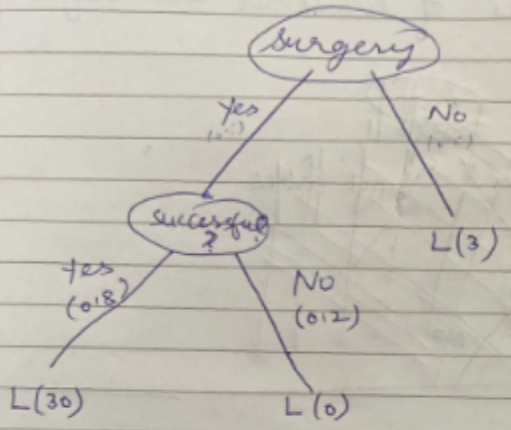
SECTION A

a) Decision trees

P(he doesn't survive after surgery) = 0.2



L(x) denotes the patient's living func^n with x representing the number of days he will survive

b) Given, L(0) = 0 and L(30) = 1, $y = x/30$
as when $x = 3$, $y = 0.1$

∴ P(Patient lives for 3 days) = 0.1

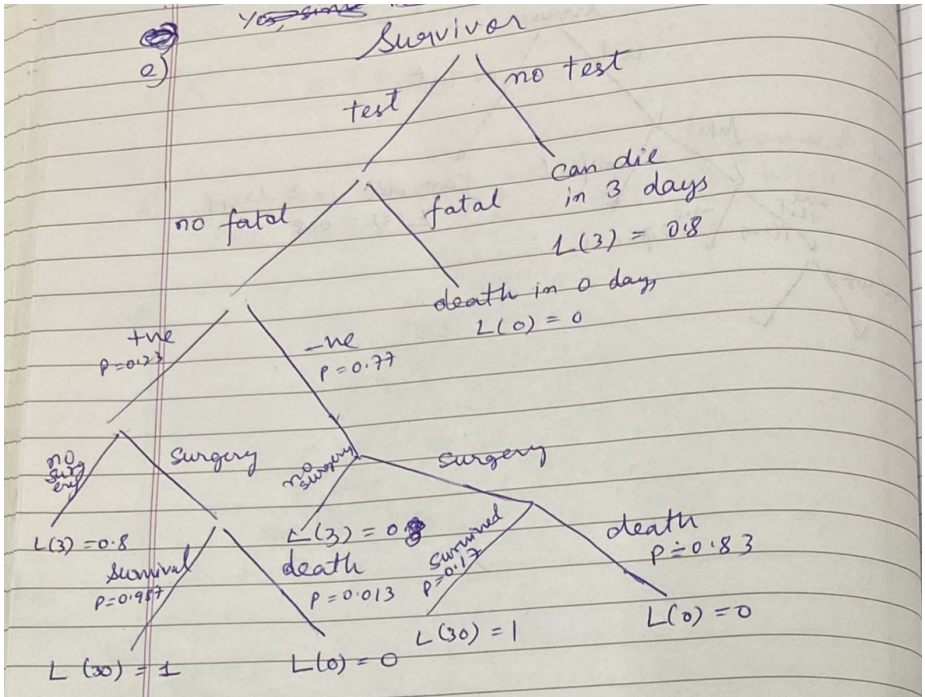
c) $$P(\text{survives} \mid \text{test +ve}) = \frac{P(\text{test +ve} \mid \text{survives}) * P(\text{survives})}{P(\text{test +ve} \mid \text{survive}) * P(\text{survives}) + P(\text{test +ve} \mid \text{not survive}) * P(\text{not survive})}$$

DATE: / /
PAGE:

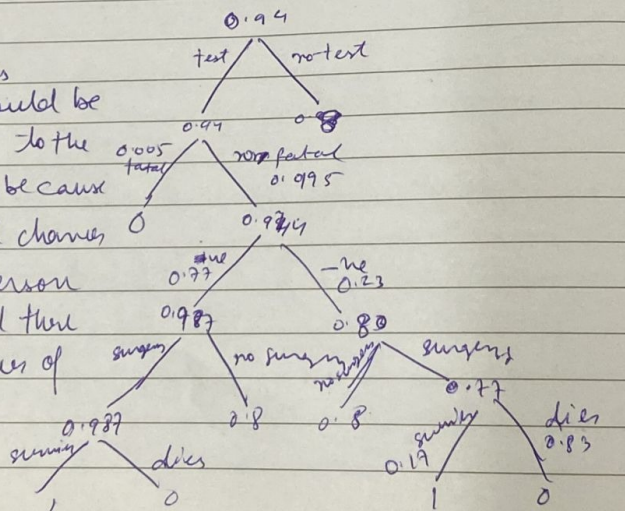
$$= \frac{0.95 \times 0.8}{0.95 \times 0.8 + 0.05 \times 0.2} = \frac{0.95 \times 0.8}{0.76 + 0.01}$$

$$= \frac{0.76}{0.77} = \frac{76}{77} = 0.99$$

d) Yes, the surgery should be performed as the chances of survival are high at 99%.



f) Yes the test should be taken prior to the operation because the survival chances of the person increase and there are less chances of him dying.



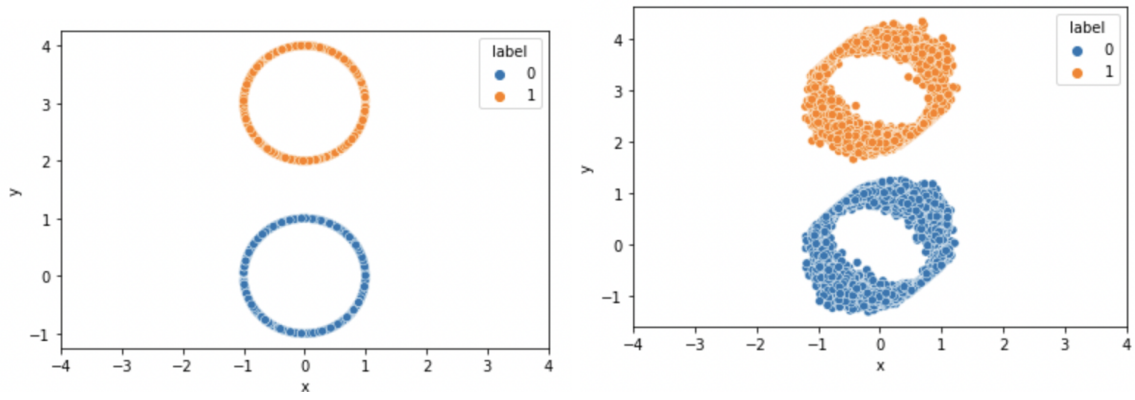
SECTION-B

A.

```
class dataset:
    def __init__(self,numofpoints):
        self.numofpoints = numofpoints
    def get (add_noise):
        xlist = []
        ylist = []
        label = []
        if add_noise == False:
            for i in range (10000):
                ch = random.choice([0, 1])
                label.append(ch)
                if(ch==0):
                    x = random.uniform(-1.0, 1.0)
                    xlist.append(x)
                    sign = random.choice([-1, 1])
                    y = sign*math.sqrt(1-(x**2))
                    ylist.append(y)
                if(ch==1):
                    x = random.uniform(-1.0, 1.0)
                    xlist.append(x)
                    sign = random.choice([-1, 1])
                    y = sign*math.sqrt(1-(x**2)) + 3
                    ylist.append(y)

        elif add_noise == True:
            noise = np.random.normal(0,0.1,10000)
            for i in range (10000):
                ch = random.choice([0, 1])
                label.append(ch)
                if(ch==0):
                    x = random.uniform(-1, 1)
                    xlist.append(x+noise[i])
                    sign = random.choice([-1, 1])
                    y = sign*math.sqrt(1-(x**2))
                    ylist.append(y+noise[i])

                if(ch==1):
```

I implemented a class named ‘dataset’ with all the relevant functions used to return the different points which were lying on the given equation of the circle. I also took care of the noise factor which was to be added and not added to the given dataset based on what was asked.

