ASSIGNMENT 1 Roll No. MT2021055

03-10-2021

Regression Task On Student Performance Dataset

Regression task - Predict student performance in secondary education (attribute named G3) in Student Performance Dataset

We have the following Datasets in CSV format and their details in TXT format

- student.csv - Student Performance Dataset
- student.txt - Dataset description for student data

PREPROCESSING PHASE:-

- First we are checking and verifying our dataset that its in good format with heading or not by introducing the dataframe in the Pandas library.

- Then we are checking with the NULL or NAN values if it available or not in our datasets if it is available the first we have to tackle that

- So in my datasets there are total of 395 rows of data of performance of students are stored in which the age columns are having 15 NAN values which I am verifying by the isna().sum() function.

- Now what my goal is to just remove that NAN values or replace it with some meaningful values so what I am doing is I am using the information given in the description text of datasets that in our datasets only the students of age between (17-23) are present so what I am doing is replacing all the NAN values with the mean age of the students.

- After that further again I am checking that is there any rows of column are filled with “?” values so in this there is no rows of any columns are present where there are this “?” values.
- So now since first I am applying the univariate linear Regression on my datasets so I am picking up only one features as my feature vector and we have target vector as given in assignment is G3 marks (whixh is final grade marks of maths)
- So I am picking my feature vector as one of the highly corelated columns with our target vector or which I have to predict
- So I am doing df.corr() for checking the correlation with our target data with all the other columns
- So I am choosing the feature vector as the mean of G1 and G2 marks by adding the also as one of the column of our datsaets
- So my preprocessing is done actually for this datasets

APPLYING LINEAR REGRESSION ON SINGLE FEATURES:-

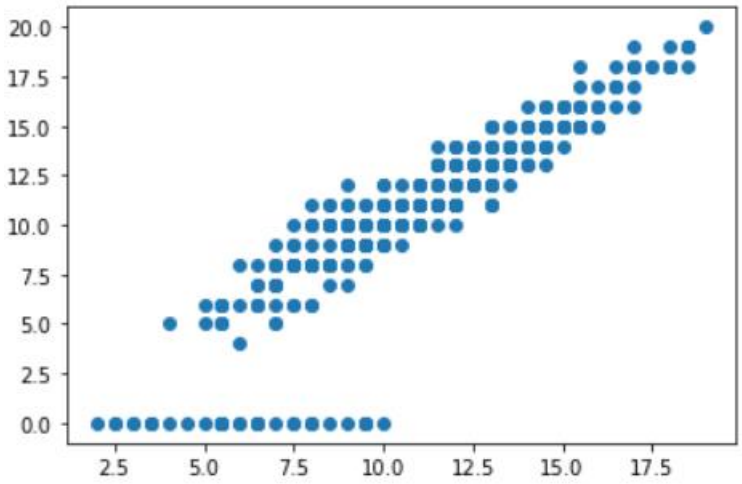
(UNIVARIATE LINEAR REGRESSION)

In statistics, linear regression is a linear approach to modelling the relationship between a dependent variable(target variable= Y) and one or more independent variables(Feature vectors=X). Let X be the independent variable and Y be the dependent variable. We will define a linear relationship between these two variables as follows:

$$Y = mX+c$$

Our challenge is to determine the value of m and c, such that the line corresponding to those values is the best fitting line or gives the minimum error.

So what I am doing is first plotting the points of X(single features) and Y(target features) in the 2-D space :-



CLOSED FORM SOLUTION NOW I AM APPROACHING:-

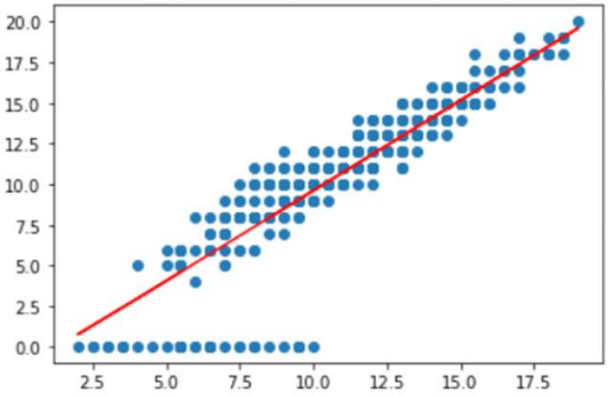
WE WILL CREATE 395x2 MATRIX then assignimg X at the 0th column and 1(coeficient of constant term=c) at the 1st column position

