ML Bootcamp

Anant Upadhyay
IIT (ISM) Dhanbad

Algorithms used:-

- Linear regression:-Train a linear model based on the training dataset using gradient descent.Find the line of best fit from the training dataset.
- Polynomial regression:-Train a polynomial model containing square,cube or higher powers of the features.
- Logistic regression:-Train the model using the sigmoid function.This is used for classification problems as then we can classify using a threshold.Eg:- a>=0.5 then outcome is positive otherwise negative.
- Neural networks:- Construct a n-layer neural network to make predictions on the given dataset.Then adjust the model's parameters by the help of back propagation.I will train the model by nudging weights and biases of each neuron.
- KNN:-This algorithm finds the "k" nearest neighbors of the datapoint that is to be predicted, and then bases its prediction on its nearest neighbors.
- Other parameters/procedures:- I have added an additional parameter λ to regularize our data(for dealing with bias and variance).Also for increasing speed and accuracy we can normalize the given data(If required)

Technology stack :-

- Basic knowledge of the python programming language
- Knowledge of the numpy library to easily work with arrays,vectors and matrices
- Knowledge of pandas library to load,edit and manipulate data
- Knowledge of the matplotlib library to show visualizations of our model using graphs,scatter plots etc.
- Knowing the usage of Jupyter/google Collab notebooks(to document our project and present it in the form of a report).Eg:-Usage of markup and code cells,running cells etc.
- Knowledge of classes and vectorization in python(To make implementation easier,faster and later convert the work in a python library)
- Knowledge of the working and implementation of different algorithms and also how to train them.(Knowing how to create models using:- linear regression,polynomial regression,logistic regression,KNN,neural networks).And also knowing how to train them (using gradient descent,backpropagation etc).
- Some basic knowledge of statistics(for feature scaling) and differential calculus(for the gradient descent and backpropagation algorithms)

Implementation Details Of the Project-

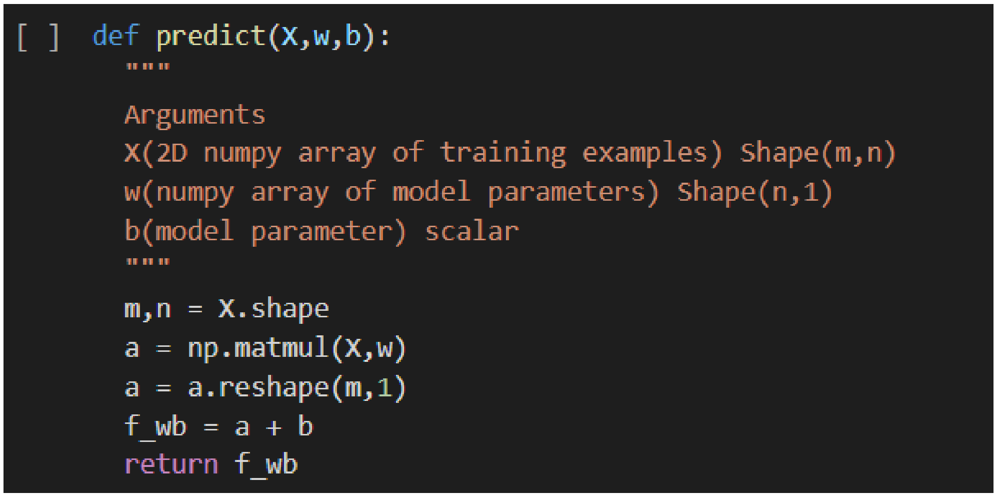
We would implement different algorithms on the training dataset such as linear regression,polynomial regression,logistic regression,KNN,neural networks and present it in the form of a report(Also convert it into a library if time permits).

Linear regression:-
Model:-

$$f_{w,b}(x^{(i)}) = w \cdot x^{(i)} + b$$

w(weight) and b(bias) are parameters

The predict function of my linear regression return the value fwb for all the training examples:-



```
[ ]  def predict(X,w,b):
        """

        Arguments
        X(2D numpy array of training examples) Shape(m,n)
        w(numpy array of model parameters) Shape(n,1)
        b(model parameter) scalar
        """

        m,n = X.shape
        a = np.matmul(X,w)
        a = a.reshape(m,1)
        f_wb = a + b
        return f_wb
```

(I have used matrix multiplication to multiply all features with their respective weights And added bias with the help of broadcasting).

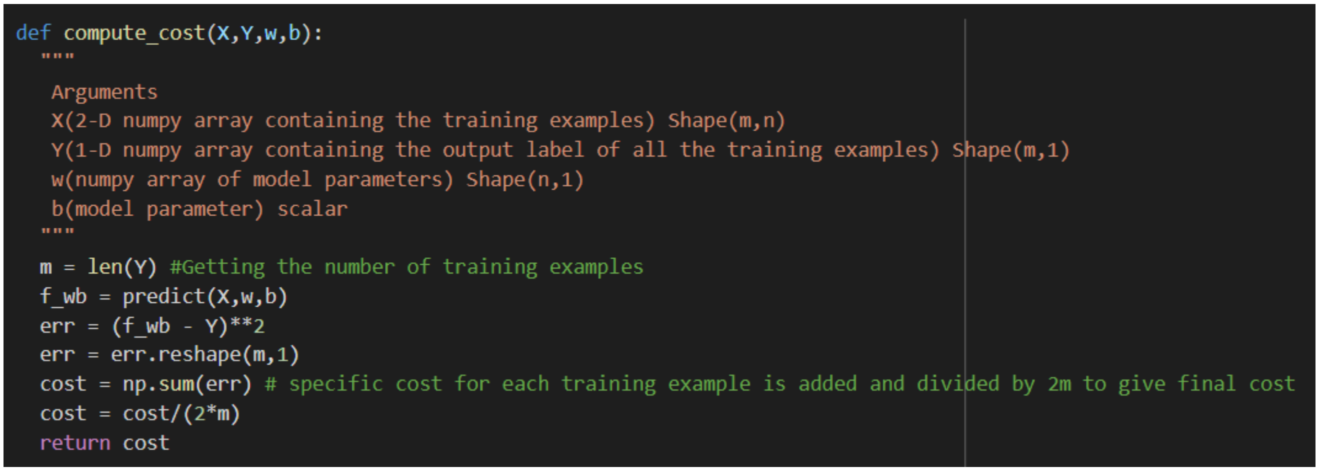
Cost function:-

We calculate the loss of each data point to measure how far our model is from the training example.Then we sum all the losses to create the cost function.

In actual implementation we use the vectorized version to implement mean square error cost.

