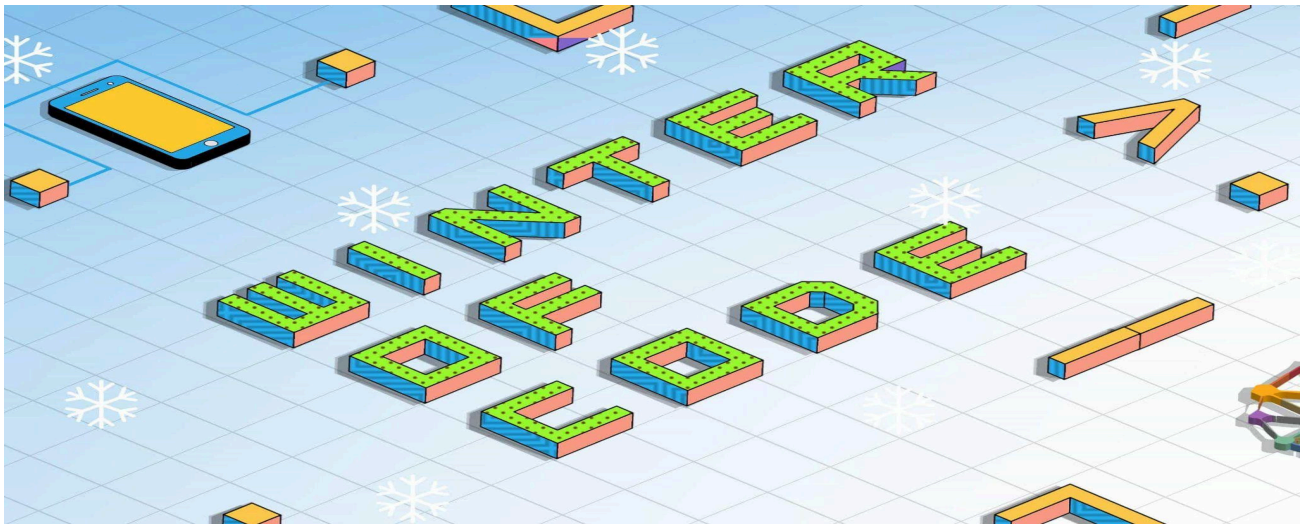


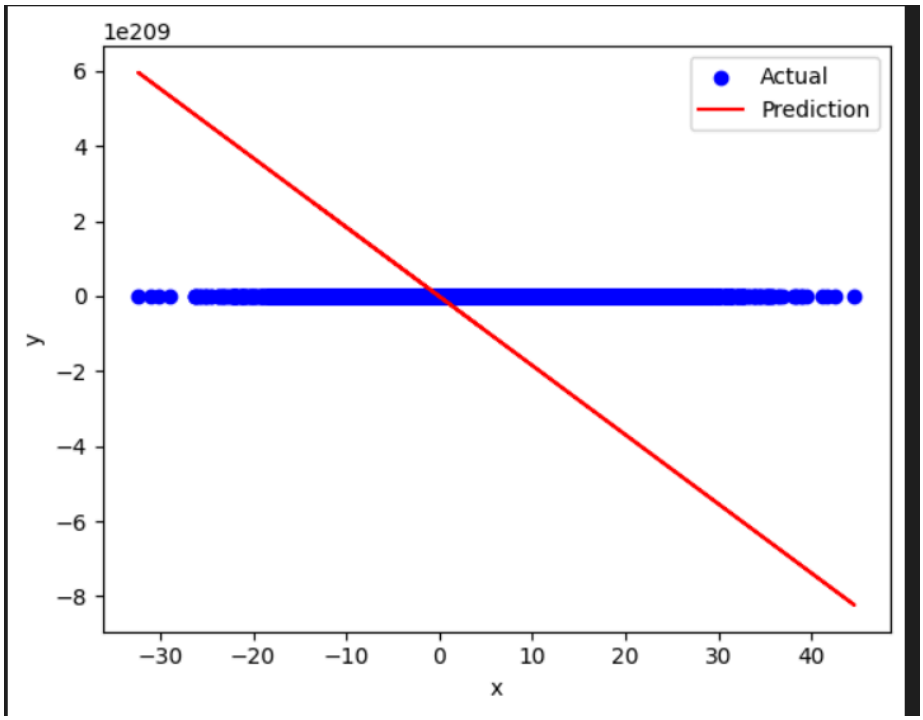
Objective:-

The main objective was to develop a machine learning library by implementing the following algorithms from scratch: Linear Regression (and Polynomial Regression), Logistic Regression, K-Nearest Neighbors (KNN), K-Means Clustering, Decision Trees, and N-Layer Neural Network without using any other libraries or tech stacks like SK learn, etc.
I enjoyed doing this project while learning a lot of new stuff. I am a **beginner code**r with **no previous experience in either machine learning or Python.**
I had been hospitalised for a surgery in my hometown during the Winter of Code and I was in bed rest until the 19th of December, so yes my algorithms are sloppy and not that good
Note :- I was unaware of Object Oriented Programming (OOPs) for the first few algorithms hence I have used them only in the last few of my projects.

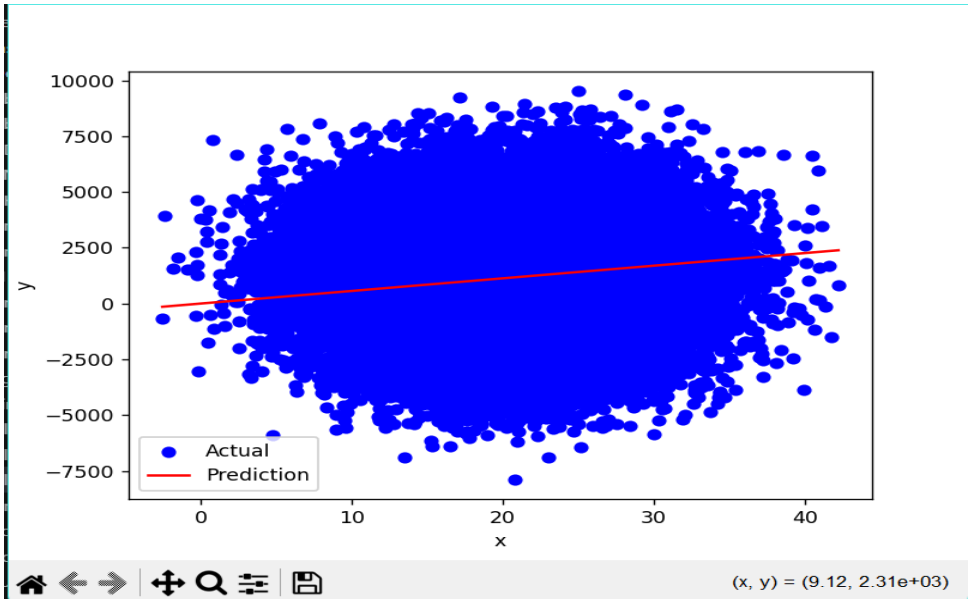
Linear Regression:-

The primary objective of this project was to develop a Linear Regression model from scratch without relying on high-level machine learning libraries like scikit-learn or TensorFlow. Firstly I got a clear idea of mathematics involved in Linear Regression and how I could use gradient descent to find the optimal line passing through the maximum of data points. The first few tries I

made were giving huge errors with the line not even matching the data!



After improving the Learning Rate and fixing the mistake I made in finding the Gradient Descent I came up with this for a couple of features given to us in the training and testing data.



I found that decreasing the learning rate to the neighbouring numbers of 0.0001 gave me the best results for the data involved.

To find the error and accuracy I made an algorithm to consider Multiple variables at once to find Multiple Variable Linear Regression Error and Accuracy using the MSE (Mean Square Error) and fed it with the training and testing data to further find the optimal accuracy and errors.