$$\theta = (X^T X)^{-1} X^T \vec{y}.$$

So we are directly using this formulae for closed form solution directly we will come to minimal cost function where the mean squared error will be about 0% so we are applying this in our assignment which is given below:-

$$E = \frac{1}{n} \sum_{i=0}^{n} (y_i - \bar{y}_i)^2$$

```
In [138]: m_disc, b_disc = np.linalg.inv(X_dummied.T@X_dummied)@(X_dummied.T@Y)

In [139]: m_disc, b_disc

Out[139]: (1.1963479922036786, -2.5189977233058434)

In [197]: Yhat = m_disc*X + b_disc
          plt.scatter(X, Y)
          plt.plot(X, Yhat, color="red")
          plt.show()
```



GRADIENT DESCENT SOLUTION:-

So now we are going to the gradient descent solution where what we will do is by the help of iteration we are reaching to the nearby 0% mean squared error

Loss Function

The loss is the error in our predicted value of m and c. Our goal is to minimize this error to obtain the most accurate value of m and c.
We will use the Mean Squared Error function to calculate the loss. There are three steps in this function:

- Find the difference between the actual y and predicted y value(y = mx + c), for a given x.
- Square this difference.
- Find the mean of the squares for every value in X.

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2$$

Here y_i is the actual value and $\bar{y}_i$ is the predicted value. Lets substitute the value of $\bar{y}_i$:-

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - (mx_i + c))^2$$

So we square the error and find the mean. hence the name Mean Squared Error. Now that we have defined the loss function, lets get into the interesting part — minimizing it and finding m and c.

Now the Algorithm part :-

Initially let m = 0 and c = 0. Let L be our learning rate. This controls how much the value of m changes with each step. L could be a small value like 0.0001 for good accuracy.

Calculate the partial derivative of the loss function with respect to m, and plug in the current values of x, y, m and c in it to obtain the derivative value D.

$$D_m = \frac{1}{n}\sum_{i=0}^{n}2(y_i - (mx_i + c))(-x_i)$$

$$D_m = \frac{-2}{n}\sum_{i=0}^{n}x_i(y_i - \bar{y}_i)$$

D_m is the value of the partial derivative with respect to m. Similarly lets find the partial derivative with respect to c, Dc :

$$D_c = \frac{-2}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)$$

3. Now we update the current value of m and c using the following equation:

$$m = m - L \times D_m$$

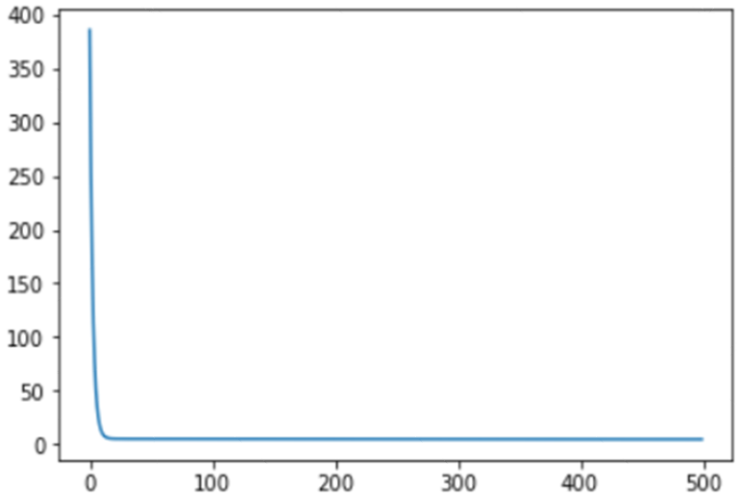
$$c = c - L \times D_c$$

So now we are giving the actual practical part on 500 iteration with the learning rate as 0.007

In [223]:
```
#m_disc, b_disc = np.random.uniform(low=-5, high=5, size=2)
costs = []

m,c=np.random.uniform(low=-5, high=5, size=2)

for i in range(n_iters):
    Yhat = m*X + c  # The current predicted value of Y

    cost = np.mean((Y - Yhat)**2)
    costs.append(cost)

    D_m = (-2/n_points) * sum(X * (Y - Yhat))  # Derivative wrt m
    D_c = (-2/n_points) * sum(Y - Yhat)  # Derivative wrt c
    m = m - learning_rate * D_m  # Update m
    c = c - learning_rate * D_c  # Update c

print (m, c)
```