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

$x^{(i)}, y^{(i)}$ represent the ith training example from our training dataset



```
def compute_cost(X,Y,w,b):
  """
   Arguments
   X(2-D numpy array containing the training examples) Shape(m,n)
   Y(1-D numpy array containing the output label of all the training examples) Shape(m,1)
   w(numpy array of model parameters) Shape(n,1)
   b(model parameter) scalar
  """
  m = len(Y) #Getting the number of training examples
  f_wb = predict(X,w,b)
  err = (f_wb - Y)**2
  err = err.reshape(m,1)
  cost = np.sum(err) # specific cost for each training example is added and divided by 2m to give final cost
  cost = cost/(2*m)
  return cost
```

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})$$

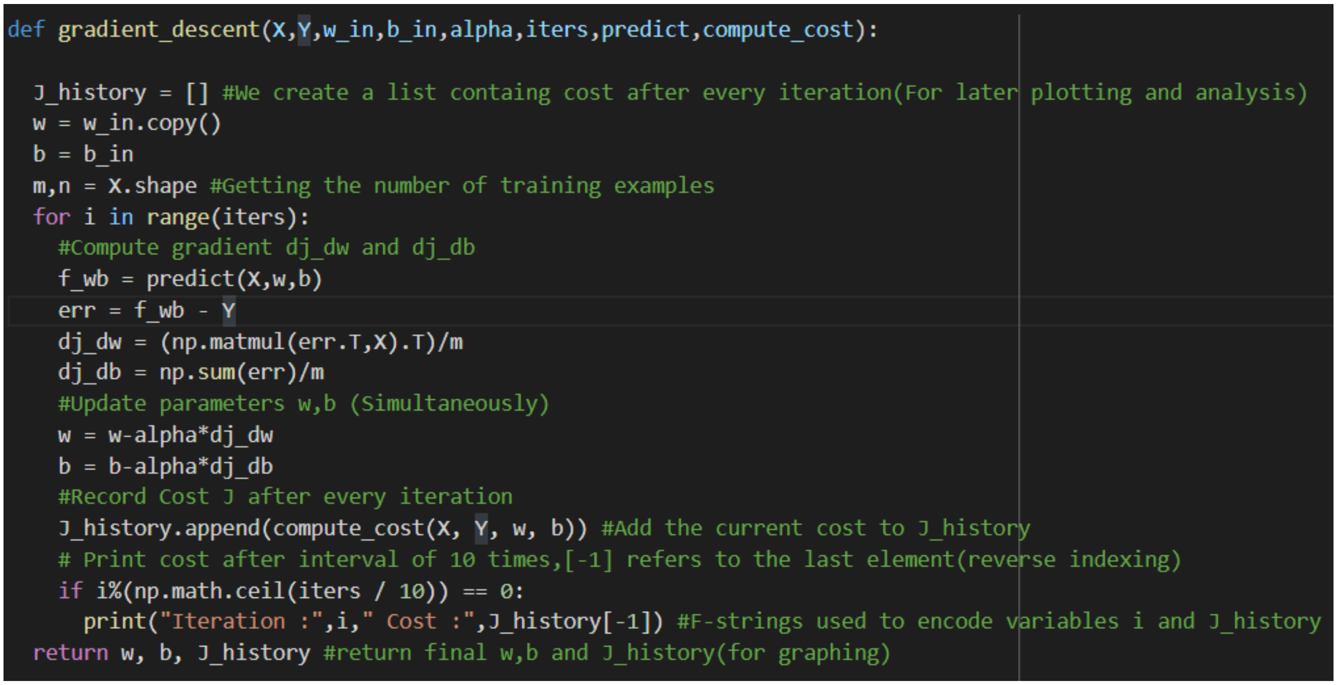
(I have used the mean-square-error metric(As shown above) to measure the cost of my model.The code is vectorized,for faster implementation so that the cost of each training example is calculated all at once.)

Training(By gradient descent):-

$$w = w - \alpha\frac{\partial J}{\partial w}$$

$$b = b - \alpha\frac{\partial J}{\partial b}$$

α is the learning rate.We simultaneously update w,b after each iteration. <u>To update Simultaneously we make a copy of w parameter,and after every iteration we store it in a list, J_train.</u>



```
def gradient_descent(X,Y,w_in,b_in,alpha,iters,predict,compute_cost):

  J_history = [] #We create a list containg cost after every iteration(For later plotting and analysis)
  w = w_in.copy()
  b = b_in
  m,n = X.shape #Getting the number of training examples
  for i in range(iters):
    #Compute gradient dj_dw and dj_db
    f_wb = predict(X,w,b)
    err = f_wb - Y
    dj_dw = (np.matmul(err.T,X).T)/m
    dj_db = np.sum(err)/m
    #Update parameters w,b (Simultaneously)
    w = w-alpha*dj_dw
    b = b-alpha*dj_db
    #Record Cost J after every iteration
    J_history.append(compute_cost(X, Y, w, b)) #Add the current cost to J_history
    # Print cost after interval of 10 times,[-1] refers to the last element(reverse indexing)
    if i%(np.math.ceil(iters / 10)) == 0:
      print("Iteration :",i," Cost :",J_history[-1]) #F-strings used to encode variables i and J_history
  return w, b, J_history #return final w,b and J_history(for graphing)
```

Polynomial regression:-

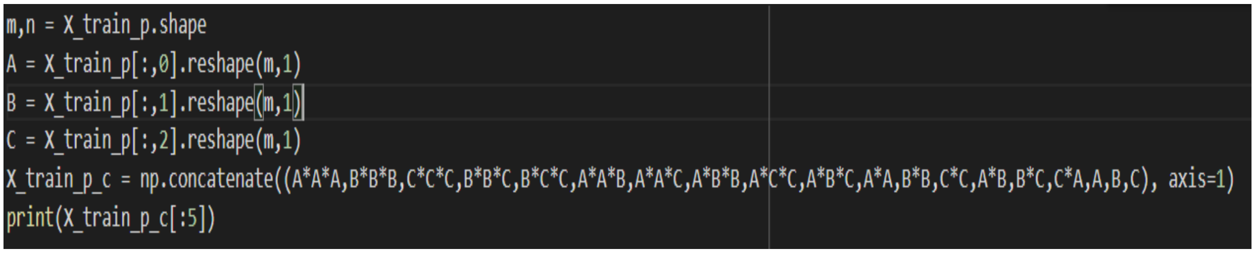
The implementation is almost same as shown for linear regression just the model is tweaked a little bit to account for higher powers of x.

Model:-

$$f_{w,b}(x^{(i)}) = w_1 \cdot x^{(i)} + w_2 \cdot (x^{(i)})^2 + w_3 \cdot (x^{(i)})^3 + b$$

More higher powers can also be used.Note the regularization parameter λ may be introduced if required.

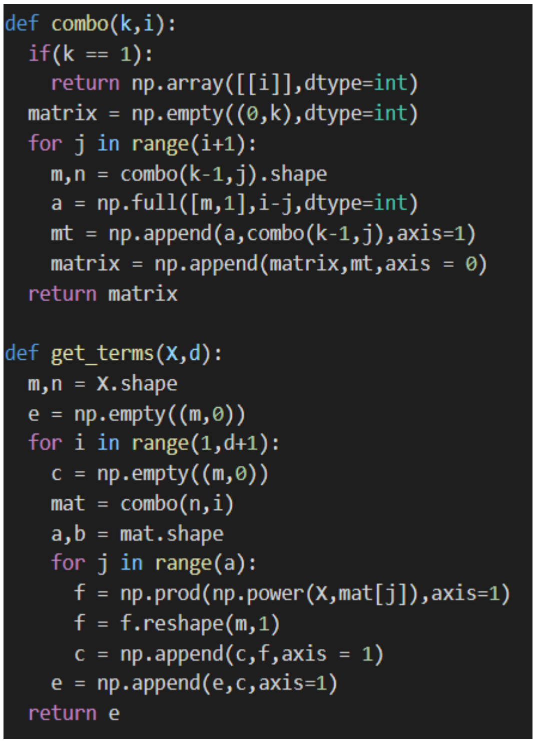
<u>In the model,the predict function and gradient descent is same as linear regression but we generate multiple terms for a 3 degree polynomial.</u>



```
m,n = X_train_p.shape
A = X_train_p[:,0].reshape(m,1)
B = X_train_p[:,1].reshape(m,1)
C = X_train_p[:,2].reshape(m,1)
X_train_p_c = np.concatenate((A*A*A,B*B*B,C*C*C,B*B*C,B*C*C,A*A*B,A*A*C,A*B*B,A*C*C,A*B*C,A*A,B*B,C*C,A*B,B*C,C*A,A,B,C), axis=1)
print(X_train_p_c[:5])
```

(All the different terms are concatenated along with A,B,C in one matrix of size (m,19) where m Is the number of training examples).

