1 Nearest Neighbor

1.1 Describe Implementation

The general idea is to compute the L2 Norm to the nearest neighbor, K point, and return the most recurring label in those neighbors.

1.1.1 KNN_TEST

This function genrates a label based on the points given to it, computes the distance, to each of those points (training points) and puts them in a list. This list is then sorted based on the distance that was calculated before. The subset of the K nearest neighbor or point will have their labels summed and then we will output the sign of the summation as the label, so if our predicted label is less than 0 we return -1 and if it is greater than 0 we return 1. In case the predicted value is 0 we will choose the result based on the ordering of the training data and from the set of neighbors in the sorted list.

```
def KNN_test(X_train, Y_train, X_test, Y_test,K):
  NumberOfCorrect = 0
  for i, j in enumerate(X_test):
    radius = []
    for n, m in enumerate(X_train):
      radius.append(((math.pow((m[0] - j[0]), 2) + math.pow((m[1] - j[1]), 2)), Y_train[n][0]))
  List_sorted = sorted(radius, key=lambda member: member[0], reverse=False)[:K]
  predictedvalue = sum([member[1] for member in List_sorted])
  if (not predictedvalue):
    predictedvalue = List_sorted[0][1]
  elif (predictedvalue < 0):
    predictedvalue = -1
  elif (predictedvalue > 0):
    predictedvalue = 1
  if predictedvalue == Y_test[i][0]:
    NumberOfCorrect = NumberOfCorrect+1
  return NumberOfCorrect / len(X_test)

```

1.1.2 Choose_K

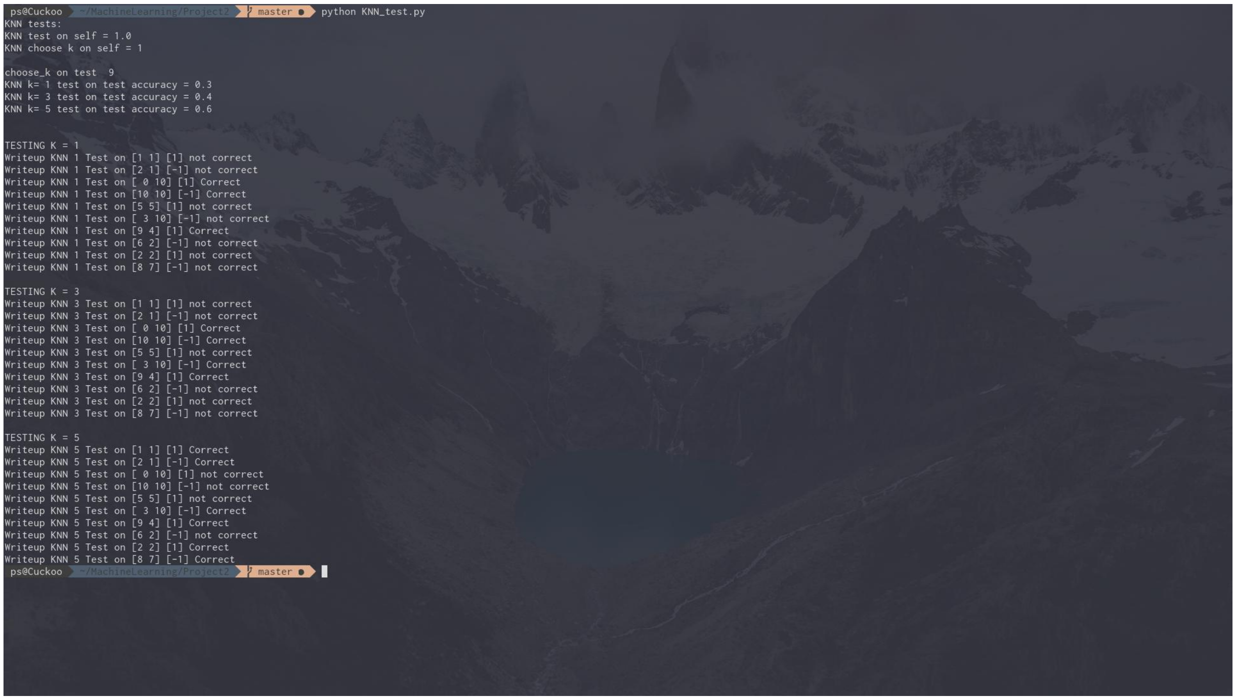
This function is simply checking all possible accuracies on the given test data and returns the index with the greatest accuracy which will be our K.

```
  OldAccuracy = 0
  NewAccuracy = 0
  Predicted_K = 1
  K = len(X_train)
  for i in range(1, K):
    OldAccuracy = KNN_test(X_train,Y_train,X_val,Y_val, i)
    if(OldAccuracy>NewAccuracy):
      NewAccuracy = OldAccuracy
      Predicted_K = i
  return Predicted_K