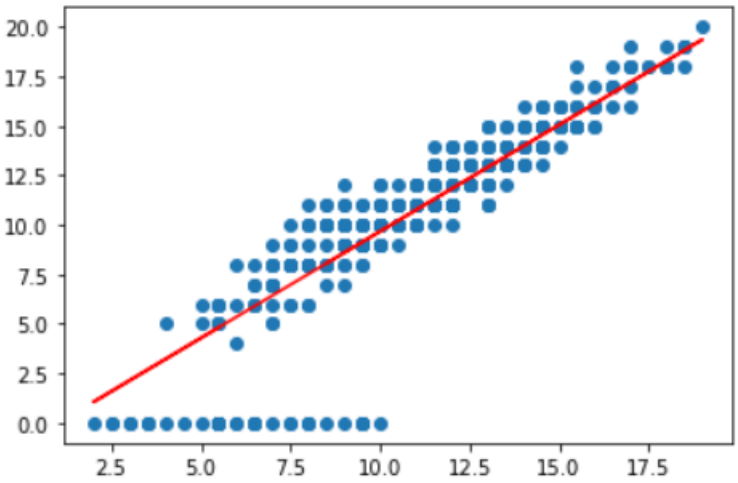
```
1.0763603190375097 -1.0942226805601525
```

```
In [224]: plt.plot(costs)
Out[224]: [<matplotlib.lines.Line2D at 0x2cc6531f3d0>]
```



So this is the graph of cost function or loss function which we have to do minimum by increasing the iteration

```
In [227]: #Yhat=m_disc*X + b_disc
          Yhat=m*X+c
          plt.scatter(X,Y)

          plt.plot(X,Yhat,color='red')
Out[227]: [<matplotlib.lines.Line2D at 0x2cc65382250>]
```



So this is the best fitting of the line picture to our datsets.

NEWTON'S METHOD SOLUTION:-

So in the newtons methos solution what we are doing that everything is same only the equation tom predict Y is different than the gradient solution which is given below as the implementation of NEWTON's method:-

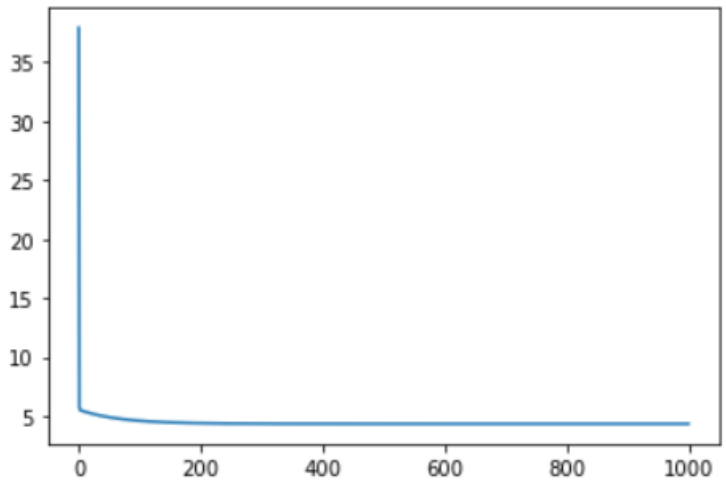
ss

In [244]:
```
n_iters=1000
m,c=np.random.uniform(low=-5, high=5, size=2)
costs=[]
for i in range(n_iters):
    Yhat = m*X + c  # The current predicted value of Y

    cost = np.mean((Y - Yhat)**2)
    costs.append(cost)

    D_m = (-2/n_points) * sum(X * (Y - Yhat))  # Derivative wrt m
    D_c = (-2/n_points) * sum(Y - Yhat)  # Derivative wrt c

    D2_m= (2/n_points)*sum(X**2)      #double derivative wrt to m
    D2_c=(2/n_points)*sum(X)          # double derivative wrt to c


    m = m -  (D_m/D2_m)  # Update m
    c = c - ( D_c/D2_c)  # Update c

print (m, c)
```

```
1.1962100421183126 -2.5173711141903787
```

