Vibhor Agarwal

2020349

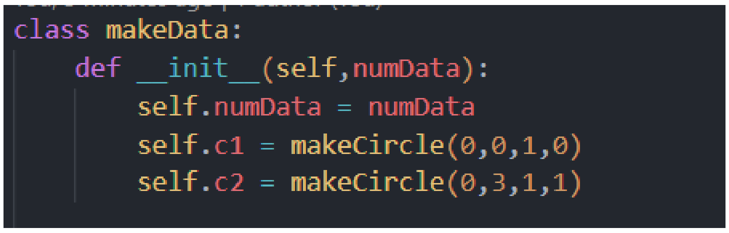
ML Assignment 2 Report

Section B Report

Part A

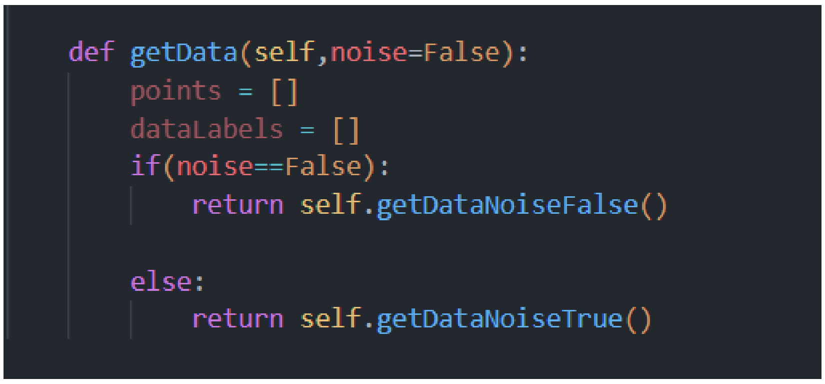
Here we implement classes makeCircle and makeData which are used for plotting a circle to get the points on it and makeData which is used to make the dataset using the points on the circle.

- The following class takes the number of points as the input



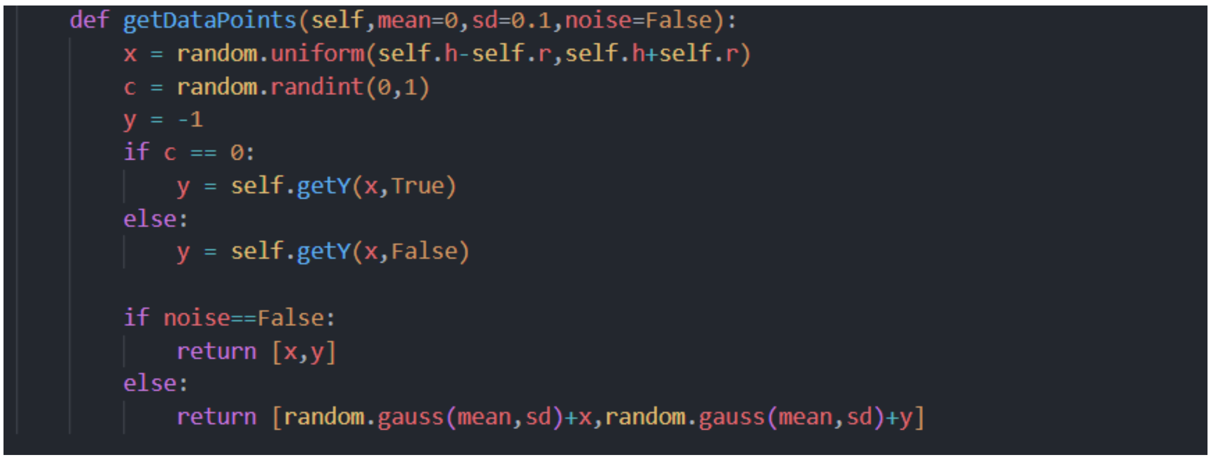
```
class makeData:
    def __init__(self,numData):
        self.numData = numData
        self.c1 = makeCircle(0,0,1,0)
        self.c2 = makeCircle(0,3,1,1)
```

- The below function returns a predefined number of points and has a noise argument



```
    def getData(self,noise=False):
        points = []
        dataLabels = []
        if(noise==False):
            return self.getDataNoiseFalse()

        else:
            return self.getDataNoiseTrue()
```

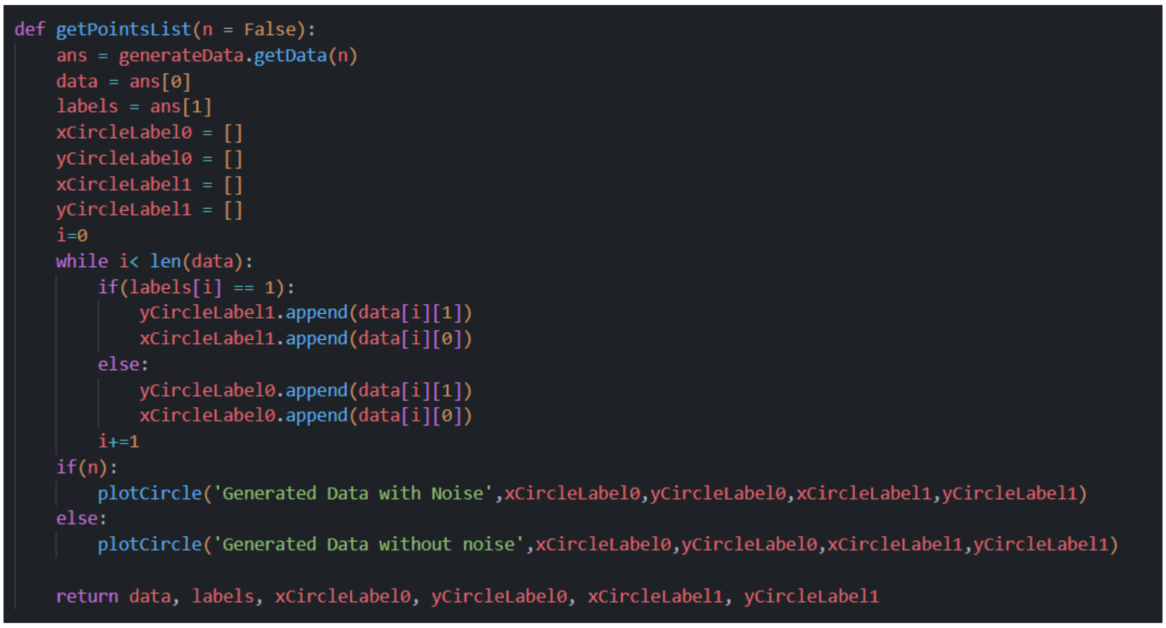
- The following function adds the noise to the data if noise=True. The noise is gaussian with mean =0 and sd=0.1



```
    def getDataPoints(self,mean=0,sd=0.1,noise=False):
        x = random.uniform(self.h-self.r,self.h+self.r)
        c = random.randint(0,1)
        y = -1
        if c == 0:
            y = self.getY(x,True)
        else:
            y = self.getY(x,False)

        if noise==False:
            return [x,y]
        else:
            return [random.gauss(mean,sd)+x,random.gauss(mean,sd)+y]
```

