

Nearest Neighbor

```
KNN_test(X_train, Y_train, X_test, Y_test, K)
```

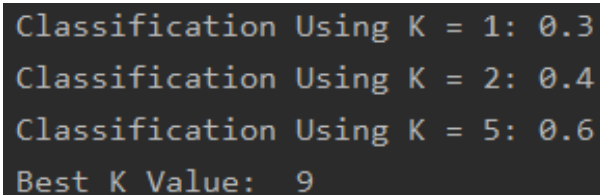
To implement this function we had helper functions to compute the distance from the test point to each training points and to predict the label for each Test Data. Once we created those helper functions, we used our helper function to create a numpy array of the predictions for each test point. To find the accuracy we compared the real labels with our predictions. We then counted the amount of times that it was correct and divided that by the total amount of test points.

```
choose_K(X_train, Y_train, X_val, Y_val)
```

To choose the best K-value we sampled the accuracy through each odd k-value from 1 to the amount of training points. While iterating through we checked that if the old k-value is better than the new k-value, then update it. From there we returned the k-value which had the best accuracy.

Testing Nearest Neighbor Functions

Below you can see the results of the tests using the data provided. It has the classification using different K values, and it also shows the best K value for the data.



```
Classification Using K = 1: 0.3  
Classification Using K = 2: 0.4  
Classification Using K = 5: 0.6  
Best K Value: 9
```

Figure 1: Nearest Neighbor Function Tests Results

Clustering

K_Means(X,K)

To implement this function we had helper functions to compute the distance from the test point to each training points and to update the cluster centers by finding the average of the distance of the points in each cluster. To actually perform K-Means, we first randomly chose our initial cluster centers using the random function making sure not to choose the same initial cluster. We then keep updating the cluster centers using our helper function until the cluster centers stop changing. From there we return the cluster centers.

K_Means_better(X,K)

For this function we used the data and created a thousand clusters and put that into an array. After putting it into an array we found all the unique cluster center values, and the count of each of those unique values. We used the unique value counts and the values and return the cluster center value that is most frequent from that array.

Testing Clustering Functions

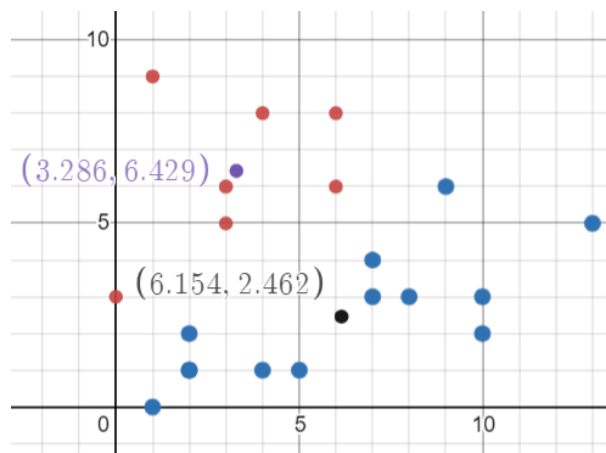


Figure 2: Testing K_Means Where K=2

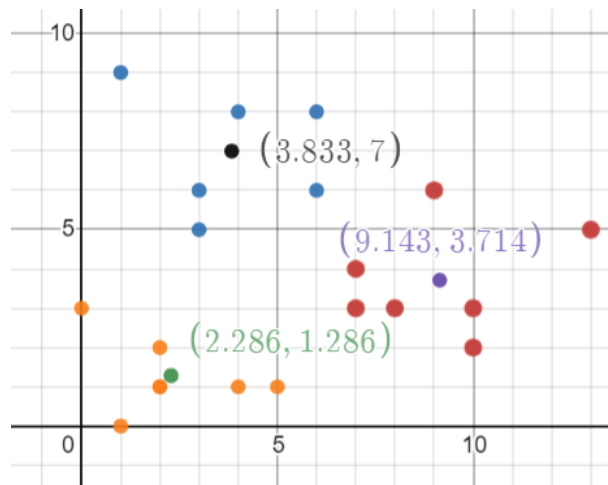


Figure 3: Testing K_Means Where K=3

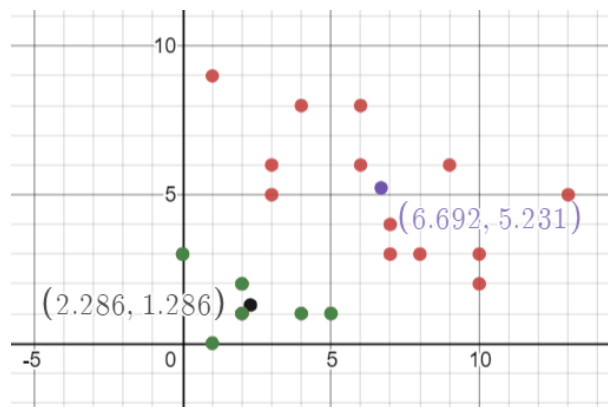


Figure 4: Testing K_Means_Better Where K=2

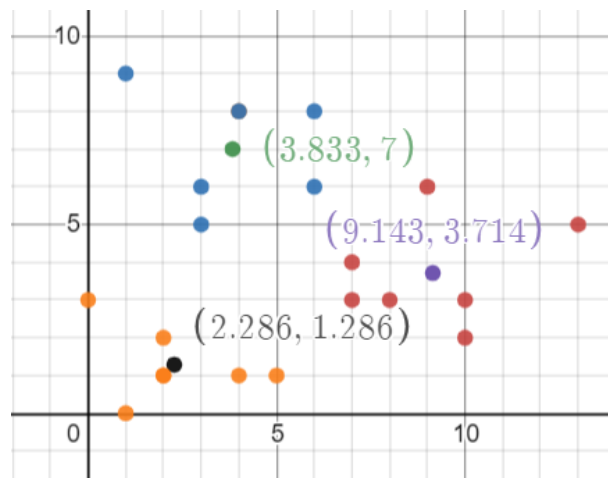


Figure 5: Testing K_Means_Better Where K=3

Perceptron

`perceptron_train(X,Y)`

For this function we had a helper function to update the weights and bias. We started the weights and bias at 0 and go through one epoch at a time until the previous weights and bias is equal to the current weights and bias. We had a boolean to keep track of whether the weights and bias have been updated on the most current epoch. We would go through each sample in the iteration and update it if the activation was less than or equal to zero. The function returned the weights and bias after all the updating.

`perceptron_test(X_test, Y_test, w, b)`

For this function we had a count which kept track of how many labels that were correct. To get the number of correct labels we iterated through all the testing samples and calculated the activation for each sample. If the activation multiplied by the label was positive, then we incremented the count. To get the accuracy we got the count of the correct labels divided by the number of samples.

Training a Perceptron

When training the perceptron, we got weights of 2 and 4 and the bias as 2. The image below shows the plot of our decision boundary using the weights and biases we found from the data provided.

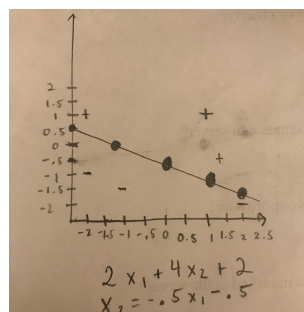


Figure 6: Perceptron Decision Boundary Plot