```

1.2 Algorithm Classification

Using the testing data given to us from the project description we will get the following results

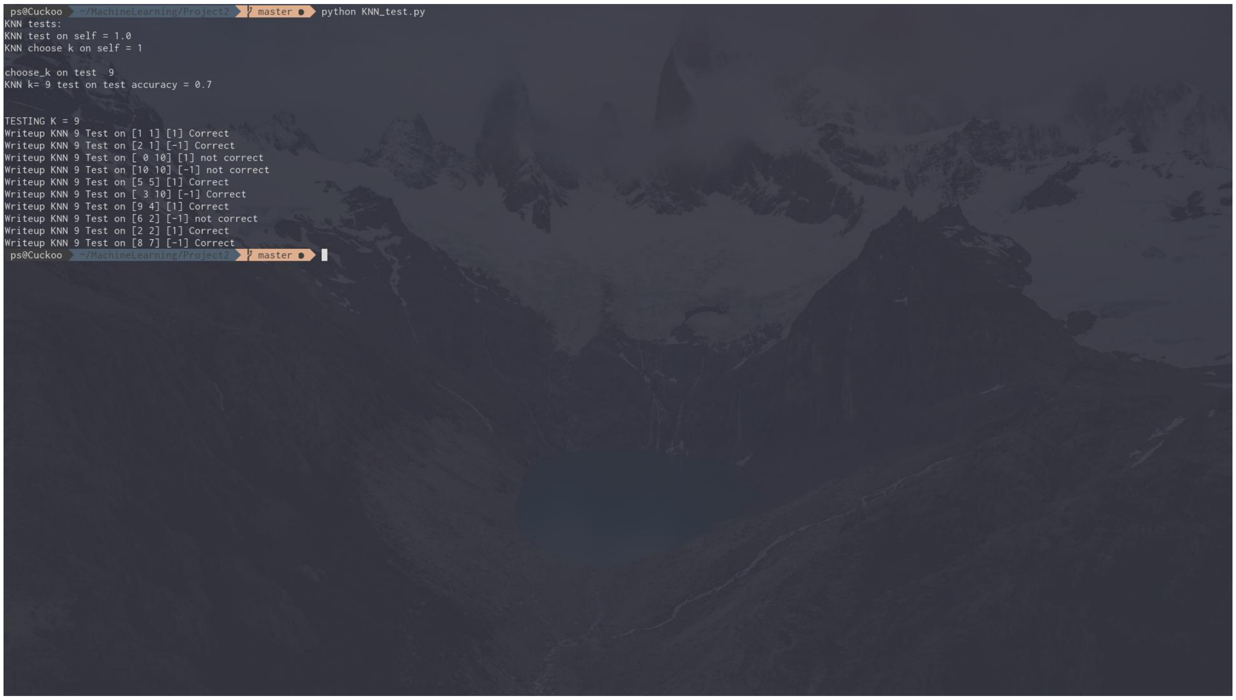
The accuracy on K=1, K=2, K=3 are 0.3, 0.4, 0.6 respectively. You can see more details like which sample was predicted incorrectly on the picture below



```
 ps@Cuckoo  ~/MachineLearning/Project2  master ●  python KNN_test.py
KNN tests:
KNN test on self = 1.0
KNN choose k on self = 1

choose_k on test  9
KNN k= 1 test on test accuracy = 0.3
KNN k= 3 test on test accuracy = 0.4
KNN k= 5 test on test accuracy = 0.6


TESTING K = 1
Writeup KNN 1 Test on [1 1] [1] not correct
Writeup KNN 1 Test on [2 1] [-1] not correct
Writeup KNN 1 Test on [ 0 10] [1] Correct
Writeup KNN 1 Test on [10 10] [-1] Correct
Writeup KNN 1 Test on [5 5] [1] not correct
Writeup KNN 1 Test on [ 3 10] [-1] not correct
Writeup KNN 1 Test on [9 4] [1] Correct
Writeup KNN 1 Test on [6 2] [-1] not correct
Writeup KNN 1 Test on [2 2] [1] not correct
Writeup KNN 1 Test on [8 7] [-1] not correct

TESTING K = 3
Writeup KNN 3 Test on [1 1] [1] not correct
Writeup KNN 3 Test on [2 1] [-1] not correct
Writeup KNN 3 Test on [ 0 10] [1] Correct
Writeup KNN 3 Test on [10 10] [-1] Correct
Writeup KNN 3 Test on [5 5] [1] not correct
Writeup KNN 3 Test on [ 3 10] [-1] Correct
Writeup KNN 3 Test on [9 4] [1] Correct
Writeup KNN 3 Test on [6 2] [-1] not correct
Writeup KNN 3 Test on [2 2] [1] not correct
Writeup KNN 3 Test on [8 7] [-1] not correct

TESTING K = 5
Writeup KNN 5 Test on [1 1] [1] Correct
Writeup KNN 5 Test on [2 1] [-1] Correct
Writeup KNN 5 Test on [ 0 10] [1] not correct
Writeup KNN 5 Test on [10 10] [-1] not correct
Writeup KNN 5 Test on [5 5] [1] not correct
Writeup KNN 5 Test on [ 3 10] [-1] Correct
Writeup KNN 5 Test on [9 4] [1] Correct
Writeup KNN 5 Test on [6 2] [-1] not correct
Writeup KNN 5 Test on [2 2] [1] Correct
Writeup KNN 5 Test on [8 7] [-1] Correct
 ps@Cuckoo  ~/MachineLearning/Project2  master ● 
```