Part B

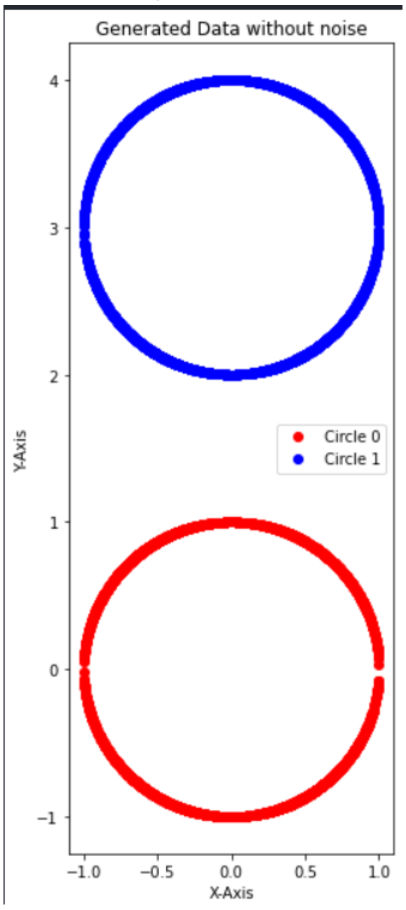
- Now to plot the data generated above with and without noise we use the following function



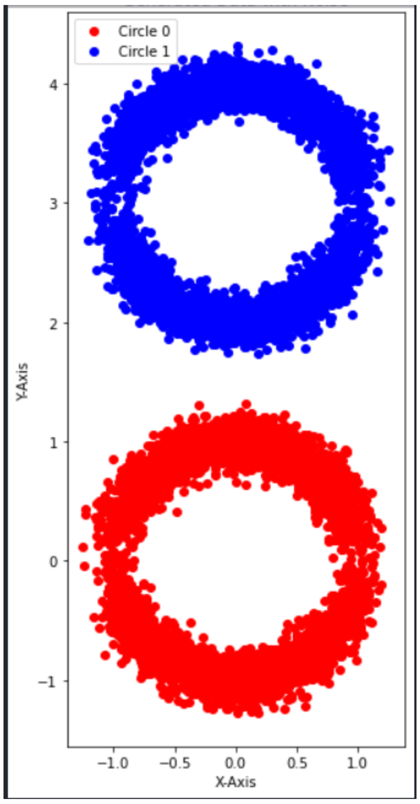
```
def getPointsList(n = False):
    ans = generateData.getData(n)
    data = ans[0]
    labels = ans[1]
    xCircleLabel0 = []
    yCircleLabel0 = []
    xCircleLabel1 = []
    yCircleLabel1 = []
    i=0
    while i< len(data):
        if(labels[i] == 1):
            yCircleLabel1.append(data[i][1])
            xCircleLabel1.append(data[i][0])
        else:
            yCircleLabel0.append(data[i][1])
            xCircleLabel0.append(data[i][0])
        i+=1
    if(n):
        plotCircle('Generated Data with Noise',xCircleLabel0,yCircleLabel0,xCircleLabel1,yCircleLabel1)
    else:
        plotCircle('Generated Data without noise',xCircleLabel0,yCircleLabel0,xCircleLabel1,yCircleLabel1)

    return data, labels, xCircleLabel0, yCircleLabel0, xCircleLabel1, yCircleLabel1
```

This function first gets the data using getData(noise) above and then separates the label and data. If the label is 1, we separate the data into two different list which contain the points with label and similarly separate the data without label. We then use Matlplotlib plot() function to plot the data which is in form of two circles.

- The following data is data without the noise



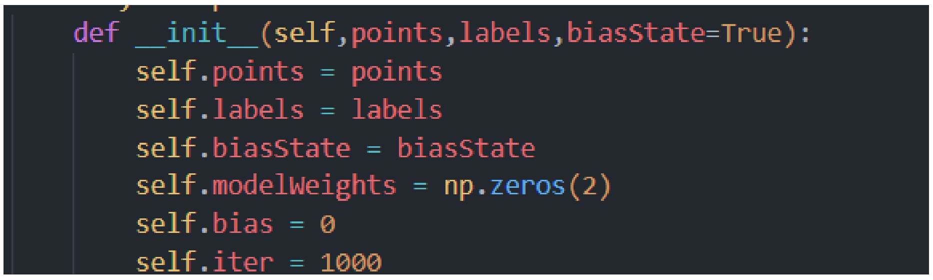
- The following data is data with noise



- As we can see that on adding the noise not all points exactly lie on the circle, but they are still linearly separable as a line clearly separates them.

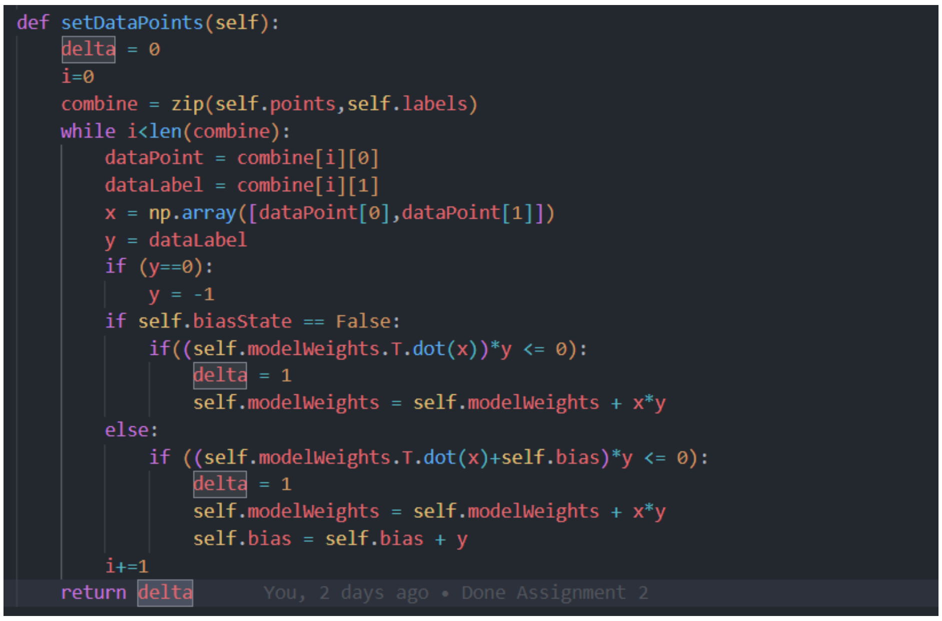
Part C

- Here we first implement MyPerceptron class which takes datapoints and datalabels as the input and trains the perceptron using the below function.



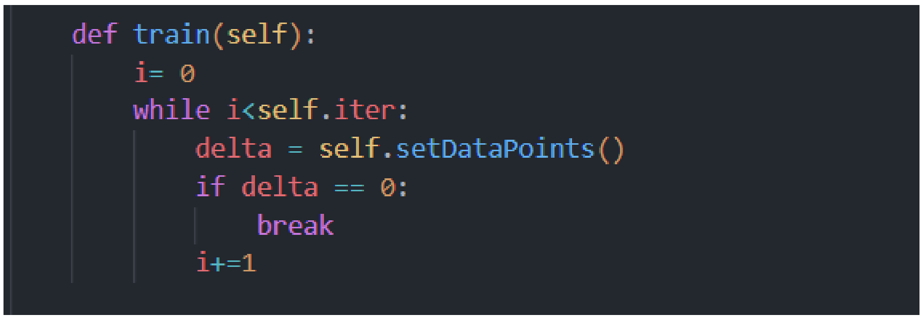
```
    def __init__(self,points,labels,biasState=True):
        self.points = points
        self.labels = labels
        self.biasState = biasState
        self.modelWeights = np.zeros(2)
        self.bias = 0
        self.iter = 1000
```

- The following function runs the perceptron training algorithm for 1 iteration and returns the delta in the weight.



