Machine Learning CSE 343/543

Assignment-2 Report

Chaitanya Garg 2021248

Section - A

1.

a) Random Forests is an ensemble learning algorithm which trains many decision trees on different sets of data which are created by bootstrapping. Each decision tree in the random forest is trained on a random set of data(created by bootstrapping) and taking a random set of features.

Correlation refers to the similarities the predictions of 2 trees are. It is important as it helps in reducing the variance of the Random Forest.

Diversity refers to how different the predictions of 2 trees are. It is important as it helps in reducing the bias of the Random Forest by capturing various possibilities and diversities in data.

There is a trade-off associated with correlation and diversity in the random forest algorithm. If the trees are too correlated then, as there would be similarity in predictions, variance will be low while bias will shot up. On the other hand, if the trees are too diverse then, as there would be differences in predictions, variance will be high while bias will be low.

This can be handled by applying bootstrapping, max_depth, random feature selection in the RF algo.

b) The “curse of dimensionality” becomes an issue in Naive Bias Classification algorithm when the dataset on which NBC is being applied on has way too many features, or the number of features in the dataset are high. This happens due to the fact that Naive Bias makes a naive assumption that all the features are independent of each other, which is an unlikely case in high dimensional data. So, in reality when the features aren’t independent the model might have a low bias and high variance and is likely to overfit.

The following strategies can be applied to NBC:

Feature Regularization - Penalizes the model on having way too complex features, so in NBC, feature regularization can be applied by taking features which have unique features upto a threshold for classification algorithm.

Feature Selection - Trying to find the correlation between features and if found to be associated with them then, choose them.

Laplace Smoothing - Takes into consideration the non-zero probability by adding a +alpha sample in the numerator and a +alpha x num_of_uniqueValues in the denominator. Doing this, removes the possibility of encountering a probability 0.

c) Yes, the Naive Bias Classifier will have issues if it encounters values not encountered in the training data. If the feature was numerical then, the Guassian model will resolve the issue. However, the model gets stuck in a prick for categorical features.

   The problems faced include making wrong predictions, setting all probabilities to 0 so, unable to make a prediction, very small numerical probabilities which might slow down the model and even lead to the wrong prediction and increasing the variance of the model.

   Yes, this will affect the inference results.

   Some ways to encounter the problem are:

   Feature Selection - An appropriate choice of features reduces the chances of seeing unknown features, hence mitigating the problem.

   Ensemble Learning - Having NBC trained on bootstrapped data and choosing a random choice of features might address the issue.

   Data Augmentation - To synthetically create unseen data.

   Smoothing - In case, an unknown sample for a feature does arrive, smooth it by laplace smoothing.

d) Yes, while splitting the data, Information Gain tends to be more biased towards features with high numeral cardinality due to this reason the model might end up choosing less useful features just cuz they have a higher information gain.

A relevant example might be of having people dataset with countries and gender as attributes where we assume them to be equally likely. Suppose there are 100 different countries while only 2 different genders. Then, the information gained from countries is log2(100) while it is only 1 from genders.

An appropriate measure to handle the situation would be to use Gini Index or Chi-squared root.

Gini Index = 1- sigma(P(a)**2)

This is much more sensitive to change in cardinality and hence tackles the problem well.

Chi-sqaured root - is a testing algo that checks the independence of algorithms with each other and tackles the problem at hand.

Eg: 100 samples 50A 50B

According to div by some feature X left(30A,20B) right(20A,30B)
Gini(A) = 1 - 2*(0.5)^2 = 0.5
Gini(Aleft) = 1 - (0.6)^2 -(0.4)^2 = 1-0.36-0.16 = 0.48
Gini(Aright) = 1 - (0.6)^2 -(0.4)^2 = 1-0.36-0.16 = 0.48
IG = Gini(A) - ProbLeft*Gini(Aleft) - ProbRight*Gini(Aright)
IG = 0.5 - (0.5)*(0.96)
IG = 0.5-0.48 =0.02

2.

Rahul’s decision making process can be broken down as follows:
1.Checking the weather
2.If not raining counting the number of his friends playing
3.If count>7 ,then football else, badminton
4.If it is raining, then check if he can borrow TT racquet from his friend, if yes then play TT else pool.

a) Decision Tree of the Process

```
                         Check the Weather
                          /                \
                     Raining              Not Raining
                        /                    \
                       /                      \
            Check if can borrow          Count his friends
            TT Racquets from frnd             /    \
                 /        \                  /      \
                /          \                /        \
            Yes            No           Count>7    Else
              |             |               |         |
              |             |               |         |
            Play TT     Play Pool    Play Football  Play Badmtn
```

The possible outcomes are that he might end up playing TT, Pool, Football or Badminton depending on some conditions.