[1]

The best K for our Algorithm would be 9 with the accuracy of 0.7, for more details like which specific sample was predicted wrong refer to the picture below.



```
 ps@Cuckoo  ~/MachineLearning/Project2  master ●  python KNN_test.py
KNN tests:
KNN test on self = 1.0
KNN choose k on self = 1

choose_k on test  9
KNN k= 9 test on test accuracy = 0.7


TESTING K = 9
Writeup KNN 9 Test on [1 1] [1] Correct
Writeup KNN 9 Test on [2 1] [-1] Correct
Writeup KNN 9 Test on [ 0 10] [1] not correct
Writeup KNN 9 Test on [10 10] [-1] not correct
Writeup KNN 9 Test on [5 5] [1] Correct
Writeup KNN 9 Test on [ 3 10] [-1] Correct
Writeup KNN 9 Test on [9 4] [1] Correct
Writeup KNN 9 Test on [6 2] [-1] not correct
Writeup KNN 9 Test on [2 2] [1] Correct
Writeup KNN 9 Test on [8 7] [-1] Correct
 ps@Cuckoo  ~/MachineLearning/Project2  master ● 
```

[2]

[1] The picture has a high resolution so you should be able to zoom in and see the detail, but the picure has also been included in the zip file so please refer to that if there is any issues

[2] The picture has a high resolution so you should be able to zoom in and see the detail, but the picure has also been included in the zip file so please refer to that if there is any issues

2 KMeans Clustering

2.1 Describe Implementation

2.1.1 K_Means

Our implementation for KMeans starts with selecting a random number generated using the random library. This number is used as an index to grab the points from the data; these points are selected as our center.