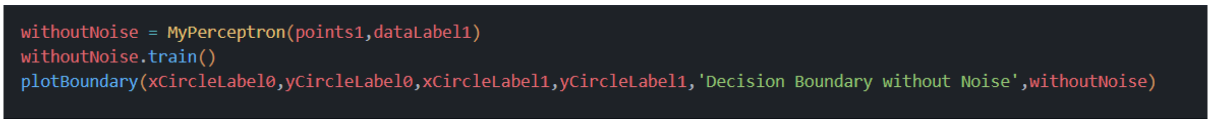
```
def setDataPoints(self):
    delta = 0
    i=0
    combine = zip(self.points,self.labels)
    while i<len(combine):
        dataPoint = combine[i][0]
        dataLabel = combine[i][1]
        x = np.array([dataPoint[0],dataPoint[1]])
        y = dataLabel
        if (y==0):
            y = -1
        if self.biasState == False:
            if((self.modelWeights.T.dot(x))*y <= 0):
                delta = 1
                self.modelWeights = self.modelWeights + x*y
        else:
            if ((self.modelWeights.T.dot(x)+self.bias)*y <= 0):
                delta = 1
                self.modelWeights = self.modelWeights + x*y
                self.bias = self.bias + y
        i+=1
    return delta        You, 2 days ago • Done Assignment 2
```

- This function runs the above function multiple times and stops when the delta becomes zero ie. the weights have converged.



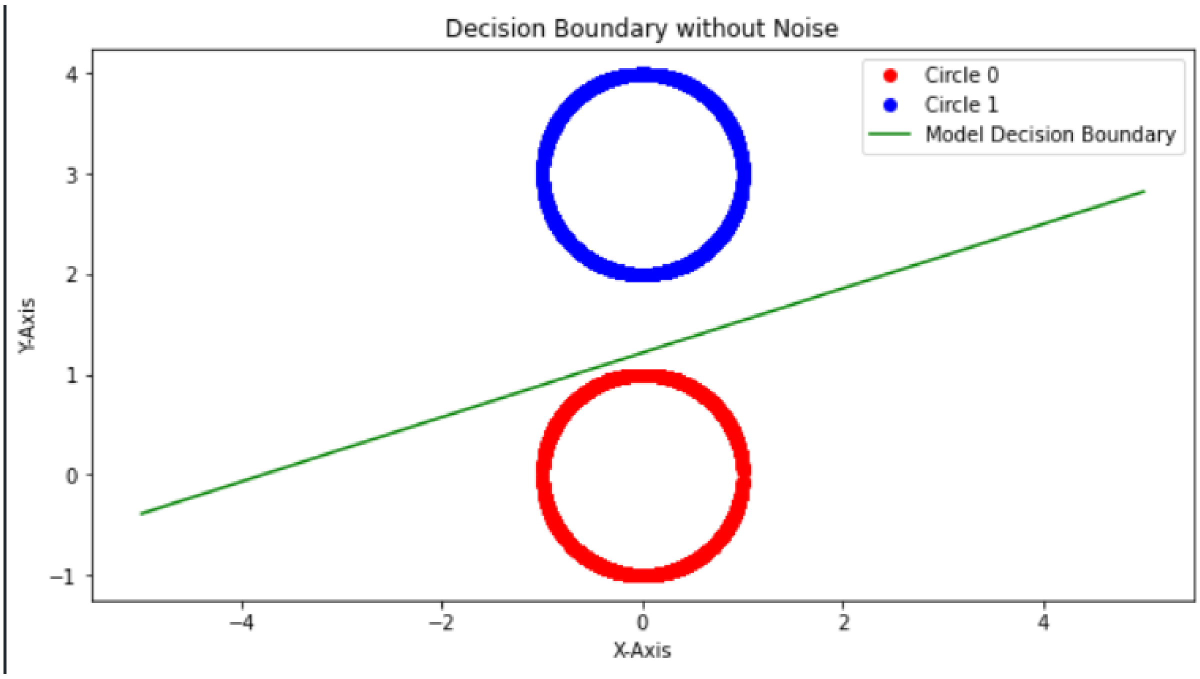
```
def train(self):
    i= 0
    while i<self.iter:
        delta = self.setDataPoints()
        if delta == 0:
            break
        i+=1
```

- We now run the perceptron training algorithm discussed above. The following functions run and plot the data for the plots and labels supplied ie. with and without the data.

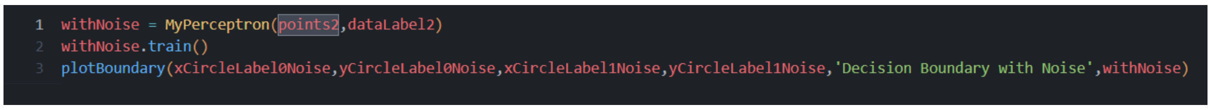


```
withoutNoise = MyPerceptron(points1,dataLabel1)
withoutNoise.train()
plotBoundary(xCircleLabel0,yCircleLabel0,xCircleLabel1,yCircleLabel1,'Decision Boundary without Noise',withoutNoise)
```

- The data we obtain for without Noise is as follows

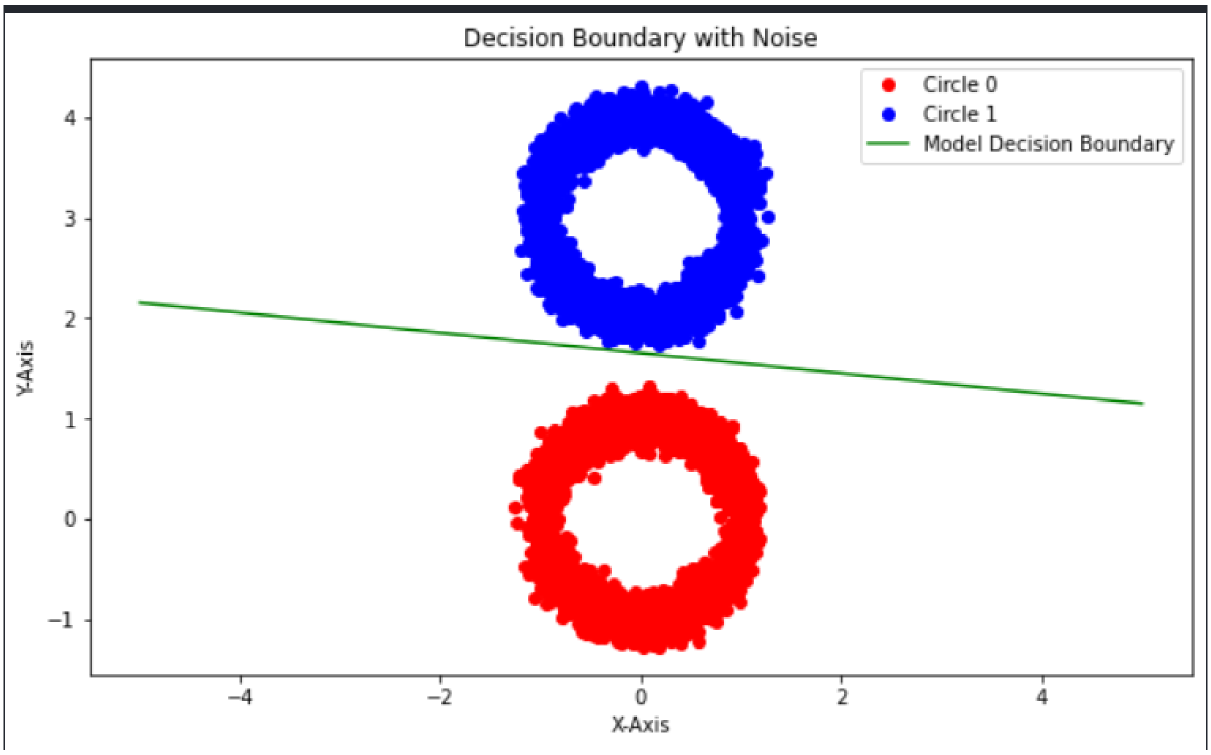


- Similarly we run the PTA for data with noise



```
withNoise = MyPerceptron(points2,dataLabel2)
withNoise.train()
plotBoundary(xCircleLabel0Noise,yCircleLabel0Noise,xCircleLabel1Noise,yCircleLabel1Noise,'Decision Boundary with Noise',withNoise)
```