I have also Implemented an algorithm to generalize this process for n degree polynomial, known as get_terms().



```
def combo(k,i):
  if(k == 1):
    return np.array([[i]],dtype=int)
  matrix = np.empty((0,k),dtype=int)
  for j in range(i+1):
    m,n = combo(k-1,j).shape
    a = np.full([m,1],i-j,dtype=int)
    mt = np.append(a,combo(k-1,j),axis=1)
    matrix = np.append(matrix,mt,axis = 0)
  return matrix

def get_terms(X,d):
  m,n = X.shape
  e = np.empty((m,0))
  for i in range(1,d+1):
    c = np.empty((m,0))
    mat = combo(n,i)
    a,b = mat.shape
    for j in range(a):
      f = np.prod(np.power(X,mat[j]),axis=1)
      f = f.reshape(m,1)
      c = np.append(c,f,axis = 1)
    e = np.append(e,c,axis=1)
  return e
```

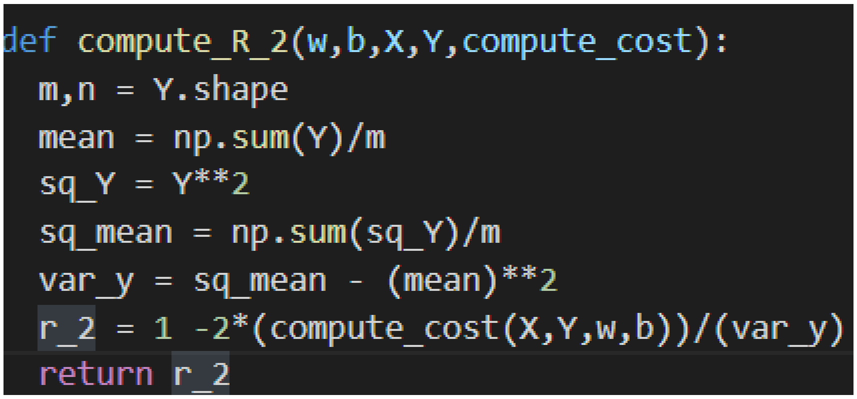
The function get_terms() makes use of another function known as combo() which gives the Pairs that satisfy the equation:-

$a_1 + a_2 ... + a_k = i$

Where a is a whole number.The pairs satisfying this equation are then used to generate the Polynomial terms.

Minimum cost is obtained at n=5 degree polynomial

Computing R2 score:-



```python
def compute_R_2(w,b,X,Y,compute_cost):
  m,n = Y.shape
  mean = np.sum(Y)/m
  sq_Y = Y**2
  sq_mean = np.sum(sq_Y)/m
  var_y = sq_mean - (mean)**2
  r_2 = 1 -2*(compute_cost(X,Y,w,b))/(var_y)
  return r_2
```

Formula of r2 score is:-

$$r^2 = 1 - \frac{\sum_{i=1}^{m} (f^{(i)}_{w,b} - y^{(i)})^2}{\sum_{i=1}^{m} (y^{(i)} - \bar{y})^2}$$

Which is then simplified to
R_2 = 1-2*(mse_cost/variance_of_y)

Logistic regression:-

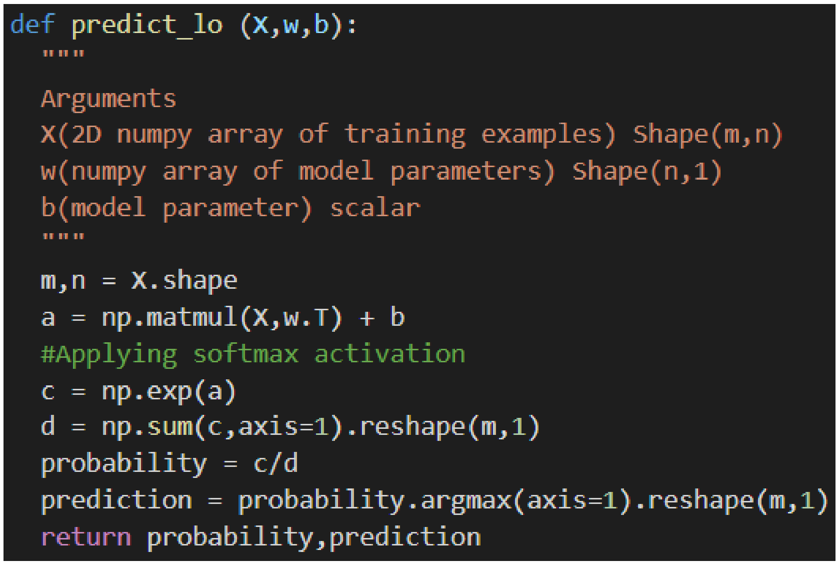
Here we use the sigmoid function to adress classification problems.In such problems the input is Some real number and the output can be represented as a 0 or 1.

<u>I have to deal with multiple classes so I have used softmax activation.</u>

Model:-

$$f_{w,b}(x^{(i)}) = \frac{e^{kj}}{\sum_{o=1}^{n} e^{ko}}$$

$$ko = w_o \cdot (x) + b_o$$



```python
def predict_lo (X,w,b):
  """
  Arguments
  X(2D numpy array of training examples) Shape(m,n)
  w(numpy array of model parameters) Shape(n,1)
  b(model parameter) scalar
  """
  m,n = X.shape
  a = np.matmul(X,w.T) + b
  #Applying softmax activation
  c = np.exp(a)
  d = np.sum(c,axis=1).reshape(m,1)
  probability = c/d
  prediction = probability.argmax(axis=1).reshape(m,1)
  return probability,prediction
```

Here we use a different kind of loss function to penalize the model(Known as cross-entropy loss).