P(TT) = P(Raining) * P(Borrow Racquets|Raining)
P(Pool) = P(Raining) * P(Can not Borrow| Raining)
P(Football) = P(Not Raining) * P(Count>7 | Not Raining)
P(Badminton) = P(Not Raining) * P(Count<=7 | Not Raining)

b) P(app predicts rainy) = 0.3
P(app predicts clear) = 1-0.3=0.7
P(app predicts rainy|rainy) = 0.8
P(app predicts clear|rainy) =1- 0.8 =0.2
P(app predicts clear|clear) = 0.9
P(app predicts rainy|clear) =1- 0.9 =0.1
P(rainy) = p
P(clear) = 1-p

P(App predicts rainy) =P(app predicts rainy|rainy)*P(rainy) +
P(app predicts rainy|clear) * P(clear)
= (0.8)p + (0.1)*(1-p)
= 0.1 + 0.7p
0.3 = 0.1 + 0.7p
0.2 = 0.7p
p=2/7
P(Rainy|App predicts rainy) = P(app predicts rainy|rainy)*P(rainy)/P(App predicts rainy)
= ((0.8)*p)/(0.1 + 0.7p)
= ((0.8)*0.28)/(0.3)
= 0.762

c)
```
                          See Mood
                         /        \
                        /          \
                 Good                 Bad
                /    \               /    \
               /      \             /      \
          Gym      No Gym      Gym       No Gym
         /    \               /              \
   Cardio      Weight    Cardio          Weight
```

P(Gym|Good Mood) = 0.8
P(No Gym|Good Mood) = 0.2
P(Gym|Bad Mood) = 0.4
P(No Gym|Bad Mood) = 0.6
P(Cardio|Gym)=0.5
P(Weight|Gym) =0.5
P(Good Mood)=p
P(Bad Mood) = 1-p
P(Gym) = P(Gym|Good Mood)*P(Good Mood)+P(Gym|Bad Mood)*P(Bad Mood)
=0.8p+0.4(1-p)
P(No Gym) = P(No Gym|Good Mood)*P(Good Mood)+P(No Gym|Bad Mood)*P(Bad Mood)
=0.2p+0.6(1-p)
P(weight) = P(weight|Gym) * P(Gym)
= 0.5 * (0.8p+0.4(1-p))
= 0.4p + 0.2(1-p)
P(cardio) = P(cardio|Gym) * P(Gym)
= 0.5 * (0.8p+0.4(1-p))
= 0.4p + 0.2(1-p)

d) P(Good Mood)= 0.6
P(Bad Mood) =0.4
P(F=7|Good Mood) = 0.7
P(F=7|Bad Mood)=0.45
P(F=7) = P(F=7|Good Mood)*P(Good Mood) + P(F=7|Bad Mood)*P(Bad Mood)
P(F=7) = 0.7*0.6 + 0.45*0.4
= 0.42 + 0.18
=0.6

P(Good Mood|F=7) = P(F=7|Good Mood)*P(Good Mood)/P(F=7)
= 0.7*0.6/0.6
=0.7 => p=0.7
P(Bad Mood|F=7) = P(F=7|Bad Mood)*P(Bad Mood)/P(F=7)
= (0.45)*(0.4)/(0.6)
=0.3

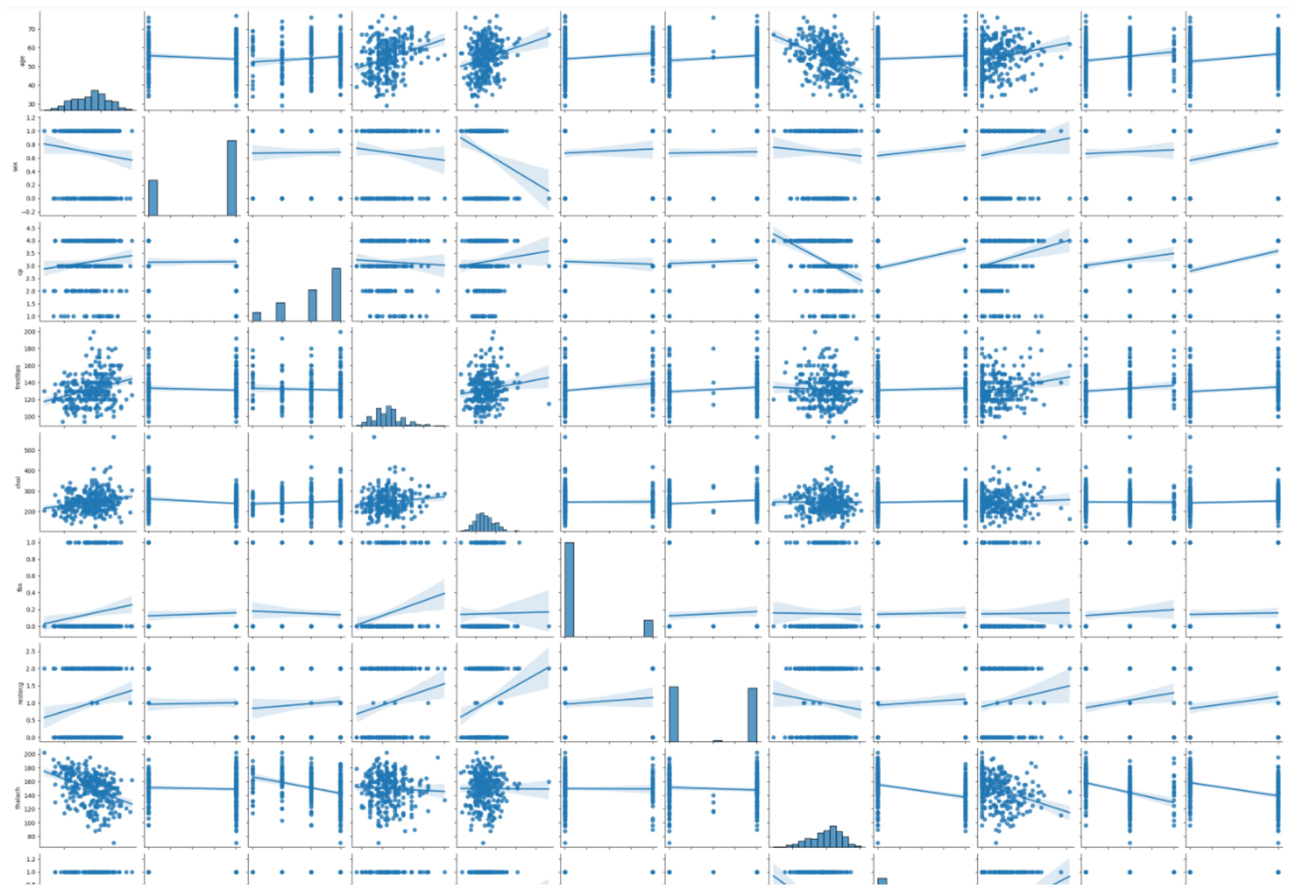
P(No Gym) = P(No Gym|Good Mood)*P(Good Mood)+P(No Gym|Bad Mood)*P(Bad Mood)
=0.2p+0.6(1-p)
=0.14+0.18
=0.32
P(weight) = P(weight|Gym) * P(Gym)
= 0.5 * (0.8p+0.4(1-p))
= 0.4p + 0.2(1-p)
=0.28 + 0.06
=0.34
P(cardio) = P(cardio|Gym) * P(Gym)
= 0.5 * (0.8p+0.4(1-p))
= 0.4p + 0.2(1-p)
=0.28 + 0.06
=0.34