- We get the following results for data with noise



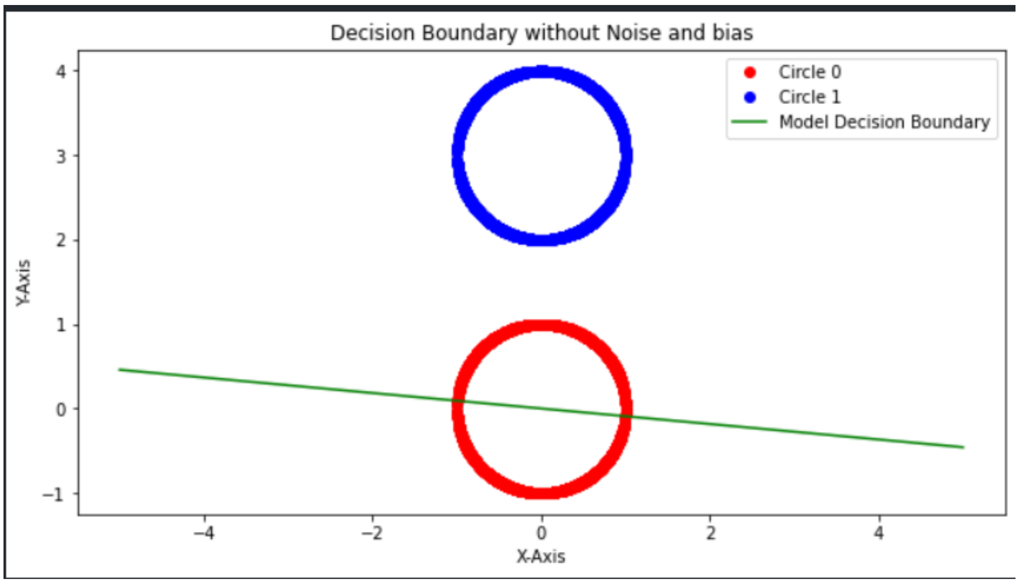
- We clearly observe there exists a decision boundary for data with and without the noise. Hence the data is linearly separable irrespective of the noise.

Part D

- We now make the bias = 0 by setting biasState = False in the following code. By setting biasState = False we set the bias = 0 in our MyPerceptron class.

```
withoutNoiseandBias = MyPerceptron(points1,dataLabel1,biasState=False)
withoutNoiseandBias.train()
```

- The result that we obtain are as follows



Clearly we observe that the data is not linearly separable and the obtained decision boundary misclassifies many of the data points.

Comparison between the results obtained with and without bias

We observe that while with bias we get a decision boundary which can linearly separate the data but without bias we can't get the boundary which linearly separates the data. Here equation of line with bias is ax+by+c=0 while without bias is ax+by=0. Now the circle in the red is centred at (0,0) so the line without bias must pass through the origin. But since it must pass through the origin it must misclassify some points on the circle in red as it will always pass through it and never above it. But the same is not true for line without bias as by adjusting the bias we can get a line passes through the space between both the circles and is clearly able to correctly classify all the data points.

So, we conclude that without having a bias we can never have a line that will correctly serve as the decision boundary between the two circles.

Part E

- We first store all the 4 points that serve as input for and gate and their respective output labels

```
possiblePoints = np.array([[0,0],[0,1],[1,0],[1,1]])
LabelAND = np.array([0,0,0,1])
LabelOR = np.array([0,1,1,1])
LabelXOR = np.array([0,1,1,0])
```

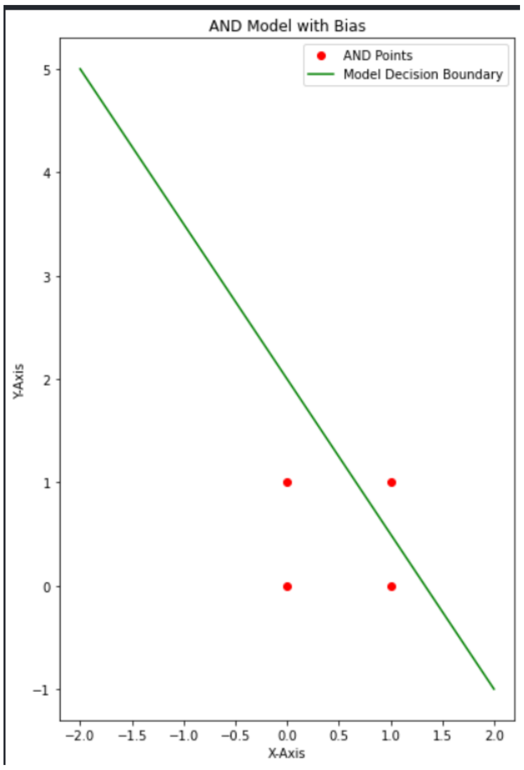
- We now run our model for with and without bias for all the three gates using the same perceptron training class earlier.

AND GATE with Bias

The following code is used to train perceptron for AND gate with BIAS

```
ANDModel = MyPerceptron(possiblePoints,LabelAND)
ANDModel.train()
plotGate('AND Model with Bias',ANDModel,'AND Points')
```

The results obtained is as follows



As is clear the decision boundary is correctly able to classify all the points.

AND GATE without BIAS

The following code is used to train perceptron for AND gate without BIAS

```
ANDModelwithoutBias = MyPerceptron(possiblePoints,LabelAND,biasState=False)
ANDModelwithoutBias.train()
plotGate('AND Model without Bias',ANDModelwithoutBias,'AND Points',hasNoBias=True)
```

The results obtained are as follows

```
...   The model is not possible
```

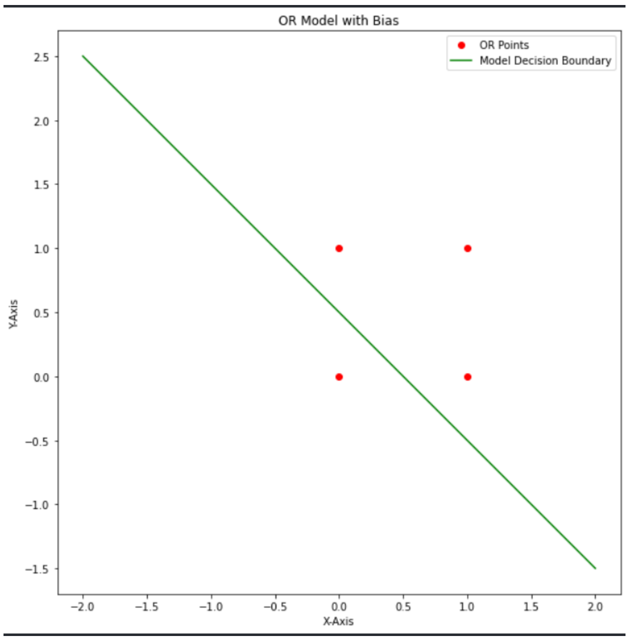
This clearly shows that a decision without bias does not exist that can correctly classify all the points of the AND Gate.