B. Scatterplot of created data frame without noise Scatterplot of the created data frame with noise



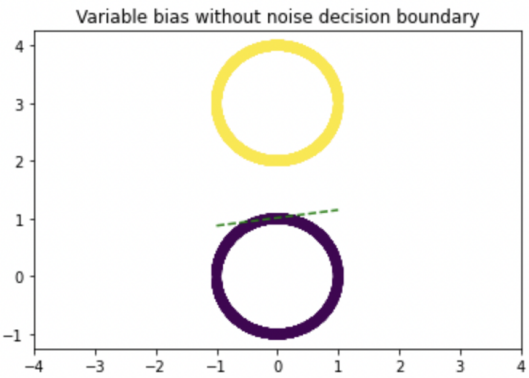
These came out to be in the form of a circle as they should have as the given equation used to create the data frame was the equation of a circle itself.

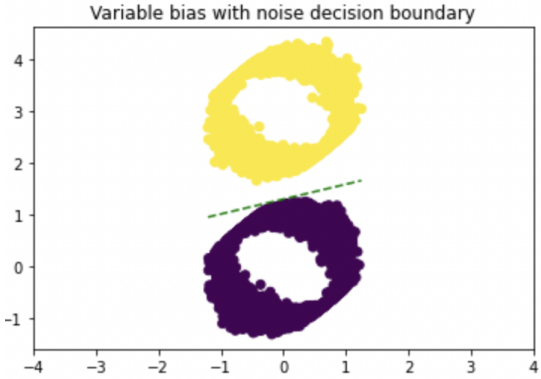
C.

```
import numpy as np
class Perceptron:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.lr = learning_rate
        self.epochs = epochs
        self.db = self.db
        self.wt = None
        self.bias = None

    def fit(self, X, y):
        n_samples, features = X.shape
        self.wt = np.zeros(features)
        self.bias = 0
        for _ in range(self.epochs):
            for idx, x_i in enumerate(X):
                linear = np.dot(x_i, self.wt) + self.bias
                y_predicted = self.db(linear)
                update = self.lr * (y[idx] - y_predicted)
                self.wt += update * x_i
                self.bias += update

    def fit2(self,X,y):
        n_samples, features = X.shape
        self.wt = np.zeros(features)
        self.bias = 0
        for _ in range(self.epochs):
            for idx, x_i in enumerate(X):
                linear = np.dot(x_i, self.wt) + 0 #bias set to 0
                y_predicted = self.db(linear)
                update = self.lr * (y[idx] - y_predicted)
                self.wt += update * x_i

    def db (self, x):
        return np.where(x >= 0, 1, 0)
```

I implemented the Perceptron Training Algorithm from scratch as shown in the picture above. I used the python constructor '__init__' to initialize the required variables and methods. The two functions 'fit' and

‘fit1’ are essentially the same functions, the only difference is that the latter has a fixed bias (equal to 0) and the former has a learnable bias. The db function is nothing but the step function which is used to set a threshold for the given inputs.

