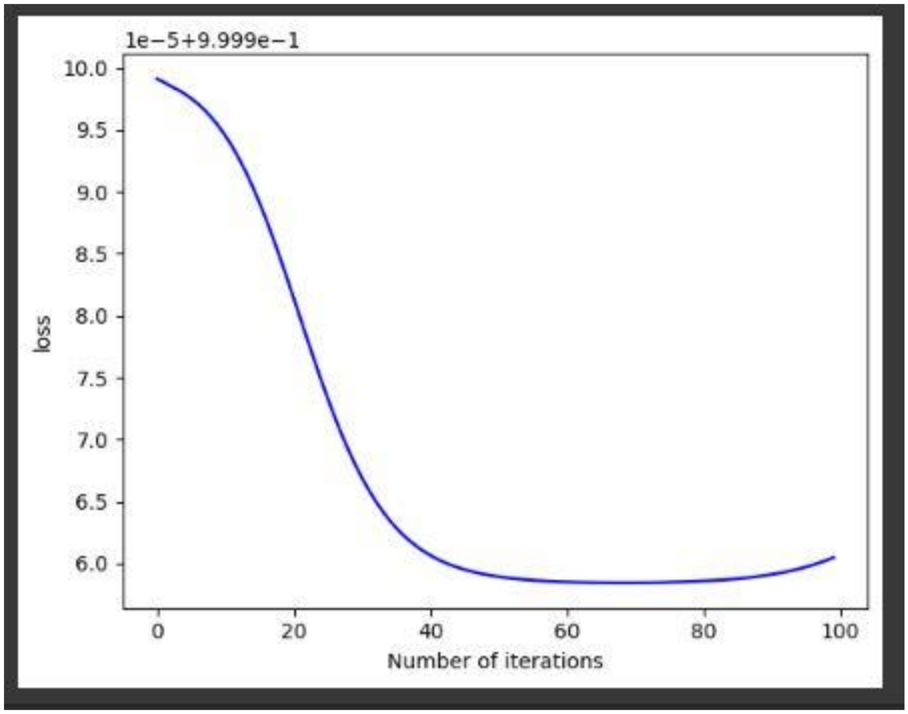
NEURAL NETWORK CLASSIFICATION

In neural network I have created a n layered neural network classification and applied forward propagation and backward propagation and applied Batch Gradient descent in this we provide a architecture for neural network then according to this I have initialized w and b parameters by random values then forward propagation starts then computed losses and then computed gradients by backward propagation.

ACCURACY: 39.5554%

NEURAL NETWORK LINEAR

NEURAL NETWORK POLYNOMIAL



1e−5+9.999e−1
10.0
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
loss
0
20
40
60
80
100
Number of iterations