Section - B

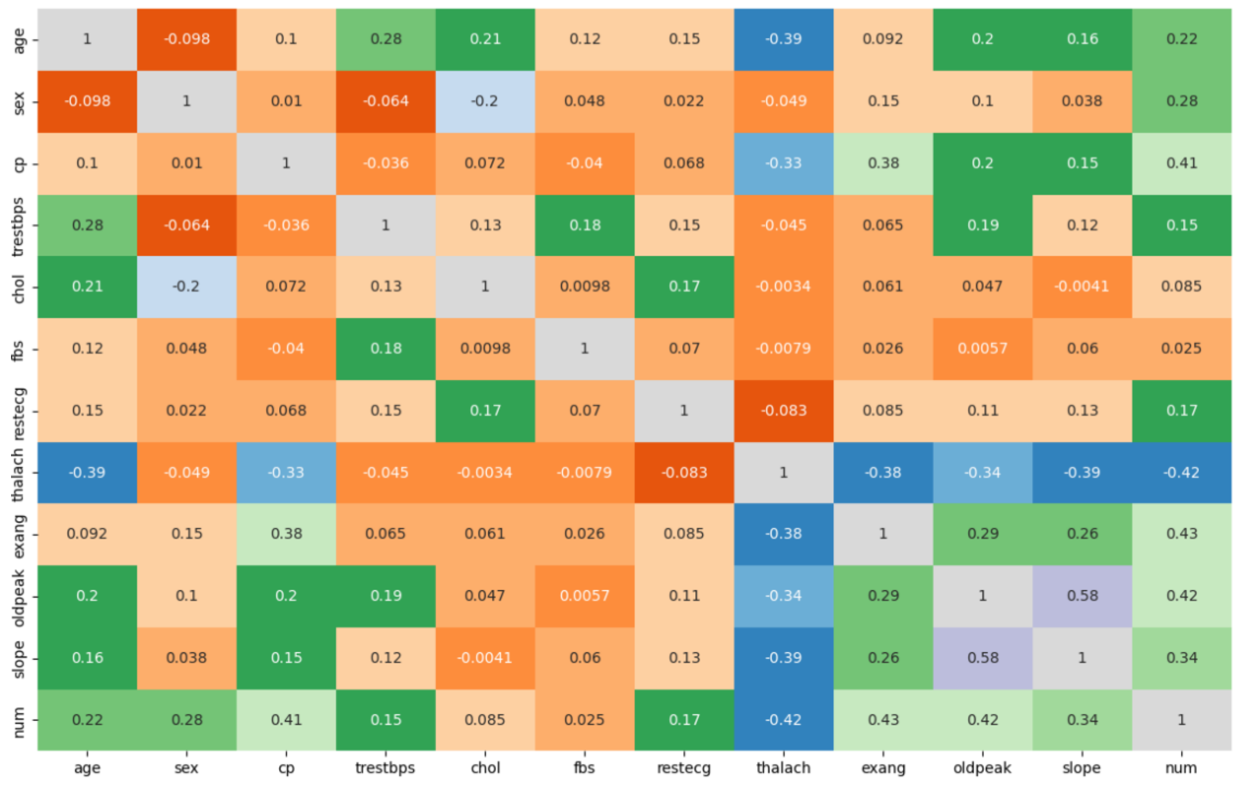
2.

a)