In [245]:
```
plt.plot(costs)
```

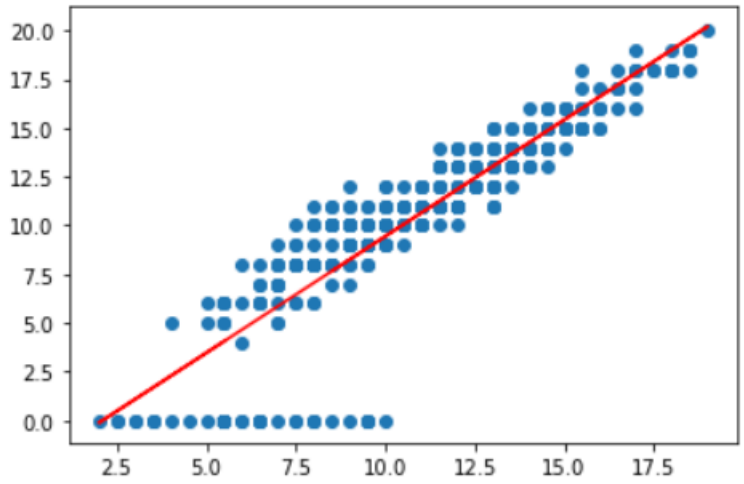
Out[245]:
```
[<matplotlib.lines.Line2D at 0x2cc65249640>]
```



so the above given figure is he observation of plot of costs function which is how decreasing at each iteration and here I have taken 1000 iteration .Now I am giving the obseravtion of fitting of line by NEWTON's method which is attached below.

```
In [246]: Yhat=m*X+c
          plt.scatter(X,Y)

          plt.plot(X,Yhat,color='red')
```

```
Out[246]: [<matplotlib.lines.Line2D at 0x2cc653abc40>]
```



MULTIVARIATE LINEAR REGRESSION:-

So for Multivariate Linear regression we are first selecting the various multiple features which I a doing on the basis of highly corelated data which we are calculated as above too.

- So I am dropping the some very low correlated data like age ,sex ,schol,address,failure ,activites,nursery,freetime,goout,health on which basically the final grade of math will not depend
- So now I am taking basically the 5 features into account which we will denote as X which is called as feature vectors which is (MEDU ,FEDU, STUDYTIME, G1, G2)

For the linear regression, we follow these notations for the same formula:

If we have multiple independent variables, the formula for linear regression will look like:

$$h = \theta_0 + \theta_1 X$$

Here, 'h' is called the hypothesis. This is the predicted output variable. Theta0 is the bias term and all the other theta values are coefficients. They are initiated randomly in the beginning, then optimized with the algorithm so that this formula can predict the dependent variable closely.

$$h = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3 + \cdots$$

Cost Function:-

$$J(\theta_0, \theta_1, \theta_2, ..) = \frac{1}{2m} \sum (h_i - y_i)^2$$

- This is called the cost function. If you notice, it deducts y(the original output) from the hypothesis(the predicted output), takes the square to omit the negativity, sum up and divide by 2 times m. Here, m is the number of training data. You probably can see that cost function is the indication of the difference between the original output and the predicted output. The idea of a machine learning algorithm is to minimize the cost function so that the difference between the original output and the predicted output is closer. To achieve just that, we need to optimize the theta values.

- Here is how we update the theta values. We take the partial differential of the cost function with respect to each theta value and deduct that value from the existing theta value,

$$\theta_0 = \theta_0 - \alpha \frac{d}{d\theta_0} J(\theta_0)$$

$$\theta_1 = \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

Here, alpha is the learning rate and it is a constant. I am not showing the same formula for all the theta values. But It is the same formula for all the theta values. After the differentiation, the formula comes out to be:

so now I am giving the coding approach of this multivariate Linear Regression technique where I have minimize the cost or loss function broadly through the linear regression technique:-