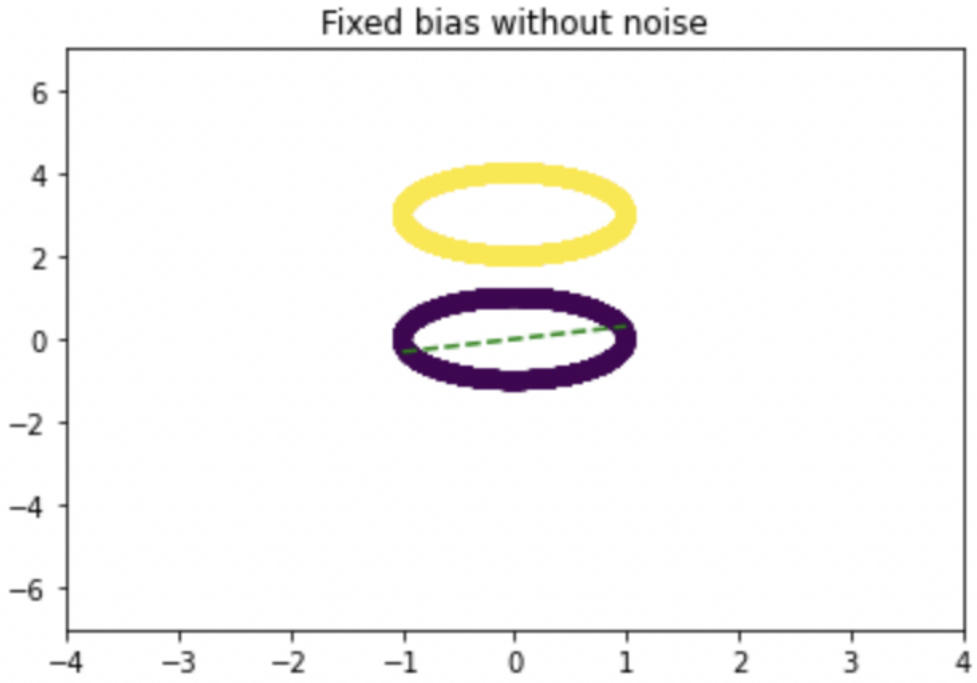




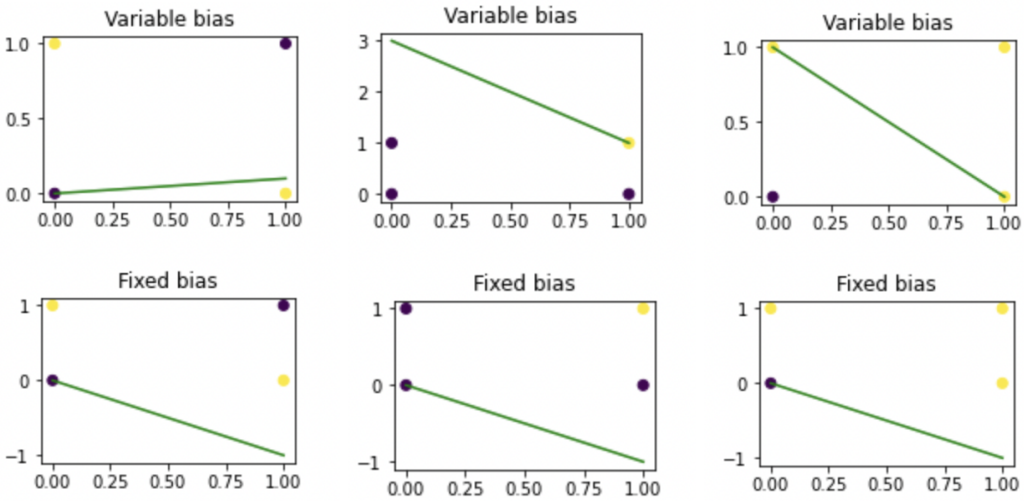
Given are the decision boundaries for both cases and we can see that they are linearly separable.

D.

On comparing the results with the C part we can easily conclude that keeping the bias fixed is clearly not a good practice here, as when we are keeping the bias to 0 we are clearly restricting the flexibility of the decision boundary and forcing it to pass through the origin which is, in fact, the centre of one of the two circles, hence it ends up intersecting the circle. Hence, it results in an incorrect decision boundary which is not the case in the C part where we were able to clearly plot a correct decision boundary. So, due to the constant term (bias) in the perceptron update equation, we are not able to change the y-intercept at 0, hence we can’t plot a valid decision boundary with a fixed bias.



E.



XOR GATE AND GATE OR GATE

We can conclude from the following plots that:
For XOR GATE - For both variable and fixed bias it is not able to draw a linearly separable decision boundary.
For AND GATE - When we fix the bias it is not able to plot a valid decision boundary but with learnable or variable bias it forms a valid decision boundary separating the two different classes.
For OR GATE - When we fix the bias it is not able to plot a valid decision boundary but with learnable or variable bias it forms a valid decision boundary separating the two different classes.

The results can be seen in the aforementioned pictures.

F.

Since we are given a hyperplane equation we can use that equation to predict the class. We insert any random point from any of the classes, we can find the polarity of the points to see where they lie wrt to the equation. We can check if the polarity comes out to be -ve or +ve and make our decision on the basis of that. We assume the classes to be strictly linearly separable for best results as we are dealing with a binary classification problem. A general hyperplane equation is , $a^T x = b$, which checks the $sign(a^T x - b)$ to classify the two different classes.