OR GATE with BIAS

The following code is used to train perceptron for OR gate with BIAS

```
ORModel = MyPerceptron(possiblePoints,LabelOR)
ORModel.train()
plotGate('OR Model with Bias',ORModel,'OR Points')
✓  0.2s
```

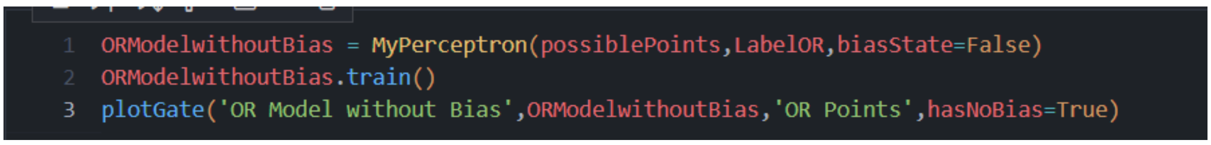
The results we obtain are as follows



As it is clear that the decision boundary obtained with BIAS for OR Gate is correctly able to classify all the points. So a decision boundary does exist in this case

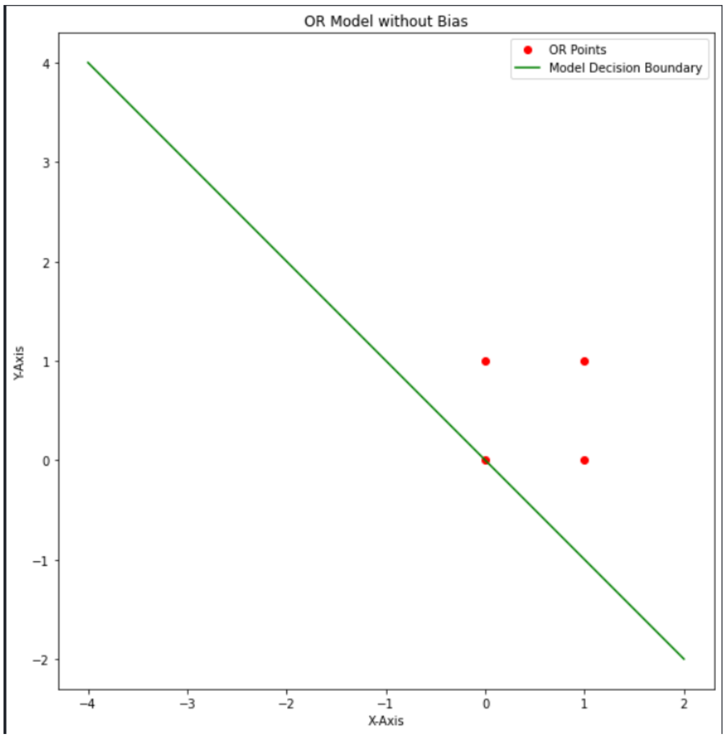
OR GATE without BIAS

The following code is used to train perceptron for OR gate without BIAS



```python
ORModelwithoutBias = MyPerceptron(possiblePoints,LabelOR,biasState=False)
ORModelwithoutBias.train()
plotGate('OR Model without Bias',ORModelwithoutBias,'OR Points',hasNoBias=True)
```

The results we obtain are as follows

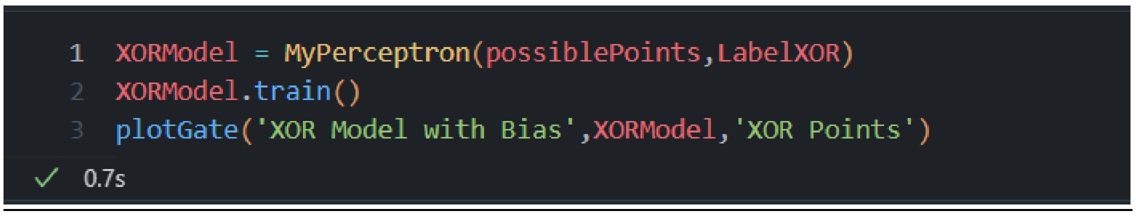


As the decision boundary obtained with BIAS for OR Gate is correctly able to classify all the points.

We have taken the points on the line to be classified as 0. So, a decision boundary does exist in this case and is able to classify (0,0) in one set and other points into other.

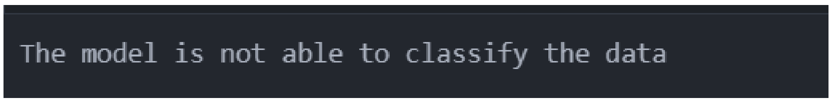
For XOR Gate with BIAS

The following code is used to train perceptron for XOR gate with BIAS



```python
XORModel = MyPerceptron(possiblePoints,LabelXOR)
XORModel.train()
plotGate('XOR Model with Bias',XORModel,'XOR Points')
```
```
✓ 0.7s
```

The results obtained are as follows

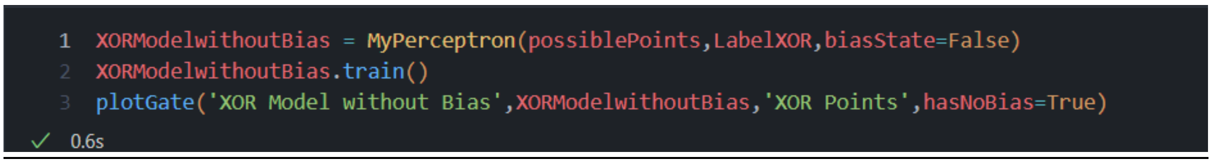


```
The model is not able to classify the data
```

It is clear that for XOR Gate there exists no decision boundary with BIAS that can correctly classify all the points correctly.

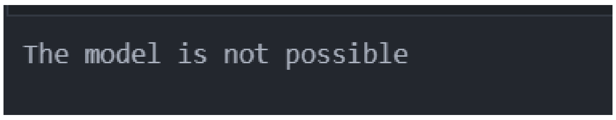
<u>For XOR Gate without BIAS</u>

The following code is used to train perceptron for XOR gate without BIAS



```python
XORModelwithoutBias = MyPerceptron(possiblePoints,LabelXOR,biasState=False)
XORModelwithoutBias.train()
plotGate('XOR Model without Bias',XORModelwithoutBias,'XOR Points',hasNoBias=True)
```

The results obtained are as follows



```
The model is not possible
```

For XOR Gate there exists no decision boundary without BIAS that can correctly classify all the points correctly.

Part F

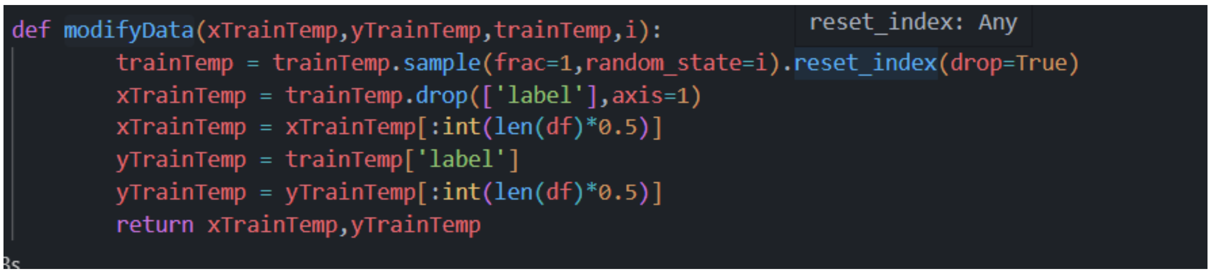
We know that a hyperplane divides a space in two different parts. The points on one side of the plane can be classified as 0 and the points on the other side of the plane as 1.