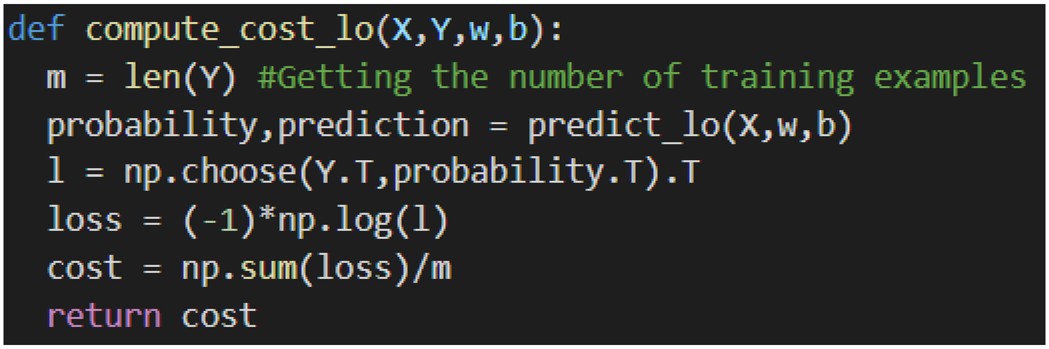
Loss function:-

$L(f_{w,b}(x^{(i)}), y^{(i)}) =- \ ln(f_{w,b}(x^{(i)}))$ **where j=Y[i]**

Cost function:-

$$J(w, b) \ = \frac{1}{m}\sum_{i=1}^{m} L(f_{w,b}(x^{(i)}), y^{(i)})$$

Compute cost for logistic:-



```
def compute_cost_lo(X,Y,w,b):
  m = len(Y) #Getting the number of training examples
  probability,prediction = predict_lo(X,w,b)
  l = np.choose(Y.T,probability.T).T
  loss = (-1)*np.log(l)
  cost = np.sum(loss)/m
  return cost
```

Training(By gradient descent):-

$$w \ = w \ - \ \alpha\frac{\partial J}{\partial w}$$

$$b \ = \ b \ - \ \alpha\frac{\partial J}{\partial b}$$

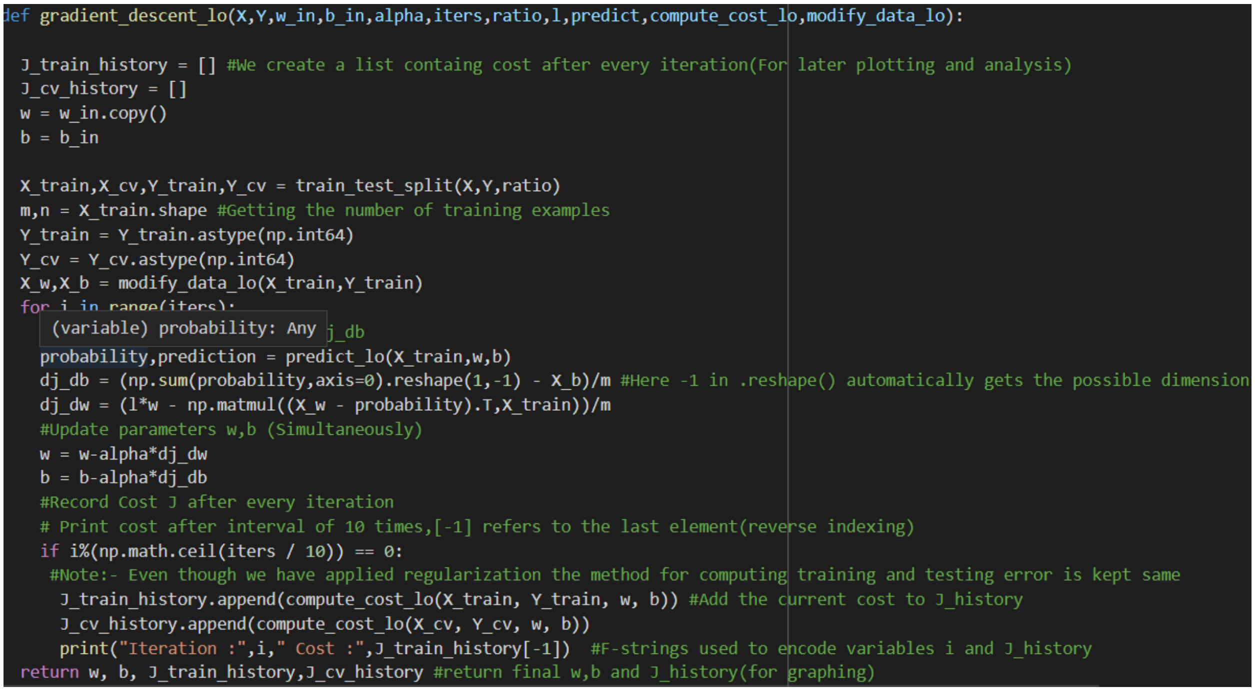
$$\frac{\partial J}{\partial w} \ = \frac{1}{m}\sum(prob \ \ - \ \ Yi) \ \cdot \frac{\partial k}{\partial w}$$

$$\frac{\partial k}{\partial w} \ = x^{(i)}$$

$$\frac{\partial J}{\partial w} \ = \frac{1}{m}\sum(prob \ \ - \ \ Yi)$$

Here Yi is the one-hot encoded version of the output classes
Eg:- if initially Yi=np.array([[1],[0],[2]]) then after one-hot encoding
Yi = np.array(([0,1,0],[1,0,0],[0,0,1]))

Gradient descent for logistic:-



```
def gradient_descent_lo(X,Y,w_in,b_in,alpha,iters,ratio,l,predict,compute_cost_lo,modify_data_lo):

  J_train_history = [] #We create a list containg cost after every iteration(For later plotting and analysis)
  J_cv_history = []
  w = w_in.copy()
  b = b_in

  X_train,X_cv,Y_train,Y_cv = train_test_split(X,Y,ratio)
  m,n = X_train.shape #Getting the number of training examples
  Y_train = Y_train.astype(np.int64)
  Y_cv = Y_cv.astype(np.int64)
  X_w,X_b = modify_data_lo(X_train,Y_train)
  for i in range(iters):
    (variable) probability: Any j_db
    probability,prediction = predict_lo(X_train,w,b)
    dj_db = (np.sum(probability,axis=0).reshape(1,-1) - X_b)/m #Here -1 in .reshape() automatically gets the possible dimension
    dj_dw = (l*w - np.matmul((X_w - probability).T,X_train))/m
    #Update parameters w,b (Simultaneously)
    w = w-alpha*dj_dw
    b = b-alpha*dj_db
    #Record Cost J after every iteration
    # Print cost after interval of 10 times,[-1] refers to the last element(reverse indexing)
    if i%(np.math.ceil(iters / 10)) == 0:
     #Note:- Even though we have applied regularization the method for computing training and testing error is kept same
      J_train_history.append(compute_cost_lo(X_train, Y_train, w, b)) #Add the current cost to J_history
      J_cv_history.append(compute_cost_lo(X_cv, Y_cv, w, b))
      print("Iteration :",i," Cost :",J_train_history[-1])  #F-strings used to encode variables i and J_history
  return w, b, J_train_history,J_cv_history #return final w,b and J_history(for graphing)
```

The gradient-descent used here is vectorized inorder to achieve faster results.

Neural Network:-

We have multiple layers of neuron in a neural network.The input is 1st propagated through the Input layer.Each neuron takes the activation of all neurons in it's previous layer as input and Uses it to compute its own activation using its activation function.The activation function of all neurons(except the output neuron) are Relu function.The activation function of output layer is Chosen according to the requirement of the problem.