In [1006]:
```
def hypothesis(theta, X):
    return theta*X
```

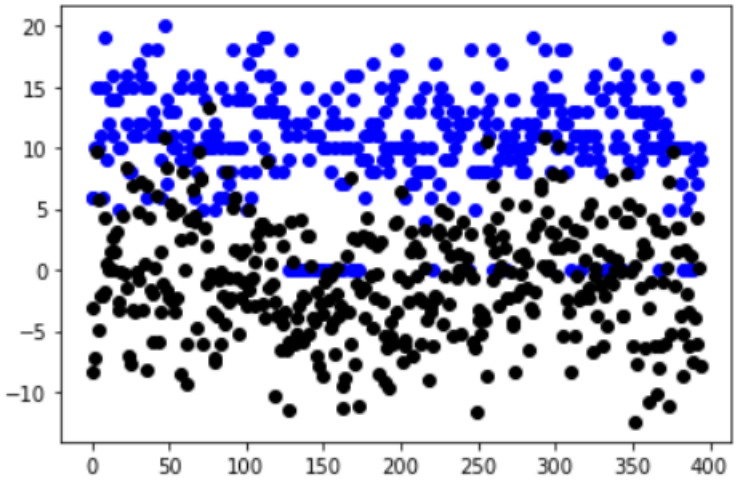
In [1007]:
```
def computeCost(X, y, theta):      # DEFINING THE COST FUNCTION
    y1 = hypothesis(theta, X)
    y1=np.sum(y1, axis=1)
    return sum(np.sqrt((y1-y)**2))/(2*395)
```

In [1018]:
```
def gradientDescent(X, y, theta, alpha, i):
    float(alpha)
    J = []  #cost function in each iterations
    k = 0
    while k < i:
        y1 = hypothesis(theta, X)
        y1 = np.sum(y1, axis=1)
        for c in range(0, len(X.columns)):
            s=sum((y1-y)*X.iloc[:,c]/len(X))

            theta[c] = theta[c] - alpha*s
        j = computeCost(X, y, theta)
        J.append(j)
        k += 1
    return J, j, theta
```

In [1016]:
```
#c=0.01
#d=float(c)


J, j, theta = gradientDescent(X, y, theta,0.0001, 10000)
```

Now I am Plotting the original y and the predicted output y_hat which is here:-

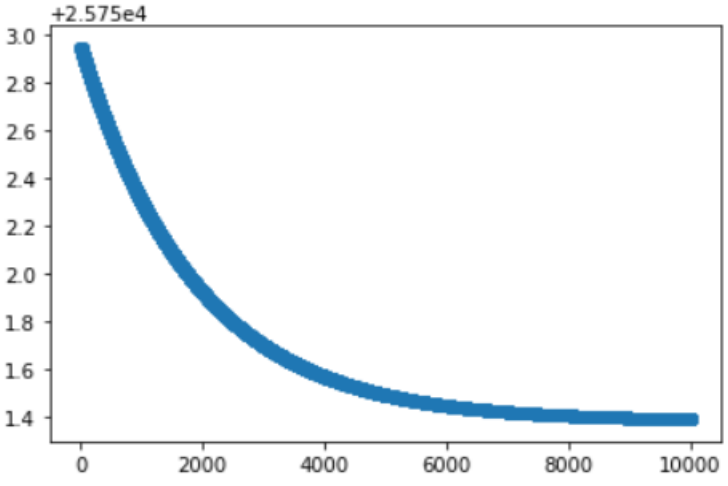
```
In [907]: %matplotlib inline
          import matplotlib.pyplot as plt
          plt.figure()
          plt.scatter(x=list(range(0, 395)), y=y , color='blue')
          plt.scatter(x=list(range(0, 395)), y=y_hat, color='black')
          plt.show()
```



Some of the output points are almost overlapping with the predicted outputs. Some are close but not overlapping.

So now I am Plotting the cost of each iteration to see the behaviour of decreament of cost or loss function:-

```
In [73]: plt.figure()
         plt.scatter(x=list(range(0, 10000)), y=J)
         plt.show()
```



CLASSIFICATION TASK ON CENSUS INCOME DATASET

CLASSIFICATION TASK - Predict whether income exceeds $50K/yr based on Census Income Dataset

We have the following Datasets in CSV format and their details in TXT format

- Census_income.csv – income of the people of any certain year Dataset
- Census_income_details.txt - Dataset description for income data

PREPROCESSING PHASE:-

- First we are checking and verifying our dataset that its in good format with heading or not by introducing the dataframe in the Pandas library.

- Then we are checking with the NULL or NAN values if it available or not in our datasets if it is available the first we have to tackle that

- So in my datasets there are total of 17899 rows of data of income of people of any certain year are stored in which the no columns are having NAN or NULL values which I am verifying by the isna().sum() function.