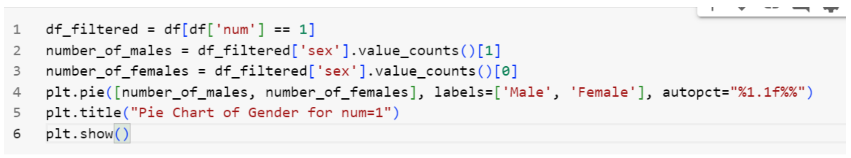
PairPlot



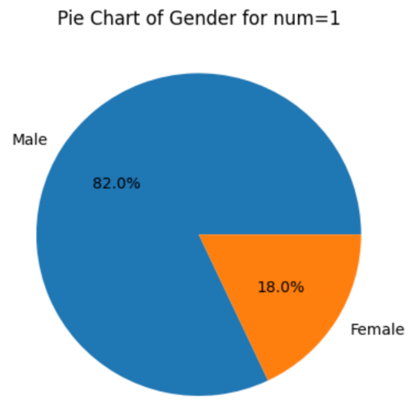
HeatMap



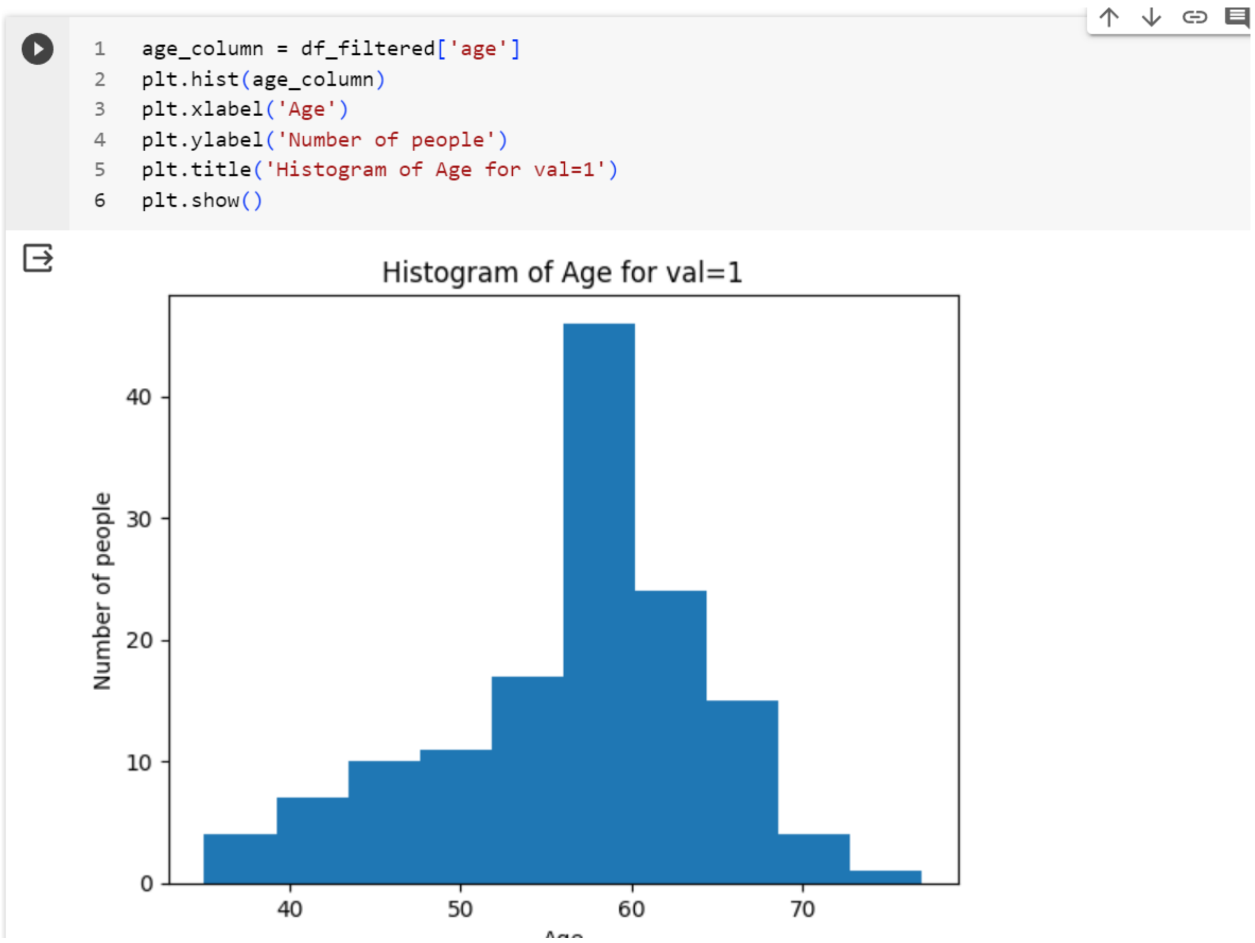
PieChart



```
df_filtered = df[df['num'] == 1]
number_of_males = df_filtered['sex'].value_counts()[1]
number_of_females = df_filtered['sex'].value_counts()[0]
plt.pie([number_of_males, number_of_females], labels=['Male', 'Female'], autopct="%1.1f%%")
plt.title("Pie Chart of Gender for num=1")
plt.show()
```