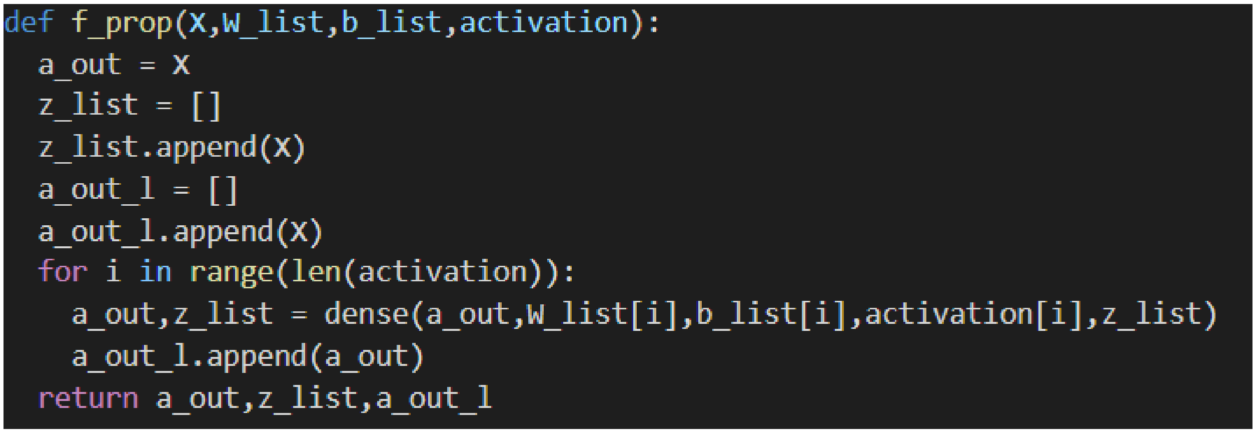
The neural network computes the output by forward propogation.

Forward propagation:-

$$a_j^{[L]} = g(\vec{w}_j^{[L]} \cdot \vec{a}^{[L-1]} + b_j^{[L]})$$

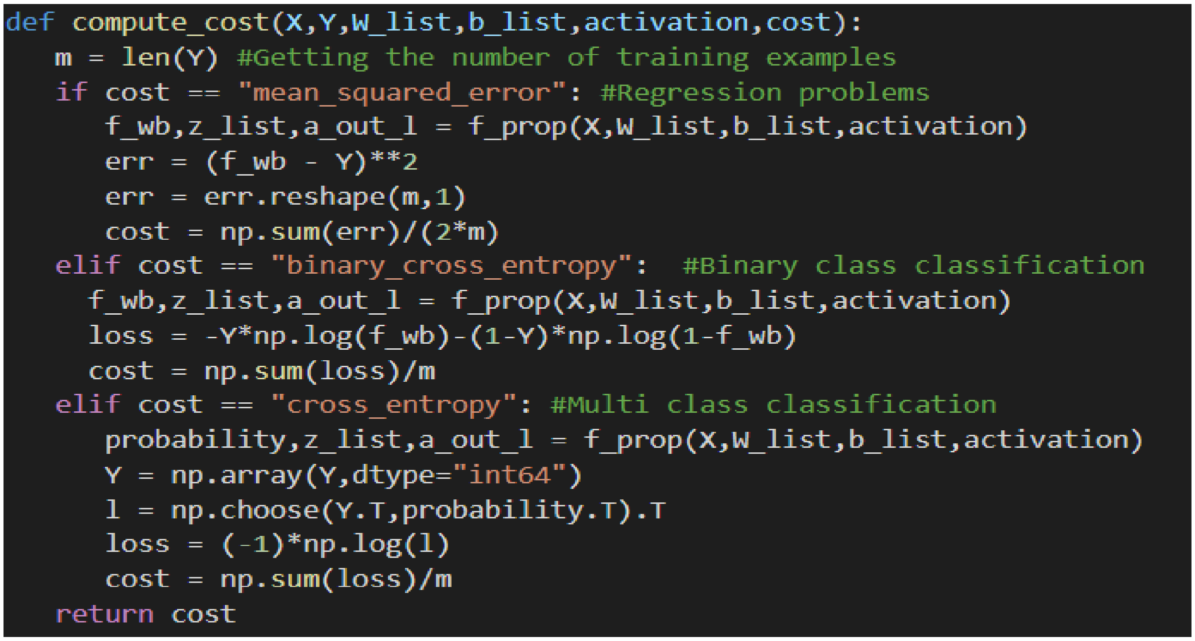
Above equation represents the activation of the jth neuron in the Lth layer of the neural network. Here g(x) represents the activation function of the neuron.

We have to recursively pass the output through each layer of the neural network.Which is Accomplished by forward propagation.



```
def f_prop(X,W_list,b_list,activation):
  a_out = X
  z_list = []
  z_list.append(X)
  a_out_l = []
  a_out_l.append(X)
  for i in range(len(activation)):
    a_out,z_list = dense(a_out,W_list[i],b_list[i],activation[i],z_list)
    a_out_l.append(a_out)
  return a_out,z_list,a_out_l
```

Computing cost of neural network:-



```
def compute_cost(X,Y,W_list,b_list,activation,cost):
   m = len(Y) #Getting the number of training examples
   if cost == "mean_squared_error": #Regression problems
      f_wb,z_list,a_out_l = f_prop(X,W_list,b_list,activation)
      err = (f_wb - Y)**2
      err = err.reshape(m,1)
      cost = np.sum(err)/(2*m)
   elif cost == "binary_cross_entropy":  #Binary class classification
     f_wb,z_list,a_out_l = f_prop(X,W_list,b_list,activation)
     loss = -Y*np.log(f_wb)-(1-Y)*np.log(1-f_wb)
     cost = np.sum(loss)/m
   elif cost == "cross_entropy": #Multi class classification
      probability,z_list,a_out_l = f_prop(X,W_list,b_list,activation)
      Y = np.array(Y,dtype="int64")
      l = np.choose(Y.T,probability.T).T
      loss = (-1)*np.log(l)
      cost = np.sum(loss)/m
   return cost
```