- Now what my goal is to just remove that NAN values or replace it with some meaningful values so what I am not doing here because here there are no NULL or NAN values.

- After that further again I am checking that is there any rows of column are filled with “?” values so in this there are many rows of columns are present where there are this “?” values.

- since we have 18000 about datapoints out of which 820 data has no workclass so i will delete the rows in which there is no workclass mentioned.

- For the ocupation column part where there are 823 unknown datapoints we will,also delete that 823 rows because that plays an important role in finding the income of an individiual.

- and for the native country we can drop the whole column because country will not play an important role in deciding the income.

- and i am dropping the column which does not play an important role in deciding the income of an individual.

- So I am picking my feature vector as one of the highly corelated columns with our target vector or which I have to predict

- So I am doing df.corr() for checking the correlation with our target data with all the other columns

- Two types f income are there so >50k is encoded as 1 and <=50k is encoded as 0 because this our target vector which is called as target class labels.

- After that we are finding the duplicate rows in our datasets and if there are duplicate rows are there then I have to certainly drop it which I have there in this so I am dropping all that rows.

- After that all the non numerical columns which is categorical columns I am doing one hot encoding of that columns .

Data division into Test and Train Set:-

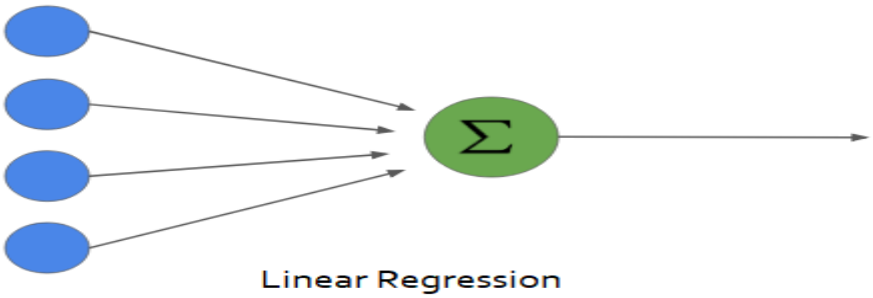
```
In [1760]: ###TRAINING AND TEST SPLITTING THE DATSETS

           #shuffle you dataset


           shuffle_df=df.sample(frac=1)


           #define size of your train set

           train_size=int(0.8*len(df))


           #split your dataset

           train_set = shuffle_df[:train_size]
           test_set=shuffle_df[train_size:]
```

So we are taking 80% of data into training set and 20% of the data into test set which are done above.

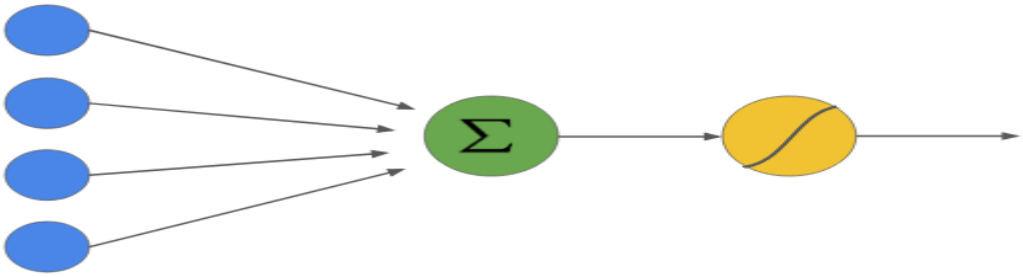
LOGISTIC REGRESSION TECHNIQUE:-

Logistic Regression is generally used for classification purposes. Unlike Linear Regression, the dependent variable can take a limited number of values only i.e, the dependent variable is categorical. When the number of possible outcomes is only two it is called Binary Logistic Regression.

In Linear Regression, the output is the weighted sum of inputs. Logistic Regression is a generalized Linear Regression in the sense that we don't output the weighted sum of inputs directly, but we pass it through a function that can map any real value between 0 and 1.



Linear Regression



Logistic Regression

The activation function that is used is known as the sigmoid function

Hypothesis and Cost Function

Till now we have understood how Logistic Regression can be used to classify the instances into different classes. In this section, we will define the hypothesis and the cost function.