Histogram



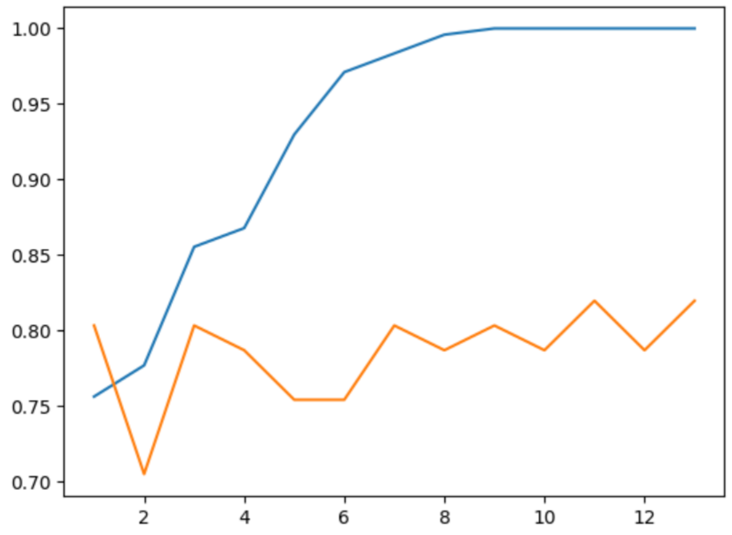
```
age_column = df_filtered['age']
plt.hist(age_column)
plt.xlabel('Age')
plt.ylabel('Number of people')
plt.title('Histogram of Age for val=1')
plt.show()
```

b) Splitting Train-Test

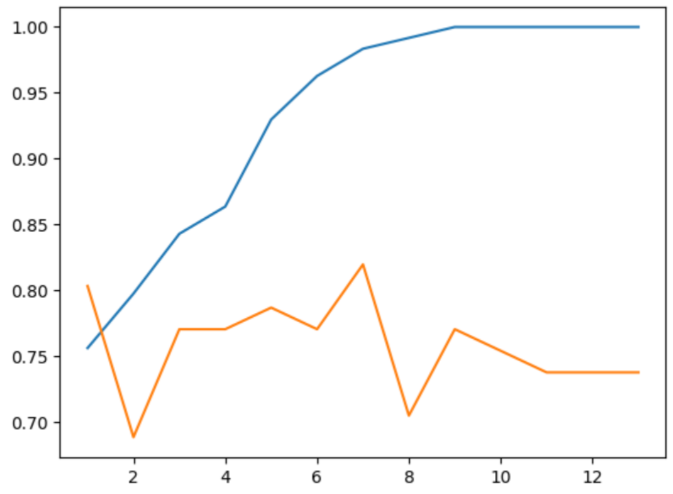
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2) #Splitting in 80:20
```

c) Entropy and Gini

Entropy Best(0.81967)



Gini Index(0.81967)



Best Fit Handling

```
if(max(AccuTestG)>max(AccuTestEnt)):
  BESTFIT='gini'
  BESTDEPTH = np.argmax(AccuTestG)+1
  BESTACCU = max(AccuTestG)
else:
  BESTFIT='entropy'
  BESTDEPTH = np.argmax(AccuTestEnt)+1
  BESTACCU = max(AccuTestEnt)

print("Best Fit Model is found to be",BESTFIT,"with accuracy",BESTACCU,"and has a depth of",BESTDEPTH)
```

d) Grid Search CV

Parameter Grid

```
#HyperParameter Intialising
Param_Grid = {'min_samples_split': [2, 5,8, 10,12,15,20],
              'max_features': ['auto', 'sqrt', 'log2']}
```

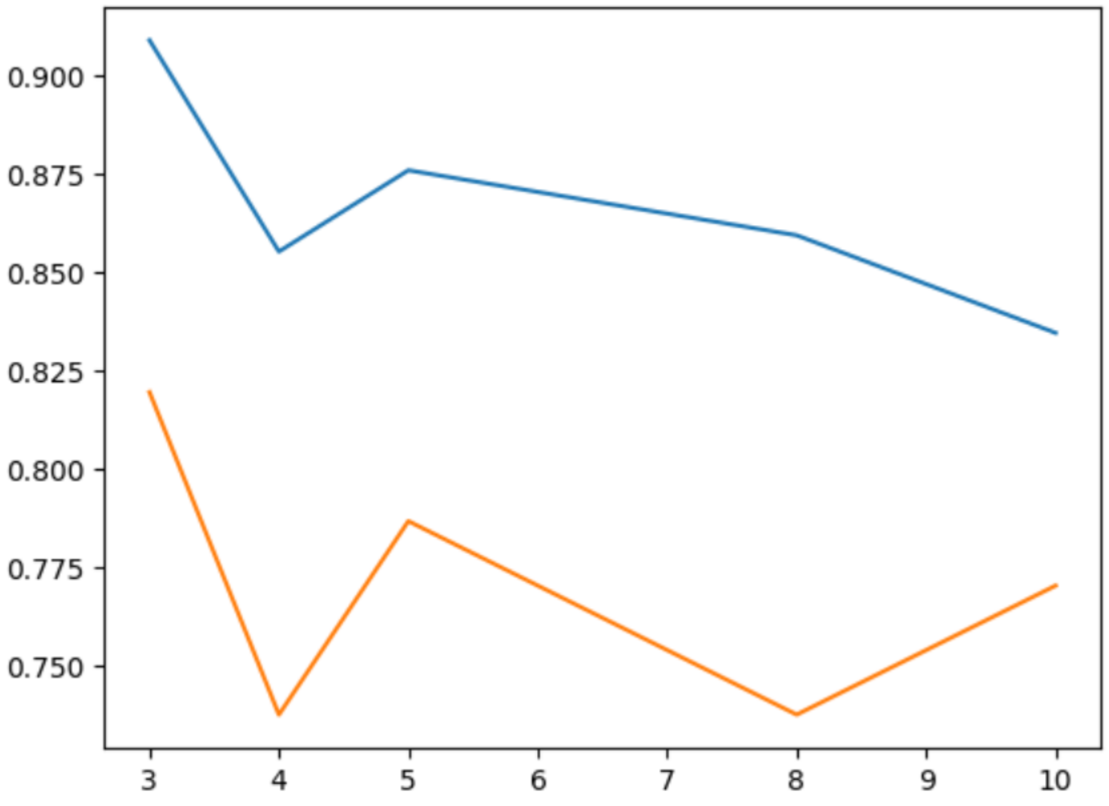
Applying Grid Search CV for a variety of KFold Values