- We first put the value of the point in the plane. If the value is > 0 then we will classify the point as 1.
- If the value is <0, then we will classify the point as 0.
- Now if the value is 0 i.e., the point lies on the plane then we will classify the point as 0. We assume that the points on the plane will always be labelled as 0.

Section C

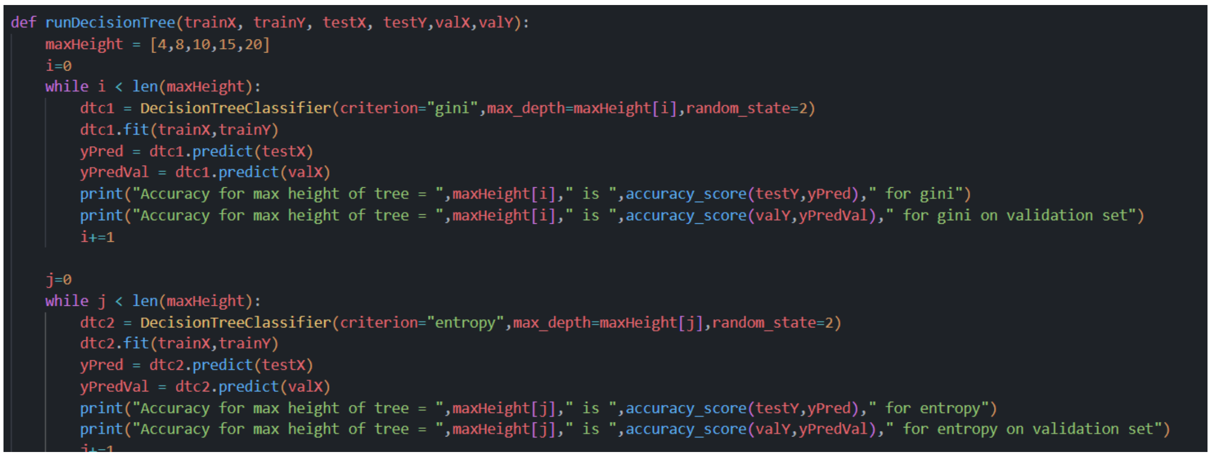
Part A

- First, we import the necessary library, read our data, and convert it to a dataframe
- We use Label Encoder to convert the strings in label and address columns to numerical values. This helps us train the model efficiently on the dataset.
- We split the data into training, validation and testing sets in 70:15:15 split using the below function



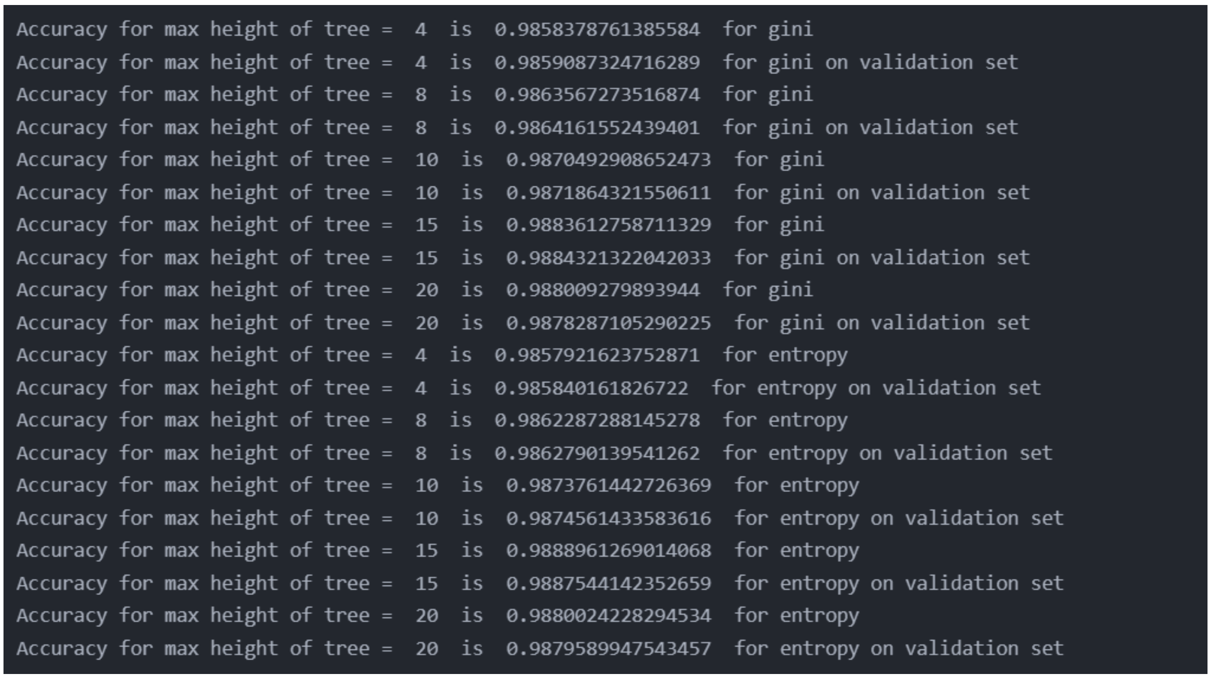
```
def modifyData(xTrainTemp,yTrainTemp,trainTemp,i):
        trainTemp = trainTemp.sample(frac=1,random_state=i).reset_index(drop=True)
        xTrainTemp = trainTemp.drop(['label'],axis=1)
        xTrainTemp = xTrainTemp[:int(len(df)*0.5)]
        yTrainTemp = trainTemp['label']
        yTrainTemp = yTrainTemp[:int(len(df)*0.5)]
        return xTrainTemp,yTrainTemp
```

- We now run SKLearn's decision tree using the following code



```
def runDecisionTree(trainX, trainY, testX, testY,valX,valY):
    maxHeight = [4,8,10,15,20]
    i=0
    while i < len(maxHeight):
        dtc1 = DecisionTreeClassifier(criterion="gini",max_depth=maxHeight[i],random_state=2)
        dtc1.fit(trainX,trainY)
        yPred = dtc1.predict(testX)
        yPredVal = dtc1.predict(valX)
        print("Accuracy for max height of tree = ",maxHeight[i]," is ",accuracy_score(testY,yPred)," for gini")
        print("Accuracy for max height of tree = ",maxHeight[i]," is ",accuracy_score(valY,yPredVal)," for gini on validation set")
        i+=1

    j=0
    while j < len(maxHeight):
        dtc2 = DecisionTreeClassifier(criterion="entropy",max_depth=maxHeight[j],random_state=2)
        dtc2.fit(trainX,trainY)
        yPred = dtc2.predict(testX)
        yPredVal = dtc2.predict(valX)
        print("Accuracy for max height of tree = ",maxHeight[j]," is ",accuracy_score(testY,yPred)," for entropy")
        print("Accuracy for max height of tree = ",maxHeight[j]," is ",accuracy_score(valY,yPredVal)," for entropy on validation set")
```