A Linear Regression model can be represented by the equation.

$$h(x) = \theta^T x$$

We then apply the sigmoid function to the output of the linear regression

$$h(x) = \sigma(\theta^T x)$$

where the sigmoid function is represented by,

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

The hypothesis for logistic regression then becomes,

$$h(x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$h(x) = \begin{cases} > 0.5, & \text{if } \theta^T x > 0 \\ < 0.5, & \text{if } \theta^T x < 0 \end{cases}$$

If the weighted sum of inputs is greater than zero, the predicted class is 1 and vice-versa. So the decision boundary separating both the classes can be found by setting the weighted sum of inputs to 0.

Cost Function

Like Linear Regression, we will define a cost function for our model and the objective will be to minimize the cost.

The cost function for a single training example can be given by:

$$cost = \begin{cases} -log(h(x), & \text{if } y = 1 \\ -log(1 - h(x)), & \text{if } y = 0 \end{cases}$$

We can combine both of the equations using:

$$cost(h(x), y) = -ylog(h(x)) - (1 - y)log(1 - h(x))$$

The cost for all the training examples denoted by J(θ) can be computed by taking the average over the cost of all the training samples

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h(x^i)) + (1-y^i)log(1-h(x^i))]$$

where m is the number of training samples.

We will use gradient descent to minimize the cost function. The gradient w.r.t any parameter can be given by

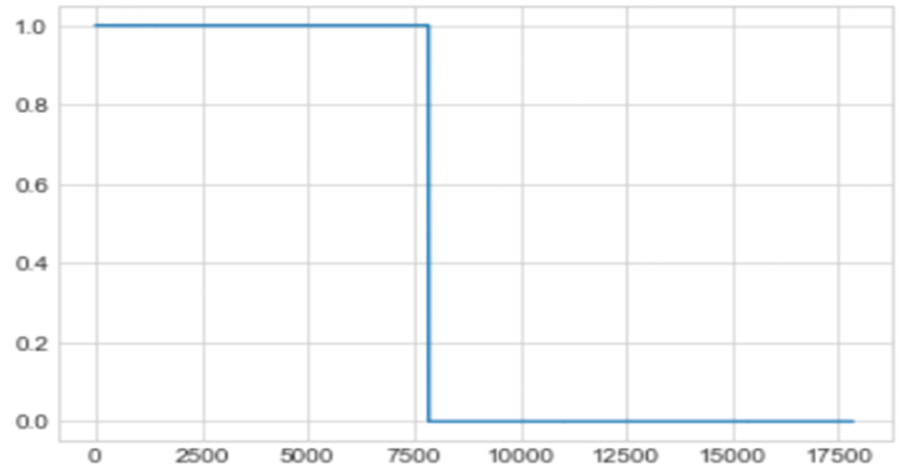
$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h(x^i) - y^i)\ x_j^i$$

So the implementation part is here:-

```
[1847]: class LogitRegression() :
            def __init__( self, learning_rate, iterations ) :
                self.learning_rate = learning_rate
                self.iterations = iterations
            # Function for model training
            def fit( self, X, Y ) :
                # no_of_training_examples, no_of_features
                self.m, self.n = X.shape
                # weight initialization
                self.W = np.zeros( self.n )
                self.b = 0
                self.X = X
                self.Y = Y
                # gradient descent learning
                for i in range( self.iterations ) :
                    self.update_weights()
                return self
            # Helper function to update weights in gradient descent
            def update_weights( self ) :
                A = 1 / ( 1 + np.exp( - ( self.X.dot( self.W ) + self.b ) ) )
        #         print("after fitting x and y shape",self.X.shape,self.Y.shape)
        # calculate gradients
                tmp = ( A - self.Y.T )
                tmp = tmp.to_numpy()
                tmp = np.reshape( tmp, self.m )

                dW = np.dot( self.X.T, tmp ) / self.m
                db = np.sum( tmp ) / self.m

                # update weights
                self.W = self.W - self.learning_rate * dW
                self.b = self.b - self.learning_rate * db
```

```
        return self
    # Hypothetical function  h( x )
    def predict( self, X ) :
        Z = 1 / ( 1 + np.exp( - ( X.dot( self.W ) + self.b ) ) )
        Y = np.where( Z > 0.5, 1, 0 )
        return Y
regressor = LogitRegression(learning_rate = 0.004, iterations = 1000)
print("before fitting x and y shape",X_train.shape,Y_train.shape)
model = regressor.fit(X_train,Y_train)
print("after fitting x and y shape",X_train.shape,Y_train.shape)
plt.plot(Y)
```

```
before fitting x and y shape (10879, 42) (10879, 1)
after fitting x and y shape (10879, 42) (10879, 1)

[<matplotlib.lines.Line2D at 0x1d753c2d250>]
```