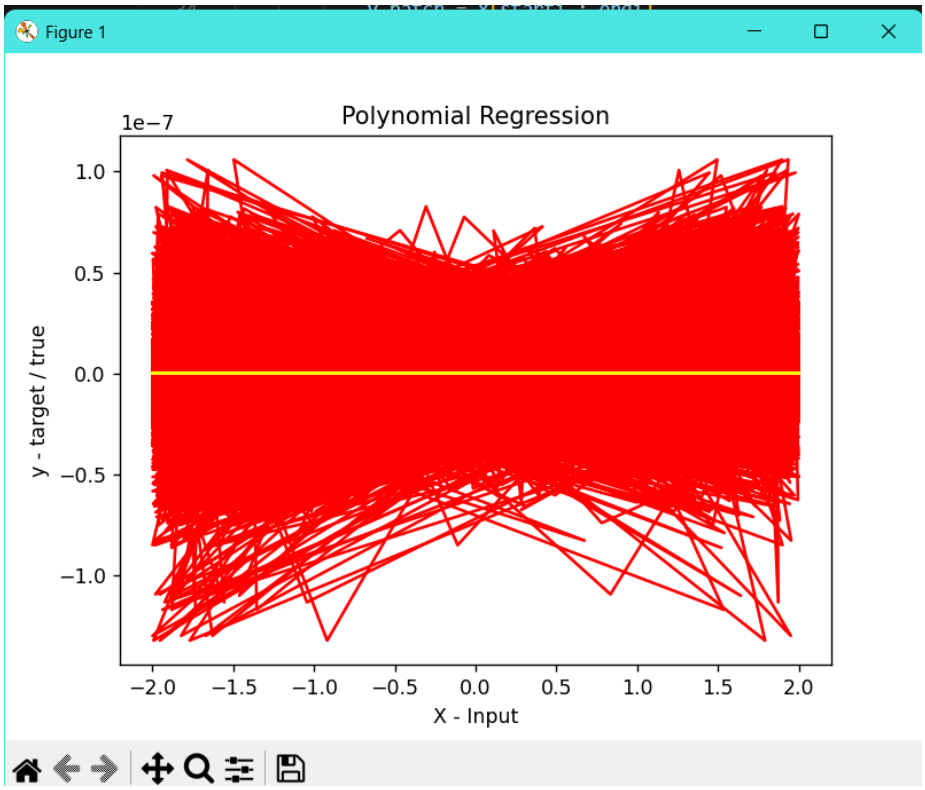
```
ear Regression/linearregression.py"
Slope: 56.55726661327288, Intercept: 3.469396409350194, Error: 99.0856734902349
```

```
ear Regression/mvlr.py"
Error:0.9587434123021001
Accuracy:[99.98052207 99.91728261 99.86060227 ... 99.92880692 99.97472564
 99.96937395], % Error :[0.01947793 0.08271739 0.13939773 ... 0.07119308 0.02527436 0.03062605]
```

After multiple fixes with the learning rate and Iterations, I came close to an **Accuracy of Around 99.8%**

Polynomial Regression:-

In polynomial regression, I had to demonstrate the steps required to extend linear regression to model non-linear relationships between features and the target variable. I used the gradient descent function in the algorithm. We define the polynomial regression function with the X, Y and the degree function. I made many mistakes with overfitting as the degrees increase can easily cause overfitting in Polynomial which led to data like this :



Hence after changing code, playing with iterations, I found that degree = 4 suited the data best and gave the best results with a R2 Score of 0.92!

```
PS C:\Users\chhru\Desktop\WOC> & C:/Users/chhru/AppData/Local/Programs/Python/Python313/python.exe c:/Users/chhru/Desktop/WOC/SupervisedLearning/Regressions/Polynomial/polytest.py
R2 Score: 0.9257635753271621
```

```python
def r2_score(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return 1 - (ss_residual / ss_total)
```

Note :- explanation of r2 score and formulae learn and taken from Neural Nine Youtube Channel

```python
theta = np.linalg.inv(X_poly.T @ X_poly) @ (X_poly.T @ Y) #Mathematical eqn from Saksham Jain's Mathematical Video of Polynomial Regression
```

For Polynomial Regression I learnt the concept and relied on videos from Neural Nine, and Normalised Nerd.