```
folds = [3,4,5,8,10]
foldTest = []
AccuTest=[]
AccuTrain=[]
for i in folds:
  print("Checking for CV-fold",i)
  GridSearchclf = GridSearchCV(estimator=DecisionTreeClassifier(criterion=BESTFIT),
                               param_grid=Param_Grid,
                               cv=i,
                               n_jobs=-1)
  GridSearchclf.fit(X_train, y_train)
  print(f"Best Parameters obtained are : {GridSearchclf.best_params_} for cv{i}")
  y_pred = GridSearchclf.predict(X_train)
  accuracy = accuracy_score(y_train, y_pred)
  AccuTrain.append(accuracy)
  y_pred = GridSearchclf.predict(X_test)
  accuracy = accuracy_score(y_test, y_pred)
  AccuTest.append(accuracy)
  print('Accuracy:', accuracy)
plt.plot(folds,AccuTrain,folds,AccuTest)
plt.show()
```

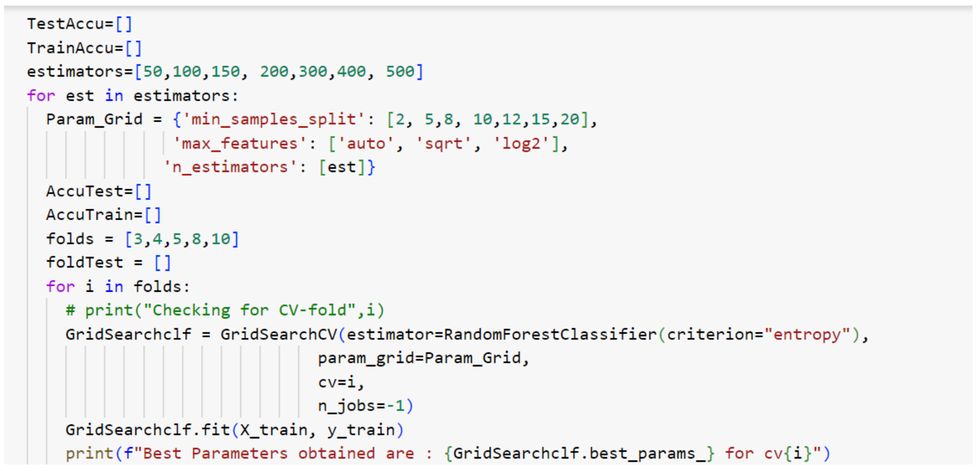
Best Fit Obtained: for K=3