The datapoints are passed into a function along with the centers and enter a while loop with the condition being it runs until error equals 0. Error is defined as the centers from one run not being the same centers generated from the next run.

A list of lists is generated depending on the number of centers passed in, e.g. if three centers were generated, K=3, so three clusters are necessary. Each point in the data is tested against the center points, calculating the Euclidean distance and putting the point in the corresponding cluster that had the least amount of distance to the center.

The new centers are then calculated using the mean of each individual cluster and tested against the old centers to see if the loop should be exited. If they are not the same these steps are repeated.

```
randCenterIndexes = random.sample(range(len(X)), K)
randCenters = []
for k in randCenterIndexes:
randCenters.append(X[k])

output = FindClusters(X, randCenters)

return output

```

2.1.2 K_Means_better

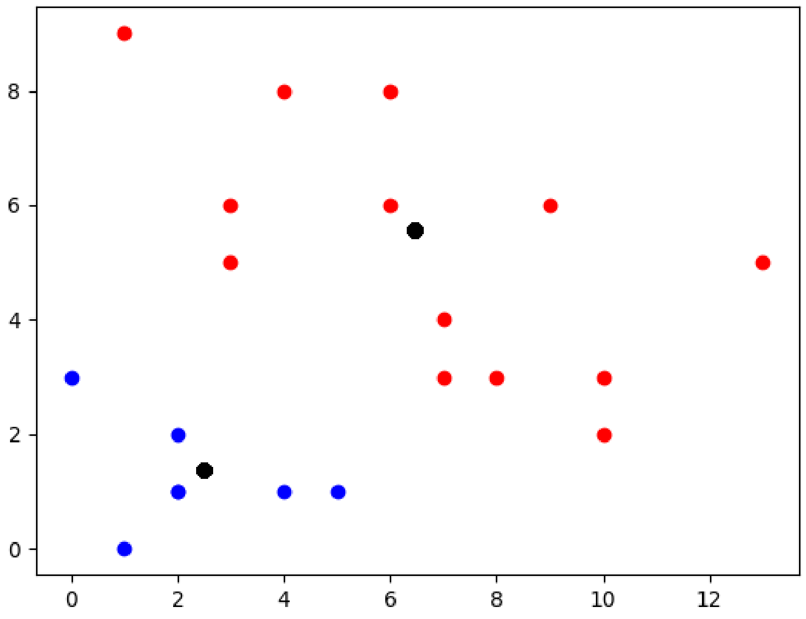
This function takes X, our sample data-set and the K value and runs the K-Mean algorithm 1000 times (can be changed by the user) we will then count the number of occurrences of each cluster center and return the ones with the most repeated or occurred cluster centers, this value will be stored in a numpy array of cluster centers and then returned.

```
  def K_Means_better(X, K):
  run = 1000
  bestCenterList = []
  occurences = []
  j = 0
  for i in range(run):
    try:
      result = K_Means(X, K)
      bestCenterList.append(result)
      j += 1
  except:
    pass
  for j in range(len(bestCenterList)):
    count = bestCenterList.count(bestCenterList[j])
    occurences.append(count)
  maxNum = 0
  for k in occurences:
    if (k > maxNum):
      maxNum = k
  return bestCenterList[maxNum]