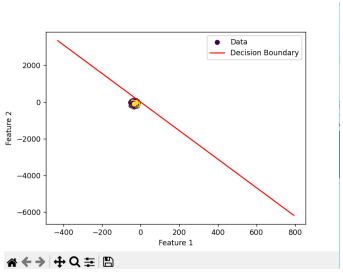
Logistic Regression:-

I made the Logistic Regression algorithm using the Sigmoid Function. Logistic Regression is used for binary classification problems hence I used the provided binary data. I implemented the Gradient Descent in the algorithm to minimize the cost function while the sigmoid function maps the predictions between 0 and 1. The cost function of Logistic Regression can be mentioned as

$$J(w,b) = J(\theta) = \frac{1}{N}\sum_{i=1}^{n}[y^i log(h_\theta(x^i)) + (1-y^i) log(1-h_\theta(x^i))]$$

Source: AssemblyAI Youtube Video
I implemented this function and minimised the cost function(error). Similarly we can adjust our weights and bias (m & c or w & b). After setting the learning rate and epochs to my desired rate, and using the average Y to get our accuracy and error, we get the following.

```
PS C:\Users\chhru\Desktop\WOC> & C:/Users/chhru/AppData/Local/Programs/Python/Python313/python.exe c:/Users/chhru/Desktop/WOC/SupervisedLearning/Regressions/Logistic/logistic.py
Weights : [ 0.00276396  0.00035431 -0.00104947 -0.00673049 -0.00259944  0.01944073
  0.0017815   0.00317422  0.00450266  0.000402   -0.00109997  0.00023755
 -0.00102498  0.00212707  0.00208644 -0.01133115  0.01265955 -0.00920772
  0.00113691  0.00662878], Bias : -8.36209245022882e-05
Predictions: [0 1 0 ... 0 0 1]
Loss : 0.24735985801020513
Accuracy = 93.46875%
```



K Nearest Neighbours:-

K- Nearest Neighbours algorithm is a powerful classification algorithm which has the potential to classify a new data point in several already existent classes in a data set, this works on the concept of ‘distance’ of the new data point from it’s K nearest neighbours (where K can take any values preferably in the range of 0 to 5) and assigning it the label which is most prevalent in all it’s K neighbours.

For it’s implementation I employed the use of Euclidean Distance (which is the most prominent formulation of ‘distance’ other than manhattan and Minkowiski), which sequentially finds the distance of test-point from all the data points, stores, sorts these values and clip the first K values, then take their means and assign that particular test point a class based on the mean of these K values

SupervisedLearning > Classifications > KNN > KNN.py > ...