SECTION -C

A. Custom train test validation split implementation:

```
tr_size = 0.70
v_size=0.15
t_size = 0.15
df['label'] = temp_label
tr_idx = int(len(temp_label)*tr_size)
df_tr = df[0:tr_idx]
v_idx = int(len(temp_label)*v_size)
df_v = df[tr_idx:tr_idx+v_idx]
df_t = df[tr_idx+v_idx:]
t_idx = int(len(temp_label)*t_size)
X_train, y_train = df_tr.drop(columns='label'), df_tr['label']
X_valid, y_valid = df_v.drop(columns='label'), df_v['label']
X_test, y_test = df_t.drop(columns='label'), df_t['label']
print(X_train.shape), print(y_train.shape)
print(X_valid.shape), print(y_valid.shape)
print(X_test.shape), print(y_test.shape)
```

Null Value Handling - Even though there were no null values in the data, I made a provision to handle the null value case in a way that wherever we encounter a null value it gets replaced with the mode of the column, i.e. the most frequent value in that column.

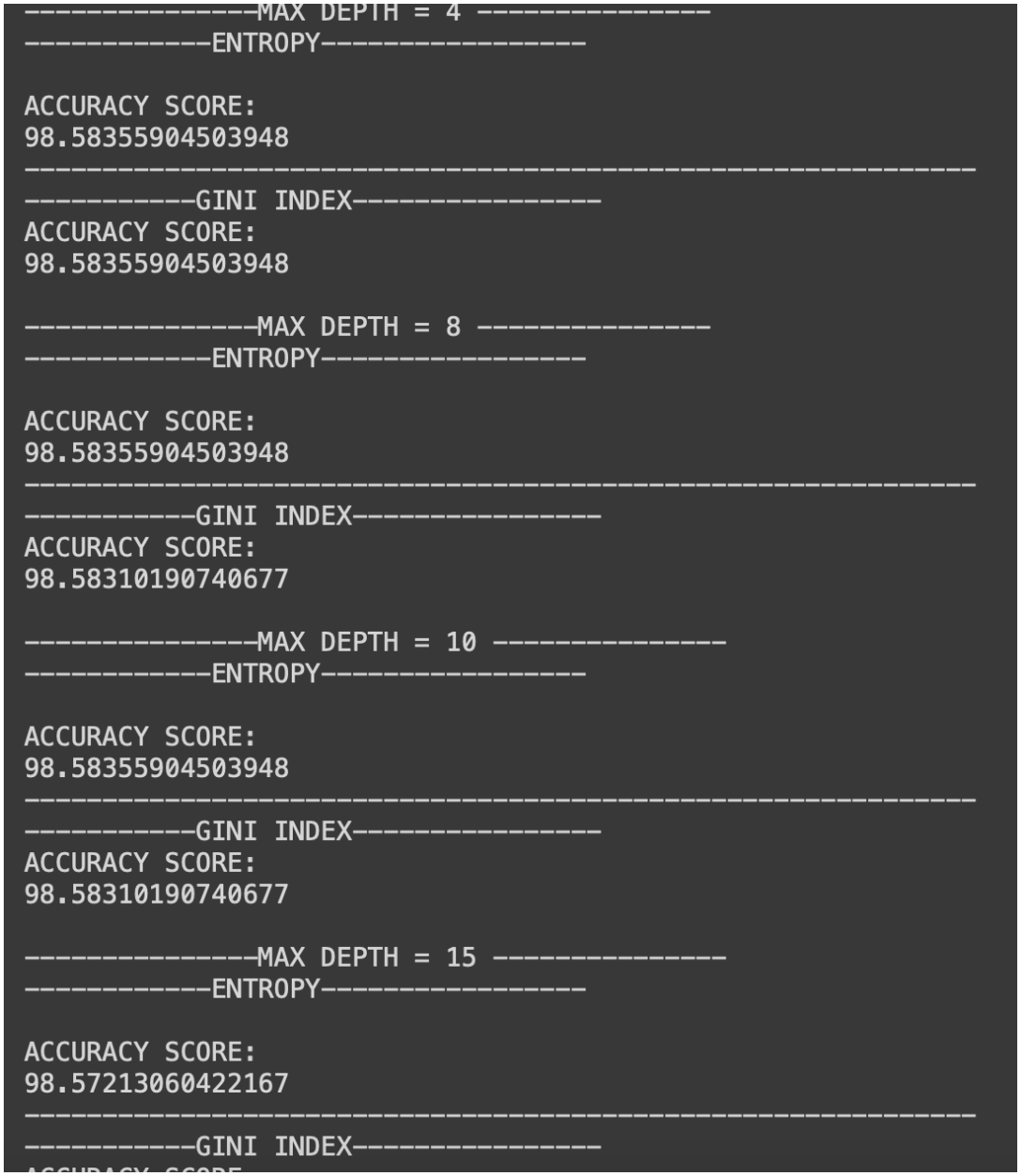
```
[5] df = df.drop(['address','label'], axis=1)
```

```
[6] df.head()
```