So now I am calculating the accuracy on our test set :-

In [1781]:
```
evaluate = 0
count = 0
for count in range( np.size( Y_pred ) ) :

    if Y_test[count] == Y_pred[count] :
        evaluate = evaluate + 1
    count = count + 1

print( "Accuracy on test set by our model :  ", (evaluate / count ) * 100 )
```

```
Accuracy on test set by our model :   61.06617647058824
```

NAÏVE BAYES METHOD FOR CLASSIFICATION TASK:-

The Mathematics Behind Naives Bayes:-

The naives bayes classifiers are completely dependent on the Bayes' Theorem (hence the name), since the classifiers simply apply the formula to sets of data. This theorem consists of a formula assessing probabilities of different events occurring. The formula below is the simplest version of it, with only two events — Event A and Event B.

$$P(B|A) = \frac{p(A|B)p(B)}{p(A)}$$

Remember, this is giving you the probability of event B occurring given that A has already occurred. It uses two types of probabilities:

Probability of each event.

Probability for each event given another event value.

When there are more than just two possible events in a data set, the formula will take this form.

$$P(x|c) = P(x_1|c_j) * P(x_2|c_j) * \ldots * P(x_d|c_j)$$

$$= \prod_{k=1}^{d} P(x_k \mid c_j)$$

While this formula may initially look confusing, it is simply using a bunch of combinations of these two types of probabilities of events in order to find the likelihood of a certain event occurring.

IMPLEMENTATION OF NAÏVE BAYES ALGORITHM:-

```
class GaussianNaiveBayes:

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        self.means = np.zeros((n_classes, n_features), dtype=np.float64)
        self.var = np.zeros((n_classes, n_features), dtype=np.float64)
        self._priors =  np.zeros(n_classes, dtype=np.float64)

        # calculating the mean, variance and prior P(H) for each class
        for i, c in enumerate(self.classes):
            X_for_class_c = X[y==c]
            self.means[i, :] = X_for_class_c.mean(axis=0)
            self.var[i, :] = X_for_class_c.var(axis=0)
            self._priors[i] = X_for_class_c.shape[0] / float(n_samples)

    # calculating the likelihood, P(E|H)
    def _calculate_likelihood(self, class_idx, x):
        mean = self.means[class_idx]
        var = self.var[class_idx]
        print(x)
        num = np.exp(-(x-mean)**2 / (2 * var))
        denom = np.sqrt(2 * np.pi * var)
        return num / denom
```

```
    def predict(self, X):
            y_pred = [self._classify_sample(x) for x in X]
            return np.array(y_pred)

    def _classify_sample(self, x):
        posteriors = []

        # calculating posterior probability for each class
        for i, c in enumerate(self.classes):
            prior = np.log(self._priors[i])
            posterior = np.sum(np.log(self._calculate_likelihood(i, x)))
            posterior = prior + posterior
            posteriors.append(posterior)

        # return the class with highest posterior probability
        return self.classes[np.argmax(posteriors)]
```

```
]: gb = GaussianNaiveBayes()
   gb.fit(X_train, Y_train)
   predictions = gb.predict(X_test.values)
```

```
  0  0  0  1  0  0  0  0  0  0  0  0  0  1  0  0  0  0]
[21  6  0  0 50  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  1  0  0  0  0  0  0  0  0  0  1  0  0  0  0]
[51 10  0  0 15  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0
```

So now I am comparing the accuracy of my NAÏVE BAYES by using with scikit learn and without scikit learn:-

```
[1841]:
from sklearn.metrics import accuracy_score

accuracy_score(Y_test, predictions)
```

```
[1841]: 0.5095588235294117
```

```
[1843]: gb = GaussianNB()
        gb.fit(X_train,Y_train)
        pred = gb.predict(X_test)
```

```
[1844]: accuracy_score(pred, Y_test, normalize = True)
```

```
[1844]: 0.6996323529411764
```