```

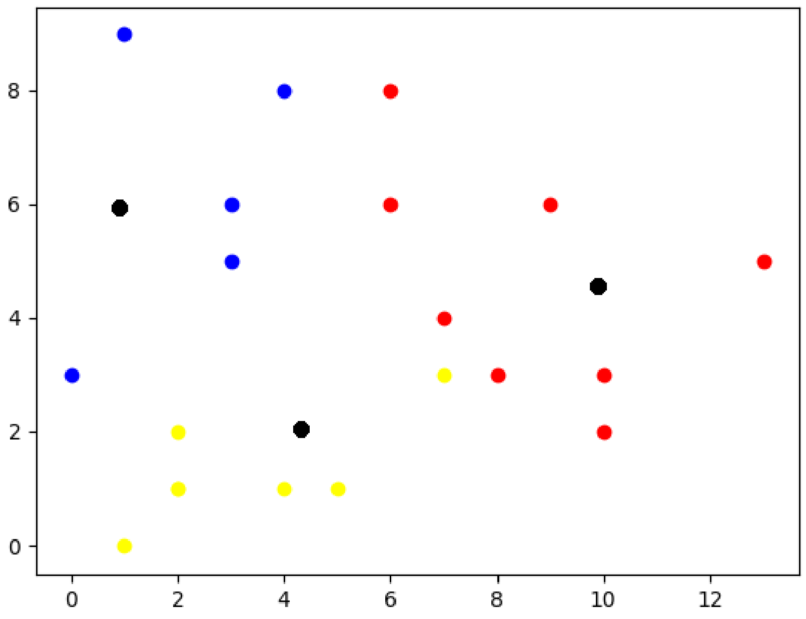
2.2 Algorithm Classification

2.2.1 KMeans on K = 2, K = 3

K = 2 The cluster centers have been labeled as a black dot.

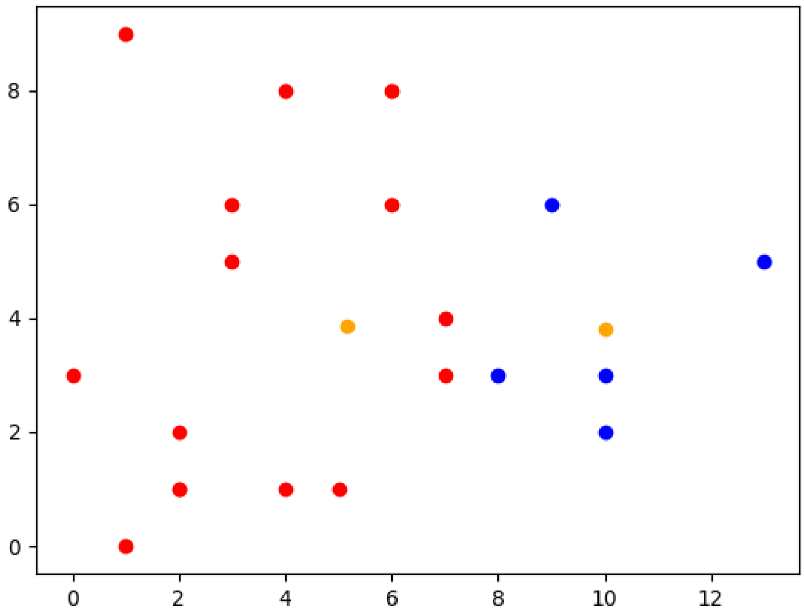


K = 3 The cluster centers have been labeled as a black dot.