- The results obtained are as follows

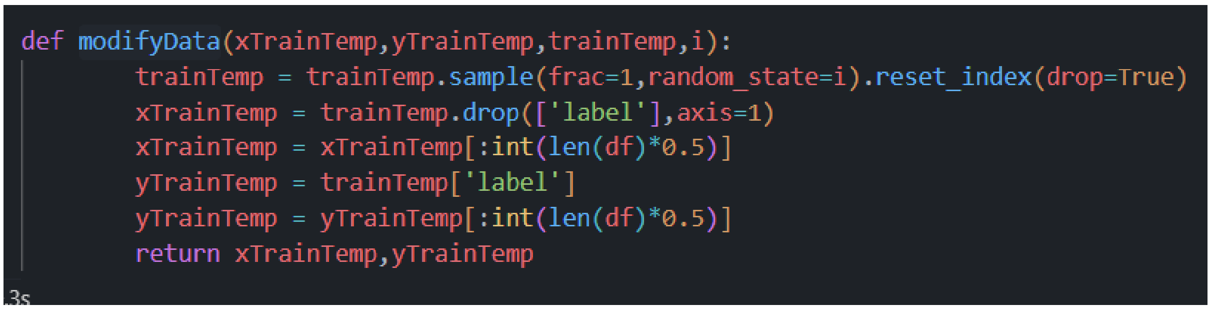


```
Accuracy for max height of tree =  4  is  0.9858378761385584  for gini
Accuracy for max height of tree =  4  is  0.9859087324716289  for gini on validation set
Accuracy for max height of tree =  8  is  0.9863567273516874  for gini
Accuracy for max height of tree =  8  is  0.9864161552439401  for gini on validation set
Accuracy for max height of tree =  10  is  0.9870492908652473  for gini
Accuracy for max height of tree =  10  is  0.9871864321550611  for gini on validation set
Accuracy for max height of tree =  15  is  0.9883612758711329  for gini
Accuracy for max height of tree =  15  is  0.9884321322042033  for gini on validation set
Accuracy for max height of tree =  20  is  0.988009279893944  for gini
Accuracy for max height of tree =  20  is  0.9878287105290225  for gini on validation set
Accuracy for max height of tree =  4  is  0.9857921623752871  for entropy
Accuracy for max height of tree =  4  is  0.985840161826722  for entropy on validation set
Accuracy for max height of tree =  8  is  0.9862287288145278  for entropy
Accuracy for max height of tree =  8  is  0.9862790139541262  for entropy on validation set
Accuracy for max height of tree =  10  is  0.9873761442726369  for entropy
Accuracy for max height of tree =  10  is  0.9874561433583616  for entropy on validation set
Accuracy for max height of tree =  15  is  0.9888961269014068  for entropy
Accuracy for max height of tree =  15  is  0.9887544142352659  for entropy on validation set
Accuracy for max height of tree =  20  is  0.9880024228294534  for entropy
Accuracy for max height of tree =  20  is  0.9879589947543457  for entropy on validation set
```

- The max accuracy is found for max height/depth of the tree =15 and using entropy as the criteria. We therefore use these values for further analysis.

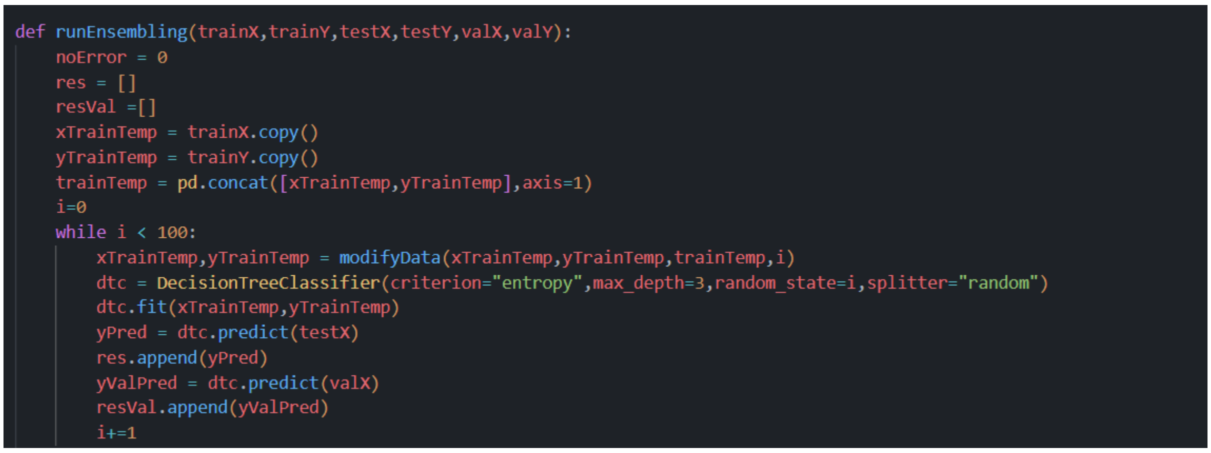
Part B

- Here for 100 iterations and max depth =3, we randomly select 50% of the training data using the following function



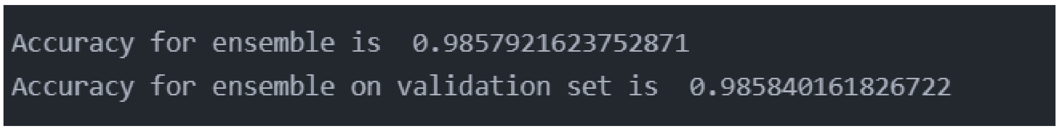
```
def modifyData(xTrainTemp,yTrainTemp,trainTemp,i):
        trainTemp = trainTemp.sample(frac=1,random_state=i).reset_index(drop=True)
        xTrainTemp = trainTemp.drop(['label'],axis=1)
        xTrainTemp = xTrainTemp[:int(len(df)*0.5)]
        yTrainTemp = trainTemp['label']
        yTrainTemp = yTrainTemp[:int(len(df)*0.5)]
        return xTrainTemp,yTrainTemp
3s
```

- We run decision tree on this data using the following code



```
def runEnsembling(trainX,trainY,testX,testY,valX,valY):
    noError = 0
    res = []
    resVal =[]
    xTrainTemp = trainX.copy()
    yTrainTemp = trainY.copy()
    trainTemp = pd.concat([xTrainTemp,yTrainTemp],axis=1)
    i=0
    while i < 100:
        xTrainTemp,yTrainTemp = modifyData(xTrainTemp,yTrainTemp,trainTemp,i)
        dtc = DecisionTreeClassifier(criterion="entropy",max_depth=3,random_state=i,splitter="random")
        dtc.fit(xTrainTemp,yTrainTemp)
        yPred = dtc.predict(testX)
        res.append(yPred)
        yValPred = dtc.predict(valX)
        resVal.append(yValPred)
        i+=1
```

- The results obtained are as follows

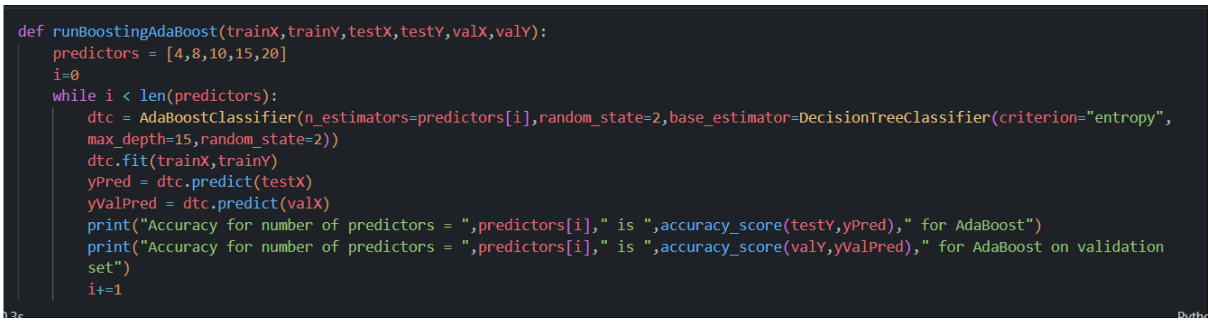


```
Accuracy for ensemble is  0.9857921623752871
Accuracy for ensemble on validation set is  0.985840161826722
```

- We observe that even for max depth =3 the result is almost as accurate as running the decision tree with max depth =15 in earlier parts. Also, for max depth =4 the results are identical. This shows that ensembling indeed works very well despite using week classifiers.