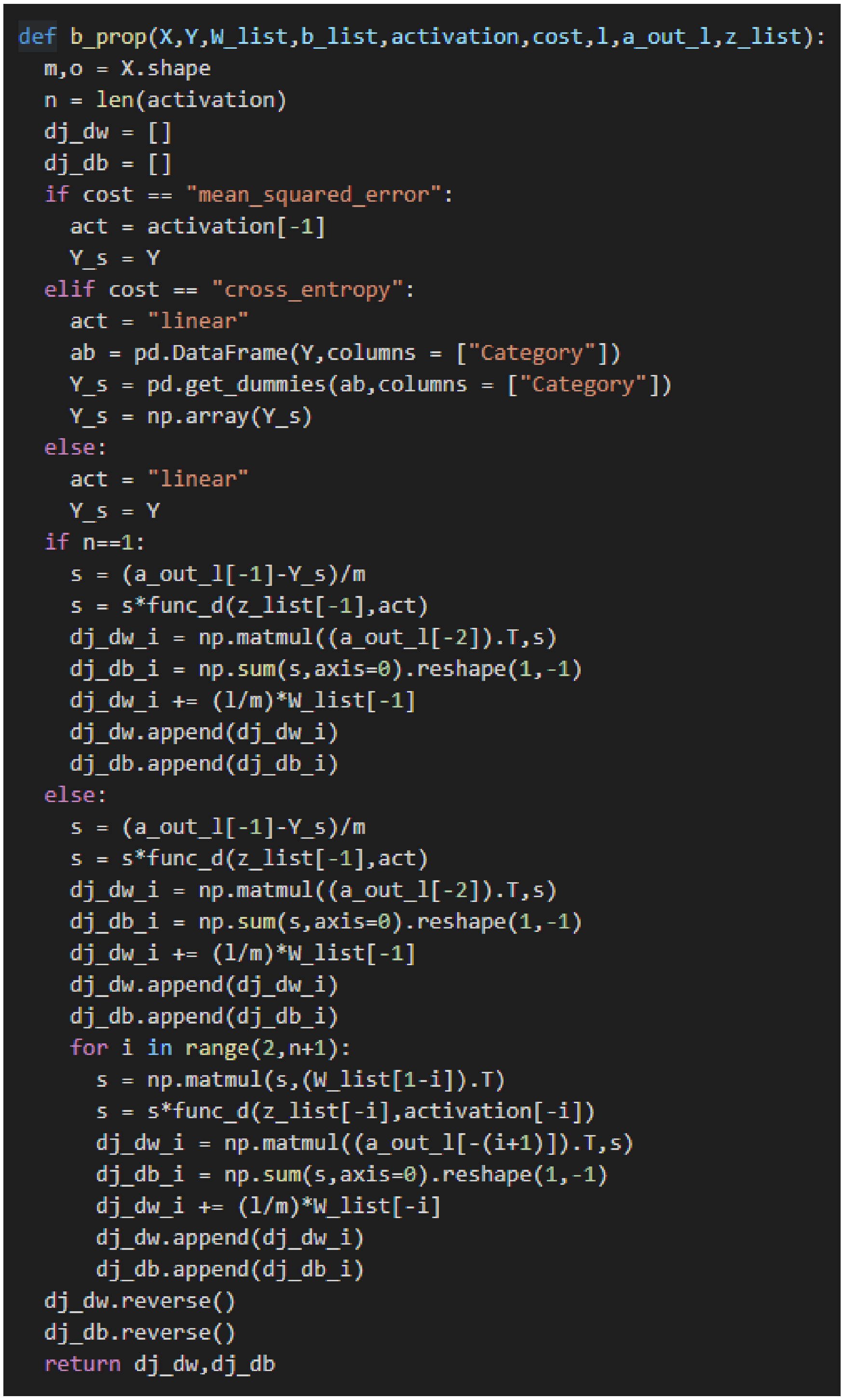
The cost function depends on the kind of model being trained.Eg:-For regression problems the Cost function is generally mean_squared_error.

Training(By gradient descent):-

$$w = w - \alpha\frac{\partial J}{\partial w}$$

$$b = b - \alpha\frac{\partial J}{\partial b}$$

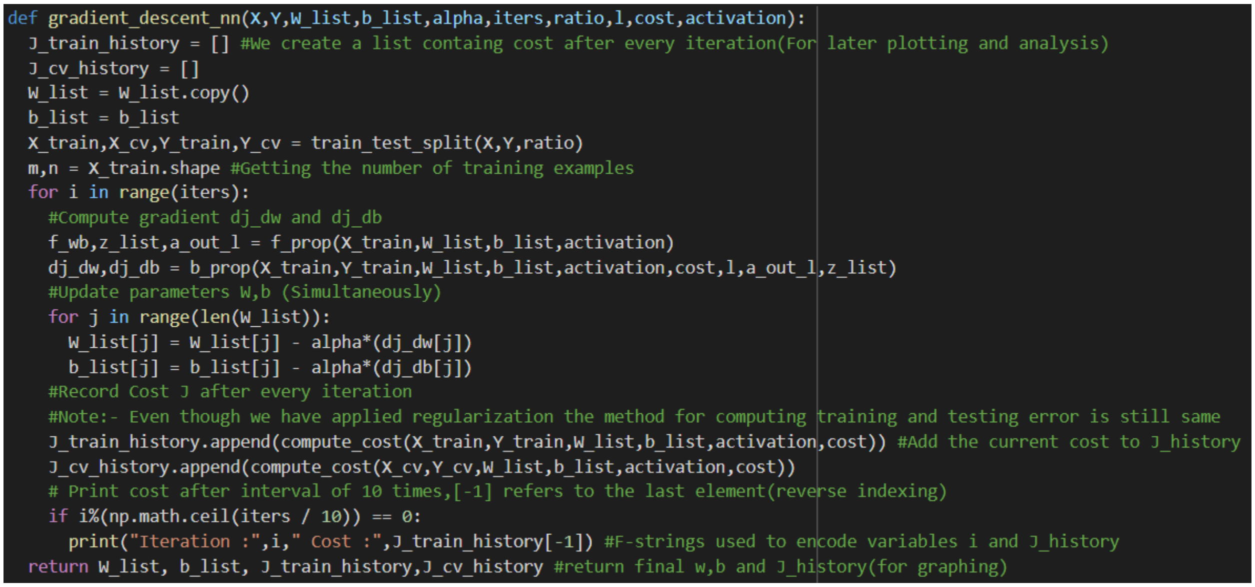
The updation of weights and biases is done using the back-propagation algorithm:-



```python
def b_prop(X,Y,W_list,b_list,activation,cost,l,a_out_l,z_list):
  m,o = X.shape
  n = len(activation)
  dj_dw = []
  dj_db = []
  if cost == "mean_squared_error":
    act = activation[-1]
    Y_s = Y
  elif cost == "cross_entropy":
    act = "linear"
    ab = pd.DataFrame(Y,columns = ["Category"])
    Y_s = pd.get_dummies(ab,columns = ["Category"])
    Y_s = np.array(Y_s)
  else:
    act = "linear"
    Y_s = Y
  if n==1:
    s = (a_out_l[-1]-Y_s)/m
    s = s*func_d(z_list[-1],act)
    dj_dw_i = np.matmul((a_out_l[-2]).T,s)
    dj_db_i = np.sum(s,axis=0).reshape(1,-1)
    dj_dw_i += (l/m)*W_list[-1]
    dj_dw.append(dj_dw_i)
    dj_db.append(dj_db_i)
  else:
    s = (a_out_l[-1]-Y_s)/m
    s = s*func_d(z_list[-1],act)
    dj_dw_i = np.matmul((a_out_l[-2]).T,s)
    dj_db_i = np.sum(s,axis=0).reshape(1,-1)
    dj_dw_i += (l/m)*W_list[-1]
    dj_dw.append(dj_dw_i)
    dj_db.append(dj_db_i)
    for i in range(2,n+1):
      s = np.matmul(s,(W_list[1-i]).T)
      s = s*func_d(z_list[-i],activation[-i])
      dj_dw_i = np.matmul((a_out_l[-(i+1)]).T,s)
      dj_db_i = np.sum(s,axis=0).reshape(1,-1)
      dj_dw_i += (l/m)*W_list[-i]
      dj_dw.append(dj_dw_i)
      dj_db.append(dj_db_i)
  dj_dw.reverse()
  dj_db.reverse()
  return dj_dw,dj_db
```

I have created a list dj_dw,dj_db which stores all the updates for the weights and biases.The list Is then reversed so that during update each dj_dw(j) matches with its corresponding w(j).