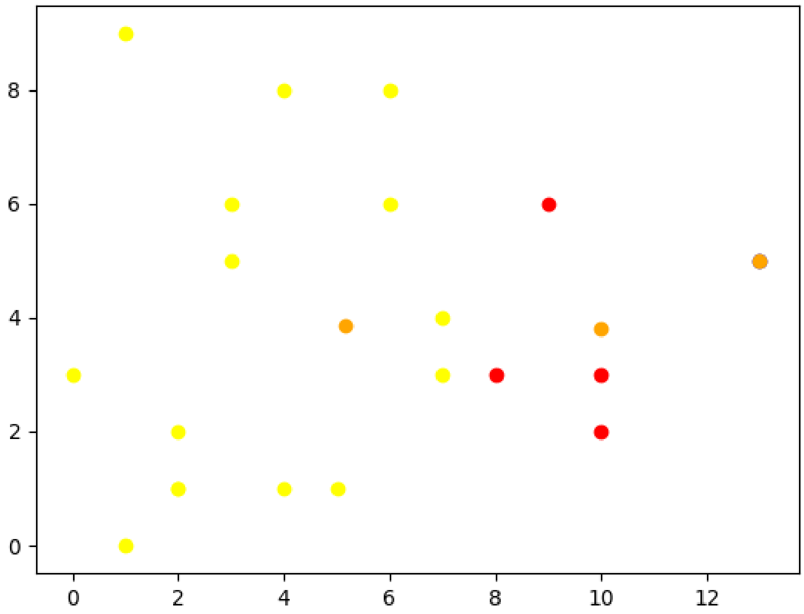


2.2.2 KMeans Better on K = 2, K = 3

K = 2 The centers are the two orange dots



K = 3 the center is the point at 13, 4.5 (it is a point and a center)



3 Perceptron

3.1 Describe Implementation

3.1.1 perceptron_train

This function will be computing a weight (vector) and a bias using the data-set X and the labels for that set Y. We start with the initial weights and bias of 0 and we will kep track of epoch by a variable called epoch, we would also have to consider the fact that some data might not be linearly separable hence we will have another variable called maxEpoch which we would use to break our if the algorithm is not successful after maxEpoch tries. We will be using a while loop that checks if we are done yet, so we've gone an entire epoch without updating, or if we have reached the maxEpoch in which case the program gets out of the while loop and returns the weights and bias of the last iteration. Inside the while loop we are computing the activation and if the activation returns a number less than or equal to 0 we would ned to update hence we update our weight vector and the bias, they're both in the same list, the first element is weight vector, 2nd is bias(W_B_List) if data is linearly separable we would update our Done flag to true and get out of the loop and finally return the weight vector and bias (W_B_List)

```
Done = False
epoch = 0
maxEpoch = 199
activation = 0
W_B_List = [[],[]]
W_B_List[0] =[0]*len(X[0])
W_B_List[1] = [0]
while ( not Done or epoch < maxEpoch):
  epoch+=1
  for i in range (len(X)):
    activation = 0
    for m in range(len(X[i])):
      activation += X[i][m] * W_B_List[0][m]
    activation+= W_B_List[1][0]
    if  (( activation * Y[i][0] ) <= 0):
      for j in range(len(X[0])):
        W_B_List[0][j] = W_B_List[0][j] + Y[i][0] * X[i][j]
      W_B_List[1][0] = W_B_List[1][0] + Y[i][0]
    else:
      Done = True
return W_B_List
```

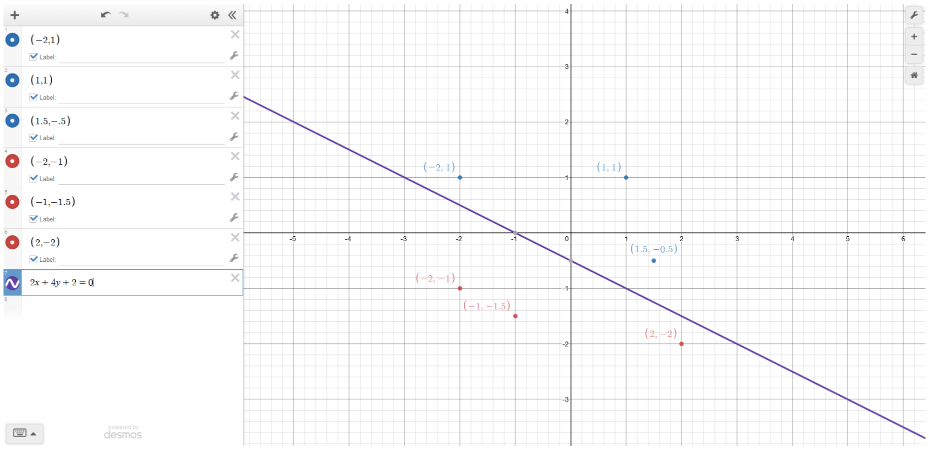
3.1.2 perceptron_test

This function computes the accuracy of the weights and bias given the data-set X and their label Y. This function computes the activation for each of the samples in the data-set X and compares it with their label. If the activation matches the label we would increment our NumberOfCorrect variable at the end we will divide this number by the length of our data-set X to find the accuracy and we would return that number.

```
NumberOfCorrect = 0
activation = 0
for i, j in enumerate(X_test):
  activation = np.dot(j,w) + b
  if activation * Y_test[i] > 0:
    NumberOfCorrect+=1
return NumberOfCorrect/len(X_test)
```