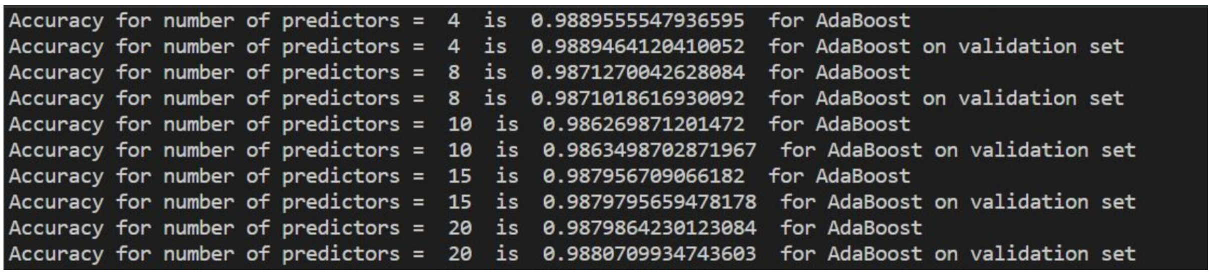
Part C

- We run the AdaBoost using the following code and taking max depth =15 and entropy criteria.



```python
def runBoostingAdaBoost(trainX,trainY,testX,testY,valX,valY):
    predictors = [4,8,10,15,20]
    i=0
    while i < len(predictors):
        dtc = AdaBoostClassifier(n_estimators=predictors[i],random_state=2,base_estimator=DecisionTreeClassifier(criterion="entropy",
        max_depth=15,random_state=2))
        dtc.fit(trainX,trainY)
        yPred = dtc.predict(testX)
        yValPred = dtc.predict(valX)
        print("Accuracy for number of predictors = ",predictors[i]," is ",accuracy_score(testY,yPred)," for AdaBoost")
        print("Accuracy for number of predictors = ",predictors[i]," is ",accuracy_score(valY,yValPred)," for AdaBoost on validation
        set")
        i+=1
```

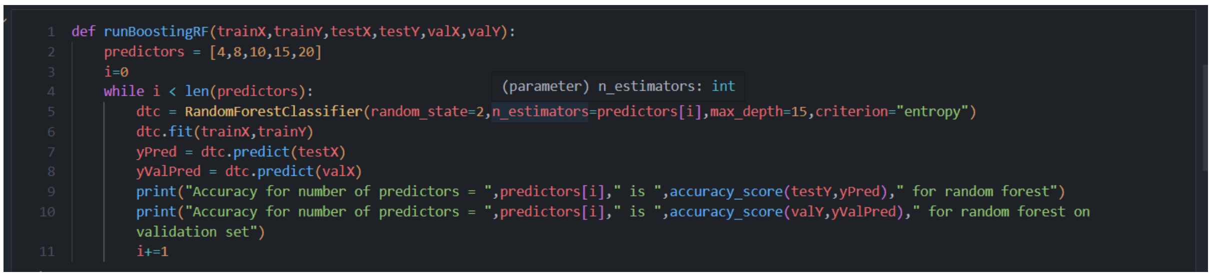
- The results that we obtain are as follows



```
Accuracy for number of predictors =  4  is  0.9889555547936595  for AdaBoost
Accuracy for number of predictors =  4  is  0.9889464120410052  for AdaBoost on validation set
Accuracy for number of predictors =  8  is  0.9871270042628084  for AdaBoost
Accuracy for number of predictors =  8  is  0.9871018616930092  for AdaBoost on validation set
Accuracy for number of predictors =  10  is  0.986269871201472  for AdaBoost
Accuracy for number of predictors =  10  is  0.9863498702871967  for AdaBoost on validation set
Accuracy for number of predictors =  15  is  0.987956709066182  for AdaBoost
Accuracy for number of predictors =  15  is  0.9879795659478178  for AdaBoost on validation set
Accuracy for number of predictors =  20  is  0.9879864230123084  for AdaBoost
Accuracy for number of predictors =  20  is  0.9880709934743603  for AdaBoost on validation set
```

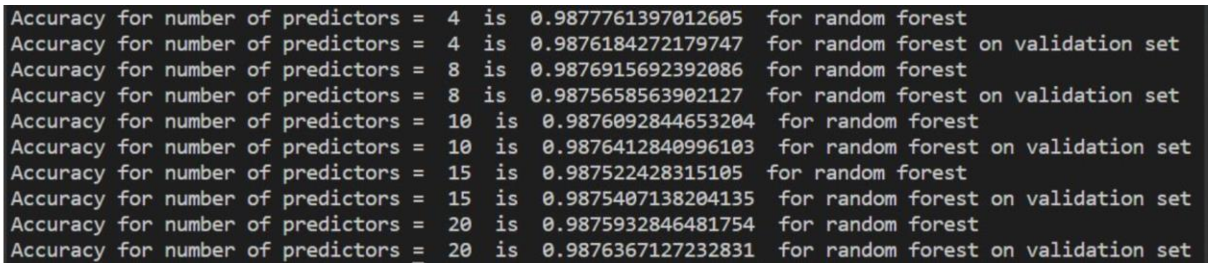
- We observe that almost all the models have an accuracy of almost 98.5% which is very good using the AdaBoost boosting technique.

- Now we run RF classifier using the below code



```python
def runBoostingRF(trainX,trainY,testX,testY,valX,valY):
    predictors = [4,8,10,15,20]
    i=0
    while i < len(predictors):
        dtc = RandomForestClassifier(random_state=2,n_estimators=predictors[i],max_depth=15,criterion="entropy")
        dtc.fit(trainX,trainY)
        yPred = dtc.predict(testX)
        yValPred = dtc.predict(valX)
        print("Accuracy for number of predictors = ",predictors[i]," is ",accuracy_score(testY,yPred)," for random forest")
        print("Accuracy for number of predictors = ",predictors[i]," is ",accuracy_score(valY,yValPred)," for random forest on
        validation set")
        i+=1
```

- The results obtained are as follows



```
Accuracy for number of predictors =  4  is  0.9877761397012605  for random forest
Accuracy for number of predictors =  4  is  0.9876184272179747  for random forest on validation set
Accuracy for number of predictors =  8  is  0.9876915692392086  for random forest
Accuracy for number of predictors =  8  is  0.9875658563902127  for random forest on validation set
Accuracy for number of predictors =  10  is  0.9876092844653204  for random forest
Accuracy for number of predictors =  10  is  0.9876412840996103  for random forest on validation set
Accuracy for number of predictors =  15  is  0.987522428315105  for random forest
Accuracy for number of predictors =  15  is  0.9875407138204135  for random forest on validation set
Accuracy for number of predictors =  20  is  0.9875932846481754  for random forest
Accuracy for number of predictors =  20  is  0.9876367127232831  for random forest on validation set
```

- Here also the obtained results have an accuracy greater than 98.5 for all the cases.

Comparison of the results

- We observe that both the models have very good results.
- We observe that both the models achieve the maximum accuracy for n=4 predictors.
- However, we observe that the maximum accuracy for Adaboost is 0.9889555547936595 while it is 0.9877761397012605 for random forest.
- Hence the accuracy of Adaboost is just little better than Random Forest classifier.