Gradient descent algorithm for neural network:-



```python
def gradient_descent_nn(X,Y,W_list,b_list,alpha,iters,ratio,l,cost,activation):
  J_train_history = [] #We create a list containg cost after every iteration(For later plotting and analysis)
  J_cv_history = []
  W_list = W_list.copy()
  b_list = b_list
  X_train,X_cv,Y_train,Y_cv = train_test_split(X,Y,ratio)
  m,n = X_train.shape #Getting the number of training examples
  for i in range(iters):
    #Compute gradient dj_dw and dj_db
    f_wb,z_list,a_out_l = f_prop(X_train,W_list,b_list,activation)
    dj_dw,dj_db = b_prop(X_train,Y_train,W_list,b_list,activation,cost,l,a_out_l,z_list)
    #Update parameters W,b (Simultaneously)
    for j in range(len(W_list)):
      W_list[j] = W_list[j] - alpha*(dj_dw[j])
      b_list[j] = b_list[j] - alpha*(dj_db[j])
    #Record Cost J after every iteration
    #Note:- Even though we have applied regularization the method for computing training and testing error is still same
    J_train_history.append(compute_cost(X_train,Y_train,W_list,b_list,activation,cost)) #Add the current cost to J_history
    J_cv_history.append(compute_cost(X_cv,Y_cv,W_list,b_list,activation,cost))
    # Print cost after interval of 10 times,[-1] refers to the last element(reverse indexing)
    if i%(np.math.ceil(iters / 10)) == 0:
      print("Iteration :",i," Cost :",J_train_history[-1]) #F-strings used to encode variables i and J_history
  return W_list, b_list, J_train_history,J_cv_history #return final w,b and J_history(for graphing)
```

KNN:-

First we find the distance of the k nearest neighbors from the point,using the formula:-

$$d(x,y) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

Here x,y are the data points and x_i and y_i represent the ith feature of x and y.

Note:- In actual implementation square root is ignored,to speed up computation

For classification:-

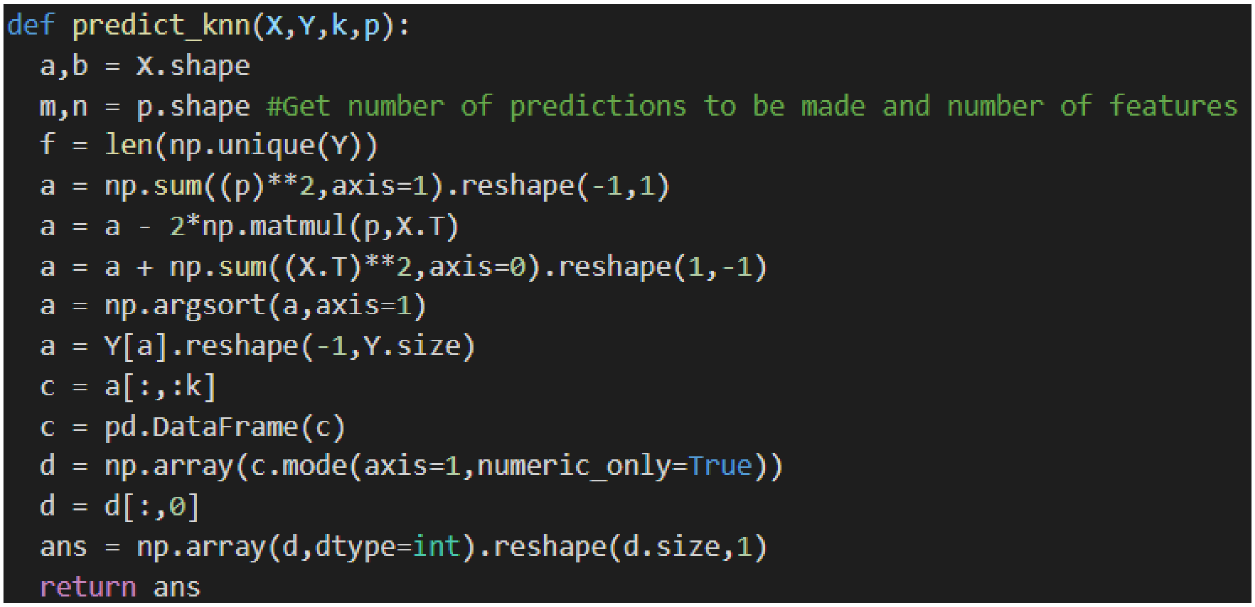
We classify the label by seeing the class which is most common to its k nearest neighbors.

For regression:-

We return the output as an average of its k nearest neighbors

$$y_{predicted} = \frac{\sum_i y_i}{k}$$

In the project we use knn to make a prediction using the classification dataset.



```
def predict_knn(X,Y,k,p):
  a,b = X.shape
  m,n = p.shape #Get number of predictions to be made and number of features
  f = len(np.unique(Y))
  a = np.sum((p)**2,axis=1).reshape(-1,1)
  a = a - 2*np.matmul(p,X.T)
  a = a + np.sum((X.T)**2,axis=0).reshape(1,-1)
  a = np.argsort(a,axis=1)
  a = Y[a].reshape(-1,Y.size)
  c = a[:,:k]
  c = pd.DataFrame(c)
  d = np.array(c.mode(axis=1,numeric_only=True))
  d = d[:,0]
  ans = np.array(d,dtype=int).reshape(d.size,1)
  return ans
```

In the predict function we take the (a,b) matrix of X.(a is number of training examples,and b Is number of features)and the (m,n) matrix p(where m is number of predictions and n is Number of features).Also b=n,Using these 2 matrices we get the matrix a of shape (a,m),which Stores the distances of each prediction from each training example.Then we sort the matrix and See the first the first k columns to base out prediction on(This is achieved using the pandas Function for mode).

Development process:-

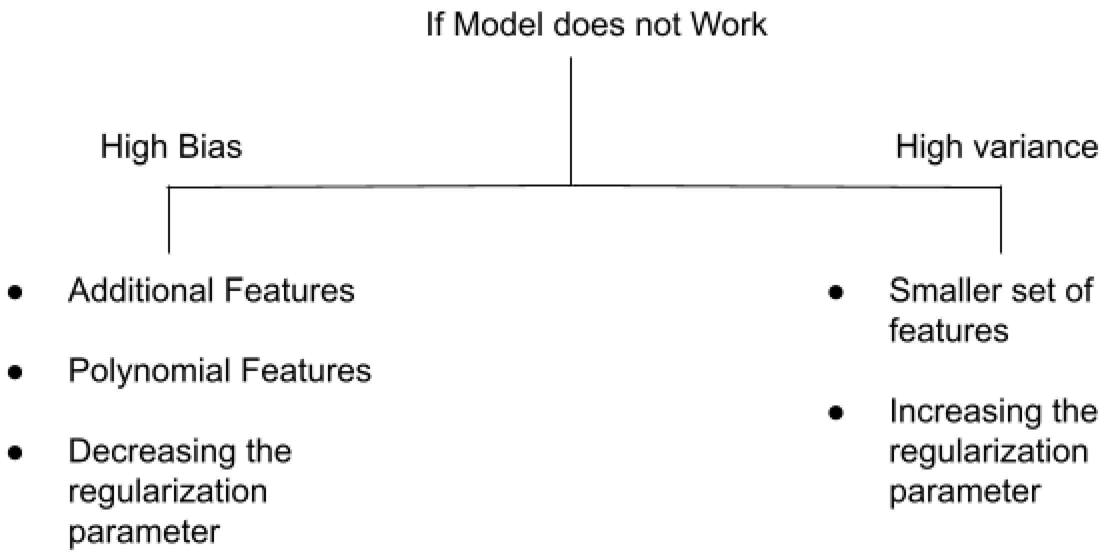
Just creating a model and training it on the training data(Using gradient descent) is not enough to ensure a successful model.For example we may need to increase/decrease the learning rate α to an optimum level so that training of model is not too slow and also ensure our model is not overshooting.So to effectively develop our model we split our training data into two parts $J_{train}$and J_{cv}.After training our data on J_{train}we test it on J_{cv}.