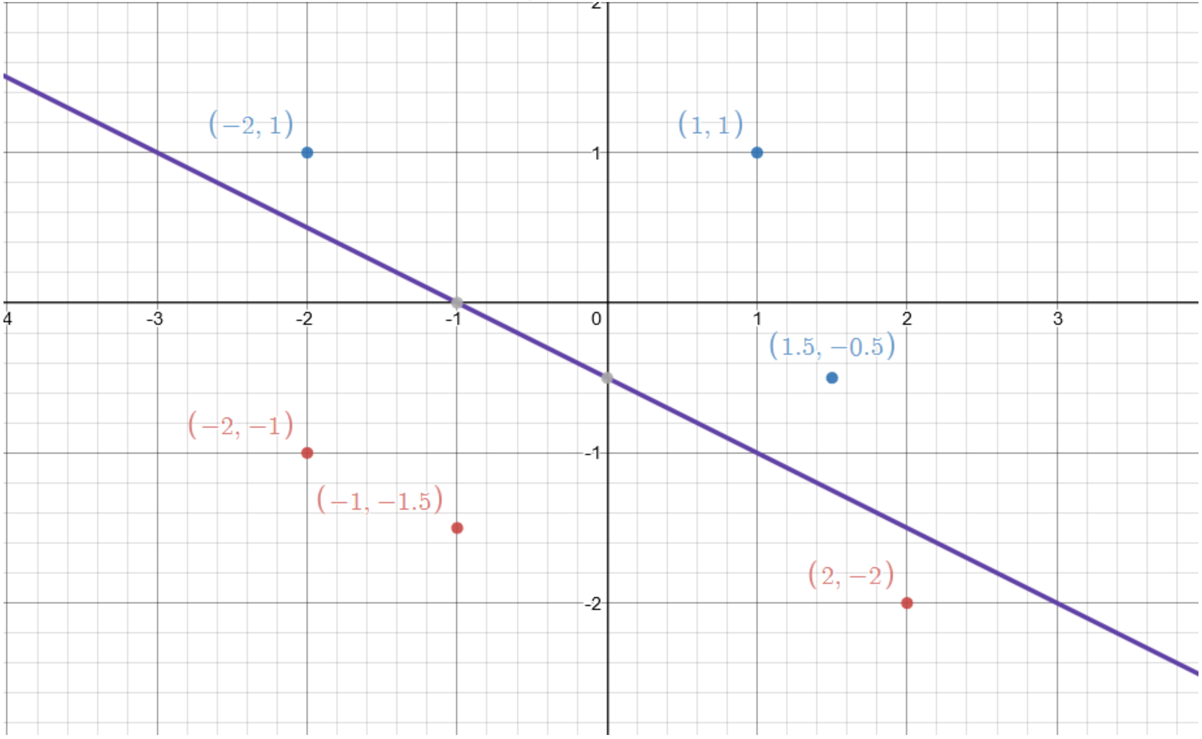
3.2 Decision Boundary

I have used the Desmos website to plot the points and the decision boundary for this part. Blue points are the ones with label 1 and red points have label of -1. The algorithms returns 2,4 for our weights and 2 for our bias hence we will be using the function $2X_1 + 4X_2 + 2 = 0$ we could simplify this function but since we will be drawing it to show the decision boundary there is no need.


(−2,1)
Label:
(1,1)
Label:
(1.5,−.5)
Label:
(−2,−1)
Label:
(−1,−1.5)
Label:
(2,−2)
Label:
2x + 4y + 2 = 0

[3]

scaled larger for better visibility



[3] The picture has a high resolution so you should be able to zoom in and see the detail, but the picure has also been included in the zip file so please refer to that if there is any issues