	year	day	length	weight	count	looped	neighbors	income
0	2017	11	18	0.008333	1	0	2	100050000.0
1	2016	132	44	0.000244	1	0	1	100000000.0
2	2016	246	0	1.000000	1	0	2	200000000.0
3	2016	322	72	0.003906	1	0	2	71200000.0
4	2016	238	144	0.072848	456	0	1	200000000.0

```
for column in df.columns:
    df[column].fillna(df[column].mode()[0], inplace=True)
    print(df[column].mode()[0])
```

```
2016
78
0
1.0
1
0
2
100000000.0
```



We tested the data using both the Entropy and Gini Index classifiers and found that even though both the methods gave good accuracies but as we kept increasing the max_depth we saw that **Gini Index** gave the best result out of the two methods.
The best accuracy was given by GINI INDEX at max depth = 15 in my case

B.

```
#2
print("^^^^^^^^^^^^^^^^^^^RandomForest^^^^^^^^^^^^^^^^^^")
clf = BaggingClassifier(n_estimators=100, max_samples=0.5 ,
clf.fit(train_x,train_y)
print("ACCURACY",clf.score(test_x,test_y))
print("-----------------------------------------------")
```

```
^^^^^^^^^^^^^^^^^^^RandomForest^^^^^^^^^^^^^^^^^^
ACCURACY 0.9858744471491755
-----------------------------------------------
```

The accuracy observed after ensembling was almost the same (just a little better in the case of RF) as observed in the previous part due to the data being highly skewed. Though theoretically we expect a better accuracy after ensembling since here we do sampling and take a forest of trees instead of just a single tree. A decision tree (part a) only combines some decisions whereas RF combines different trees, and then gives out the best output after the combination of decision results obtained from those decision trees. In theory this indeed is a much better method for greater accuracy but here because of the data being highly skewed the results came out to be almost the same.

C.

We used sklearn.ensemble to use adaboostclassifier. We tested for n-iterations =4,8,10,15,20 and kept our base estimator as 'gini' and max_depth at 15, since it gave out the best results in part (a).

```
#3
from sklearn.ensemble import AdaBoostClassifier
print("^^^^^^^^^^^^^^^^^^AdaBoostClassifier^^^^^^^^^^^^^^^^^^")
for i in {4,8,10,15,20}:
  clf = AdaBoostClassifier(n_estimators=i,base_estimator = DecisionTreeClassifier(criterion='gini',max_depth=15))

  model = clf.fit(train_x, train_y)
  print("------------N-ESTIMATORS: ",i,"--------------")
  print(clf.score(test_x,test_y)*100)
  print("-----------------------------------------------")
```

```
^^^^^^^^^^^^^^^^^^AdaBoostClassifier^^^^^^^^^^^^^^^^^^
------------N-ESTIMATORS:  4 --------------
97.6443697786311
-----------------------------------------------
------------N-ESTIMATORS:  8 --------------
95.36348156021074
-----------------------------------------------
------------N-ESTIMATORS:  10 --------------
95.02245688620702
-----------------------------------------------
------------N-ESTIMATORS:  15 --------------
95.71730608793042
-----------------------------------------------
------------N-ESTIMATORS:  20 --------------
97.69648346876035
-----------------------------------------------
```

Due to the highly skewed nature of the data we found out the accuracy here to be more or less the same here as well and found out the best accuracy was given by n_estimator = 20 in my case.

Theoretically it should give a better accuracy as it takes the mistake of previous stumps into account and then makes a new one, unlike the independent nature of all the trees in RF classification. Also, AdaBoost is more intolerant to overfitting as compared to RF.