By using J_{train} and J_{cv} we draw the following conclusions:-

- If J_{train} and J_{cv} both are high the model has **high bias**(model is underfitting)

- If J_{train} is low but J_{cv} is high model has **high variance**(model is overfitting)

Flowchart for dealing with bias and variance:-



If Model does not Work

High Bias

- Additional Features
- Polynomial Features
- Decreasing the regularization parameter

High variance

- Smaller set of features
- Increasing the regularization parameter

Note here λ is the regularization parameter.This parameter is introduced in the cost function to avoid the problem of overfitting(**High variance**).Regularization keeps the values of all the Weights low in order to avoid overfitting.

Implementation:-

$$J(\vec{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^{2}$$

As you can see an additional regularization term is introduced in the cost function $J(\vec{w}, b)$.During gradient descent we consider the regularized cost function.

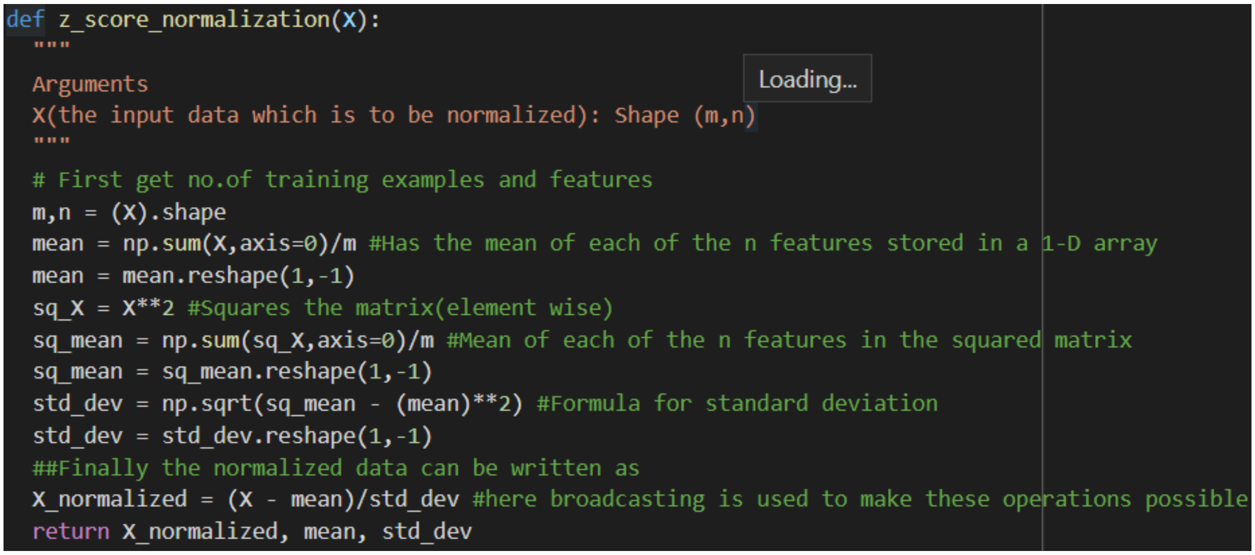
Feature engineering(If required):-

We could add/remove some features to adress underfitting and overfitting.If the range of data varies a lot among given features we could normalize our data for faster and more accurate results.

Z-score normalization:-

$$z = \frac{x-\mu}{\sigma}$$

Here z is the normalized data,x is the original data,μ is the mean of the data and σ is the Standard deviation of the data.



```
def z_score_normalization(X):
    """
    Arguments
    X(the input data which is to be normalized): Shape (m,n)
    """
    # First get no.of training examples and features
    m,n = (X).shape
    mean = np.sum(X,axis=0)/m #Has the mean of each of the n features stored in a 1-D array
    mean = mean.reshape(1,-1)
    sq_X = X**2 #Squares the matrix(element wise)
    sq_mean = np.sum(sq_X,axis=0)/m #Mean of each of the n features in the squared matrix
    sq_mean = sq_mean.reshape(1,-1)
    std_dev = np.sqrt(sq_mean - (mean)**2) #Formula for standard deviation
    std_dev = std_dev.reshape(1,-1)
    ##Finally the normalized data can be written as
    X_normalized = (X - mean)/std_dev #here broadcasting is used to make these operations possible
    return X_normalized, mean, std_dev
```

z_score_normalization is applied in all algorithms(except of knn).This is done so that all Features belong in the same range so that it can lead to faster gradient descent.

References:-

https://www.coursera.org (As I did 2 courses on machine learning from this site)
https://numpy.org/doc/stable/reference/ (For checking how different numpy functions worked)
https://pandas.pydata.org/docs/reference/api/ (For checking how different numpy functions worked)
Sites used for additional reference and learn new topics like K-means and decision trees
https://www.geeksforgeeks.org/
https://www.analyticsvidhya.com/
https://medium.com/

About Me:

- **Personal Details**

Name:- Anant Upadhyay
Branch:- 1st year,B-tech in Computer Science and Engineering
Admission No:- 22JE0110
D.O.B:- 24/11/2004

- 1 star coder in codechef :- https://www.codechef.com/users/anant_up11
- Made a project on the game tic tac toe:- https://github.com/Anant-Upadhyay/Tictactoe
- Github profile:-https://github.com/Anant-Upadhyay
- Completion of 1st course on Coursera:- https://coursera.org/share/38b6e1ab1e5d6b6ef796f0df070c337b
- Completion of 2nd course on Coursera:- https://www.coursera.org/account/accomplishments/certificate/LVZBLU7NC9NP

- **Why should I be selected for this Cyberlabs?**

I am a keen learner with a childhood interest in machine learning.I look forward for pursuing a career in machine learning and artificial intelligence as these topics have always fascinated me.Being part of Cyberlabs will allow me to explore these fields deeper and explore topics like

Computer vision,NLP,Supervised Learning,Unsupervised Learning etc.Being in cyberlabs will provide me with a team of like-minded people which will help in my personal growth,as well as the growth of the cyberlabs team.I would also like to participate in hackathons and discuss research papers on machine learning as I have always been fascinated by machine learning.