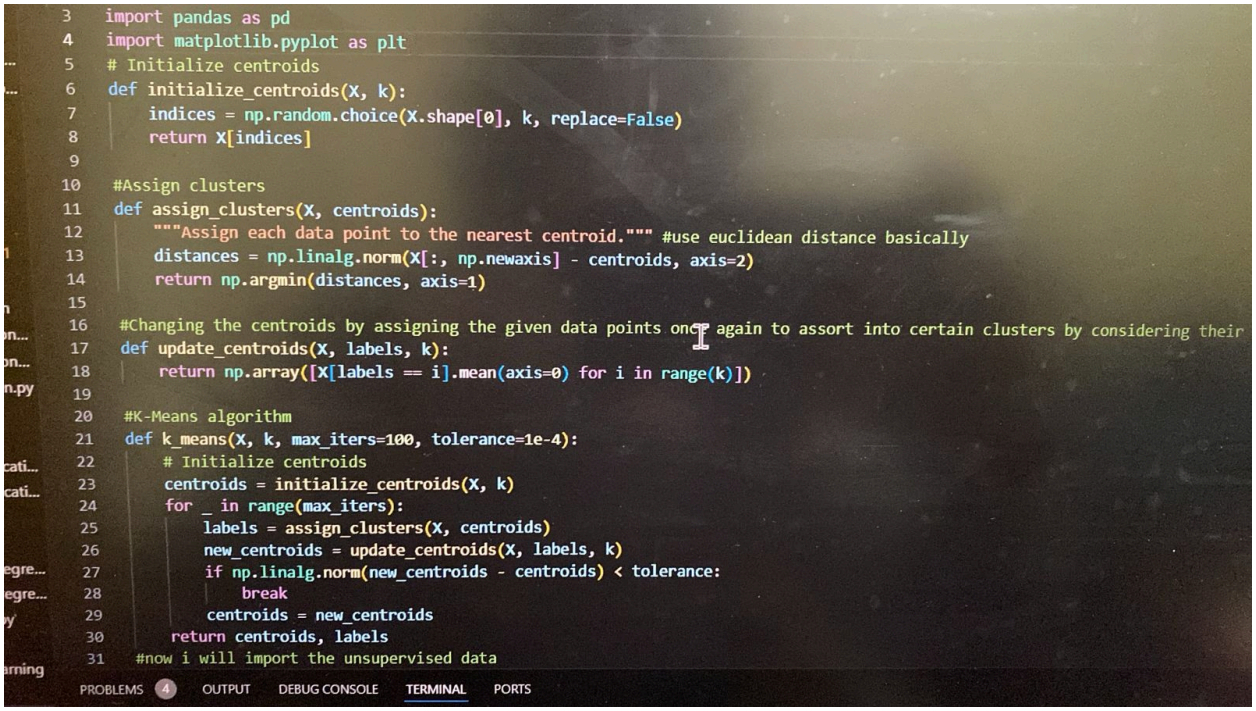
```python
#importing our tech stack required
import numpy as np
import pandas as pd
#I will Define the KNN Function here
def knn_classify(X, Y, X_test, k=3):
    pred = []
    for testpoint in X_test:
        #Computing Distances using Euclidian Distance(Most used in KNN other than Manhattan and Minkowiski)
        dist = np.sqrt(np.sum((X - testpoint)**2 , axis = 1))
        #Now we neeed to find indices of K-Nearest neighbours in this case 3 (odd no so that we don't get any ambigui
        k_indices = np.argsort(dist)[:k]
        k_value = [Y[i] for i in k_indices]
        #Now we have to consider the majority in the nearest neighbors to come to a decision
        pred.append(np.mean(k_value))
    return pred
#Now I iwll import the train and test datas using Pandas
data = pd.read_csv("SupervisedLearning\\Classifications\\KNN\\multi_classification_train.csv")
data1 = pd.read_csv("SupervisedLearning\\Classifications\\KNN\\multi_classification_test.csv")
x = data.iloc[:,1:21]
y = data['Class']
x_test = data1.iloc[:,1:21]
X = np.array(x)
Y = np.array(y)
X_test = np.array(x_test)
pred = knn_classify(X, Y, X_test, k=3)
print(pred)
#Calculating the F1 score of the model using precision and recall
def f1_score_from_scratch(y, y_pred):
```

```
(1.0), np.float64(2.0), np.float64(4.0), np.float64(2.0), np.float64(1.0), np.float64(3.0), np.float64(1.0), np.float64(2.0), np.float64(3.0), np.
float64(2.0), np.float64(1.0), np.float64(2.0), np.float64(3.0), np.float64(1.0), np.float64(2.0), np.float64(0.3333333333333333), np.float64(1.
0), np.float64(2.0), np.float64(3.0), np.float64(3.0), np.float64(3.0), np.float64(1.0), np.float64(2.0), np.float64(3.0), np.float64(2.0), np.
float64(2.0), np.float64(1.0), np.float64(1.0), np.float64(2.0), np.float64(3.0), np.float64(2.0), np.float64(2.0), np.float64(3.0), np.float64(1.
0), np.float64(1.0), np.float64(3.0), np.float64(2.0), np.float64(1.0), np.float64(4.0), np.float64(2.0), np.float64(1.0), np.float64(1.0), np.
float64(1.0), np.float64(2.0), np.float64(4.0), np.float64(3.0), np.float64(1.0), np.float64(3.0), np.float64(1.3333333333333333), np.float64(3.
0), np.float64(2.0), np.float64(1.6666666666666667), np.float64(3.0), np.float64(1.0), np.float64(2.0), np.float64(2.6666666666666665), np.float64
(3.0), np.float64(2.6666666666666665), np.float64(1.6666666666666667), np.float64(3.0), np.float64(1.0), np.float64(2.0), np.float64(0.0), np.
float64(4.0), np.float64(1.0), np.float64(2.0), np.float64(3.0), np.float64(1.0), np.float64(3.0), np.float64(3.0), np.float64(0.0), np.float64(2.
0), np.float64(1.3333333333333333), np.float64(2.0), np.float64(1.0), np.float64(4.0), np.float64(2.0), np.float64(3.3333333333333335), np.float64
(1.0), np.float64(3.6666666666666665), np.float64(3.0), np.float64(2.0), np.float64(2.0), np.float64(3.3333333333333335), np.float64(1.0), np.
float64(2.0), np.float64(3.0), np.float64(2.0), np.float64(2.0), np.float64(3.0), np.float64(0.0), np.float64(2.0), np.float64(1.0), np.float64(1.
6666666666666667), np.float64(4.0), np.float64(2.0), np.float64(2.0), np.float64(3.3333333333333335), np.float64(2.0), np.float64(2.0), np.float64
(1.0), np.float64(2.0), np.float64(3.0), np.float64(2.0), np.float64(1.0), np.float64(2.0), np.float64(3.0), np.float64(0.6666666666666666), np.
float64(3.0), np.float64(3.0), np.float64(4.0), np.float64(4.0), np.float64(1.0), np.float64(2.0), np.float64(2.0), np.float64(2.0), np.float64(2.
```

In this snippet the ‘knn_classify()’ does all this processing and returns the potential class of the datapoint. [Result Data Image Inserted Above.]

K Means Clustering:-

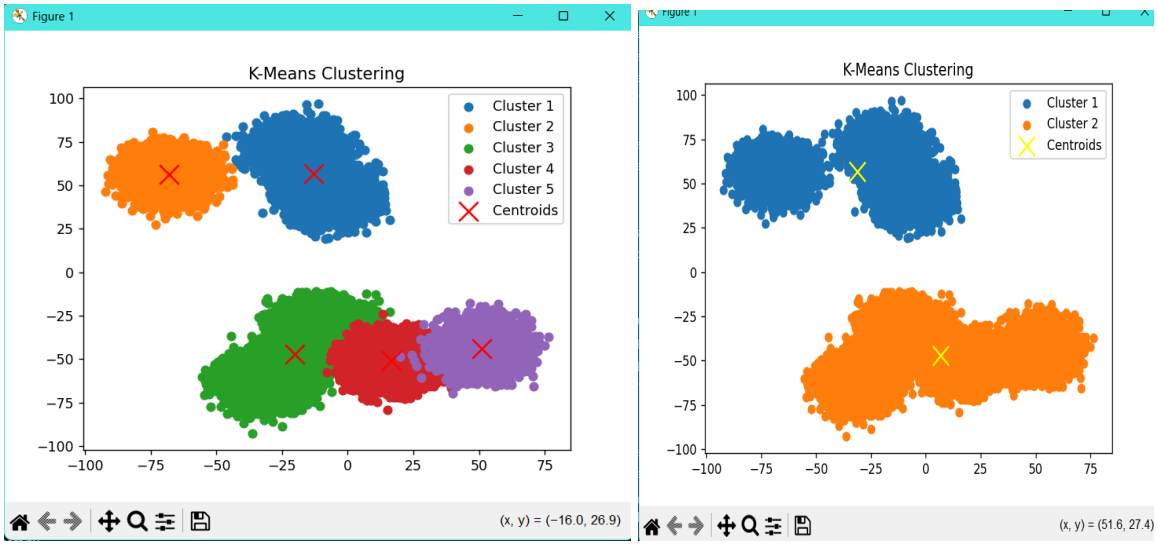
K Means Clustering is an unsupervised learning algorithm used for bundling data into smaller localized ‘clusters’ identified by apparent similarity in features of data presented. This algorithm works by finding centroids of a category of data and then adding new data points in that cluster and continuously updating the centroid for all the classes of clusters.



```
import pandas as pd
import matplotlib.pyplot as plt
# Initialize centroids
def initialize_centroids(X, k):
    indices = np.random.choice(X.shape[0], k, replace=False)
    return X[indices]

#Assign clusters
def assign_clusters(X, centroids):
    """Assign each data point to the nearest centroid.""" #use euclidean distance basically
    distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
    return np.argmin(distances, axis=1)

#Changing the centroids by assigning the given data points once again to assort into certain clusters by considering their
def update_centroids(X, labels, k):
    return np.array([X[labels == i].mean(axis=0) for i in range(k)])

#K-Means algorithm
def k_means(X, k, max_iters=100, tolerance=1e-4):
    # Initialize centroids
    centroids = initialize_centroids(X, k)
    for _ in range(max_iters):
        labels = assign_clusters(X, centroids)
        new_centroids = update_centroids(X, labels, k)
        if np.linalg.norm(new_centroids - centroids) < tolerance:
            break
        centroids = new_centroids
    return centroids, labels
#now i will import the unsupervised data
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS

This code snippet begins by randomly choosing ‘K’ points from the existing feeded data set and labelling them as the centroids, furthermore it finds the euclideans distance of each and every point from all the centroids and assigns them to any of the K clusters based on the magnitude of the distance then update the centroids. This process is repeated until all the data is not assigned under any of the K clusters.



Since K Means Clustering is an unsupervised algorithm we don't have any target variable or accuracy to aim for, hence no predefined metric can be used to judge it's accuracy rather we can use data analytics before the model to identify the classes using statistics and later confirm the presence of all those classes.

Decision Trees:-

Decision Trees help us classify random data points, in which classes cannot be easily linearly separable. It recursively splits the data until we get pure leaf nodes (only contain one type of class). I start at the root node and we initially define the Node function and further move on to classify the tree.

```
class DecisionTree():#this class will be the most important for building the decision tree
    def __init__(self, min_split=2, max_depth=2):
        self.min_split = min_split
        self.max_depth= max_depth
        self.root = None
    def buildtree(self, dataset, curr_depth=0):
        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)
        best_split = {}
        if num_samples>= self.min_split and curr_depth<=self.max_depth:
            best_split = self.get_best_split(dataset, num_samples, num_features)
            if best_split["info_gain"]>0: #info gain will be given later
                left_subtree = self.buildtree(best_split["dataset_left"], curr_depth+1)
                right_subtree = self.buildtree(best_split["dataset_right"], curr_depth+1)
                return Node(best_split["feature_index"], best_split["threshold"],left_subtree, right_subtree, best_split["info_gain"])
        leaf_value = self.calculate_leaf_value(Y)
        return Node(value = leaf_value)
```

I then used the best_split function, here we find the information gain using the gini function instead of entropy(i could have used entropy but gini takes less computing power). We split the data in the permutation where we have the highest information gain. Using the following formula,

$$\text{IG} = E(parent) - \sum w_i \; E(child_i)$$

Source: Normalised Nerd On Youtube
Which I depicted as,

```
if mode=="gini":
    gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child) + weight_r*self.gini_index(r_child))
else:
    gain = self.entropy(parent) - (weight_l*self.entropy(l_child) + weight_r*self.entropy(r_child))
```

We loop this and go on until we find the leaf node of the following binary classification data we used. Then we assess our accuracy using the requisite F1 score.

Neural Network:- [Learnt from Samson Zhang and SentDex]

Unfortunately, I didn't have much time to train my neural network algorithm When I build a neural network from scratch, I start by deciding its architecture(how many layers it has, how many neurons are in each layer, and which activation functions to use). I begin by initializing the weights and biases with random values for different layers (hidden and output layer). Then, during forward propagation, I take the inputs and pass them through the layers.

```
def initialize_parameters(input_size, hidden_size, output_size):
    np.random.seed(42)# this is just so that I can use them later
    #w1,b1 are for hidden layer and w2,b2 are for output layers
    W1 = np.random.randn(input_size, hidden_size) * 0.01
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, output_size) * 0.01
    b2 = np.zeros((1, output_size))
    return W1, b1, W2, b2

#forward propagation
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = np.maximum(0, Z1) # this is relu propagation
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    return Z1, A1, Z2, A2
```

We consecutively do the backward propagation using the relu function and then find its derivative. We then go on to find the f1 score using precision and recall using the formula given. Using gradient descent, I update the weights and biases step by step. I repeat this process of forward propagation, calculating the loss, backpropagation, and updating parameters over several iterations (epochs) until the network gets better at making accurate predictions.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Now finally once trained I applied my data from the neural network given to find my F1 Score and predictions.