```
Checking for CV-fold 3
Best Parameters obtained are : {'max_features': 'sqrt', 'min_samples_split': 10} for cv3
Accuracy: 0.819672131147541
```

Metric Graph With K changing

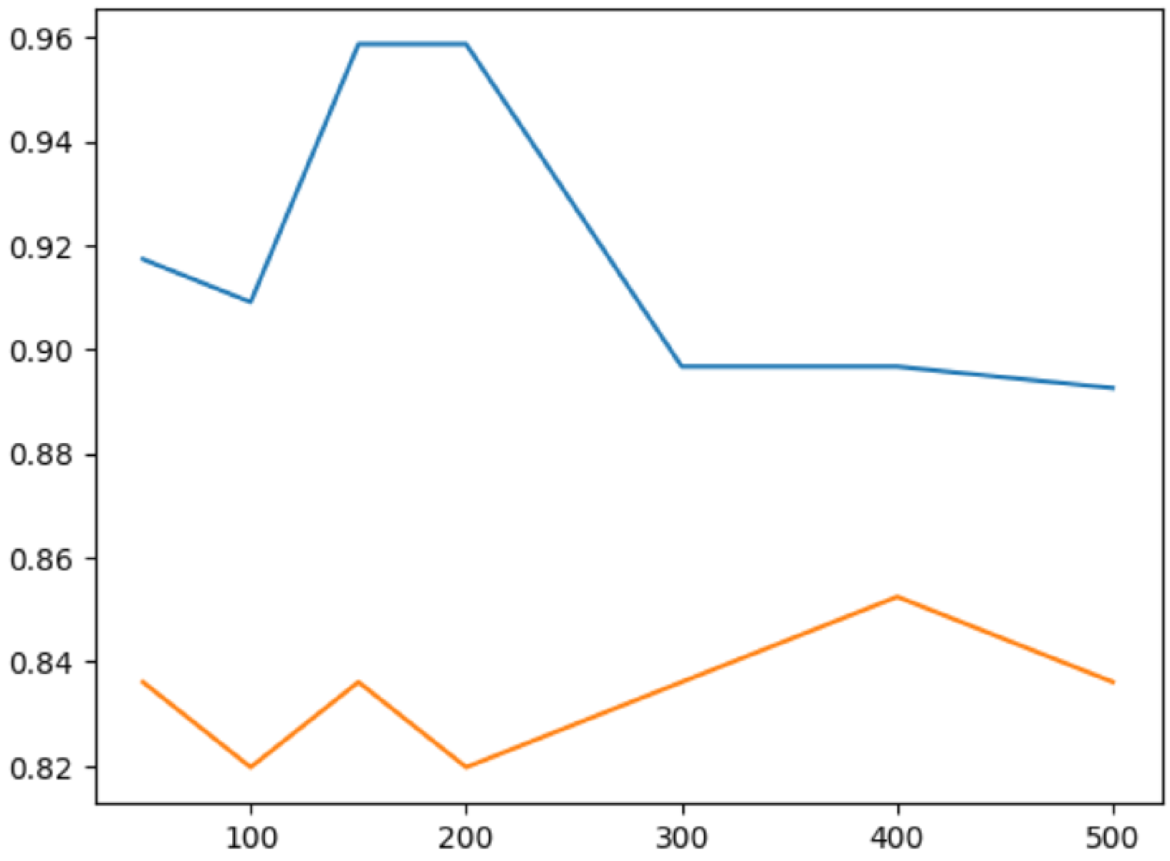


e) Random Forest Application



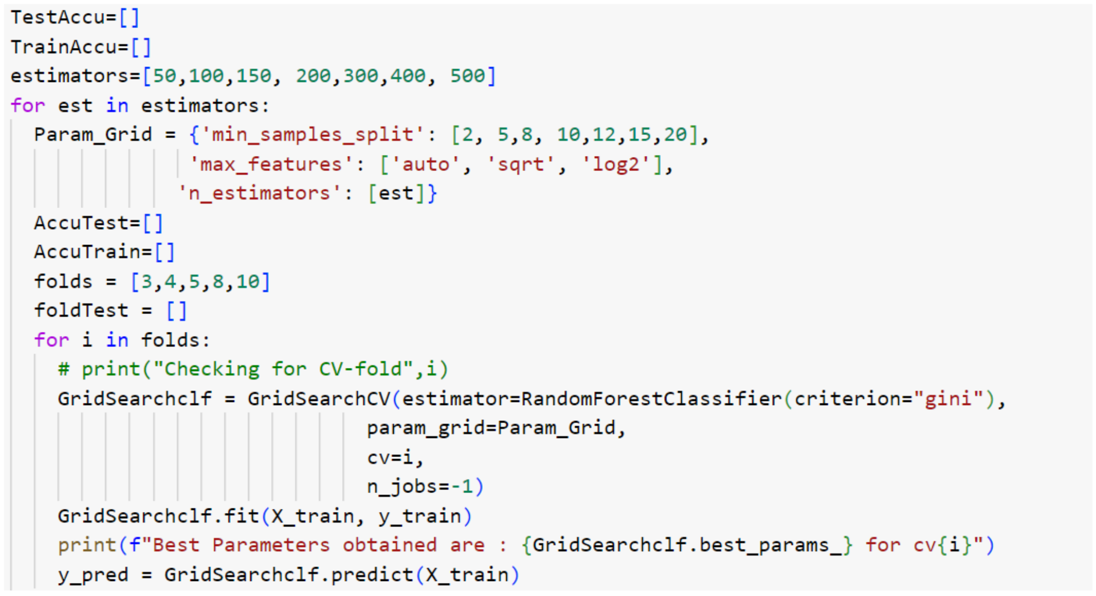
```
TestAccu=[]
TrainAccu=[]
estimators=[50,100,150, 200,300,400, 500]
for est in estimators:
  Param_Grid = {'min_samples_split': [2, 5,8, 10,12,15,20],
                'max_features': ['auto', 'sqrt', 'log2'],
               'n_estimators': [est]}
  AccuTest=[]
  AccuTrain=[]
  folds = [3,4,5,8,10]
  foldTest = []
  for i in folds:
    # print("Checking for CV-fold",i)
    GridSearchclf = GridSearchCV(estimator=RandomForestClassifier(criterion="entropy"),
                                 param_grid=Param_Grid,
                                 cv=i,
                                 n_jobs=-1)
    GridSearchclf.fit(X_train, y_train)
    print(f"Best Parameters obtained are : {GridSearchclf.best_params_} for cv{i}")
```

Graph for Varying Number of Trees:



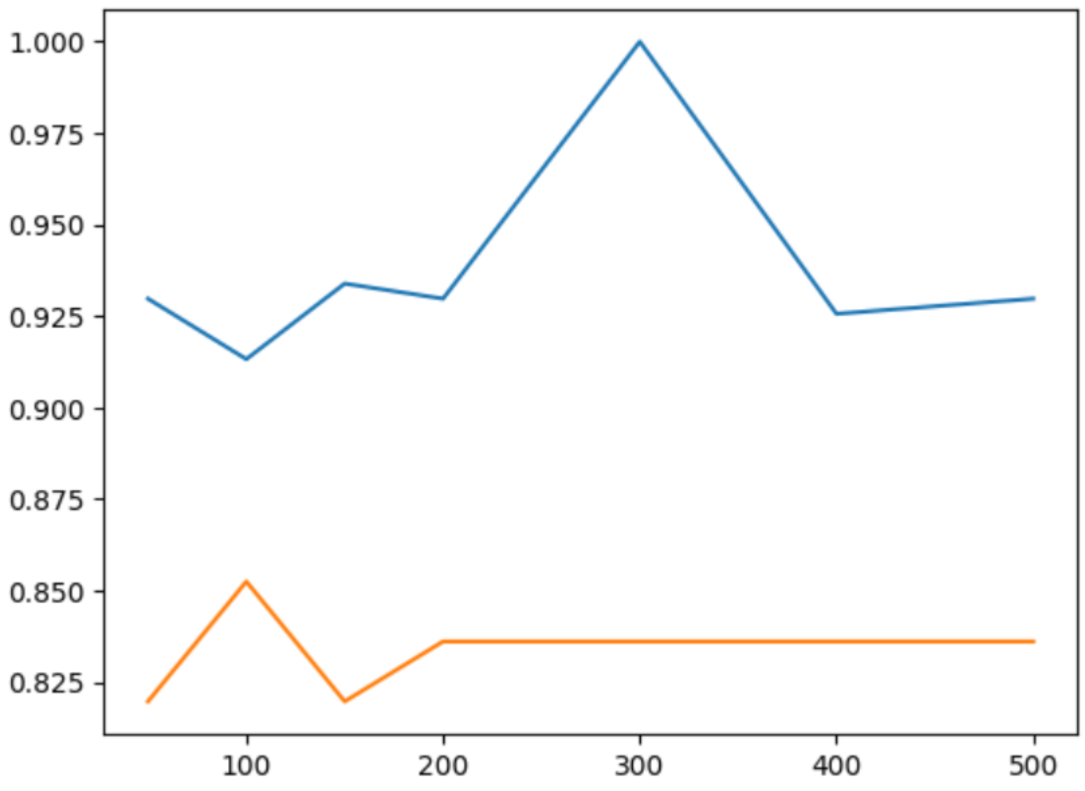
BestFit for Entropy:

```
Best Parameters obtained are : {'max_features': 'auto', 'min_samples_split': 20, 'n_estimators': 400} for cv1
Accuracy: 0.8524590163934426
```

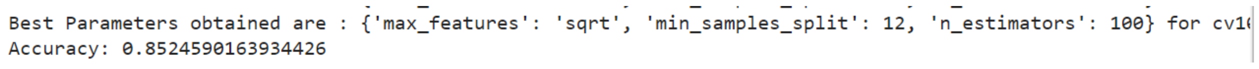


```
TestAccu=[]
TrainAccu=[]
estimators=[50,100,150, 200,300,400, 500]
for est in estimators:
  Param_Grid = {'min_samples_split': [2, 5,8, 10,12,15,20],
                'max_features': ['auto', 'sqrt', 'log2'],
               'n_estimators': [est]}
  AccuTest=[]
  AccuTrain=[]
  folds = [3,4,5,8,10]
  foldTest = []
  for i in folds:
    # print("Checking for CV-fold",i)
    GridSearchclf = GridSearchCV(estimator=RandomForestClassifier(criterion="gini"),
                                 param_grid=Param_Grid,
                                 cv=i,
                                 n_jobs=-1)
    GridSearchclf.fit(X_train, y_train)
    print(f"Best Parameters obtained are : {GridSearchclf.best_params_} for cv{i}")
    y_pred = GridSearchclf.predict(X_train)
```

Graph for Varying Number of Trees:

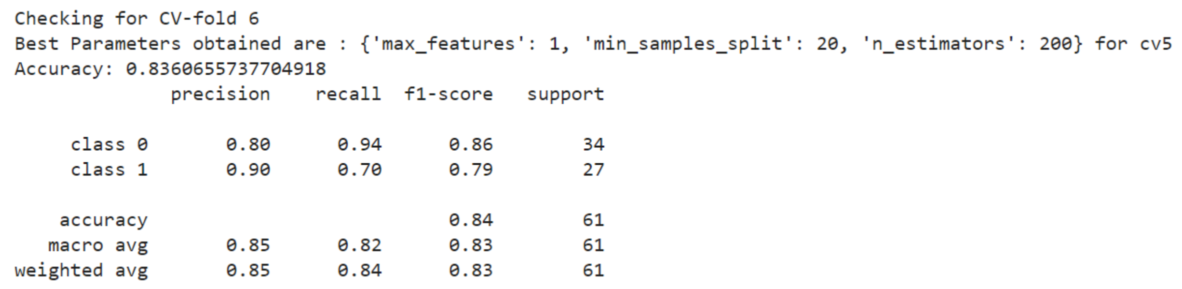


Gini Model Best Fit:



```
Best Parameters obtained are : {'max_features': 'sqrt', 'min_samples_split': 12, 'n_estimators': 100} for cv1
Accuracy: 0.8524590163934426
```

Class Report



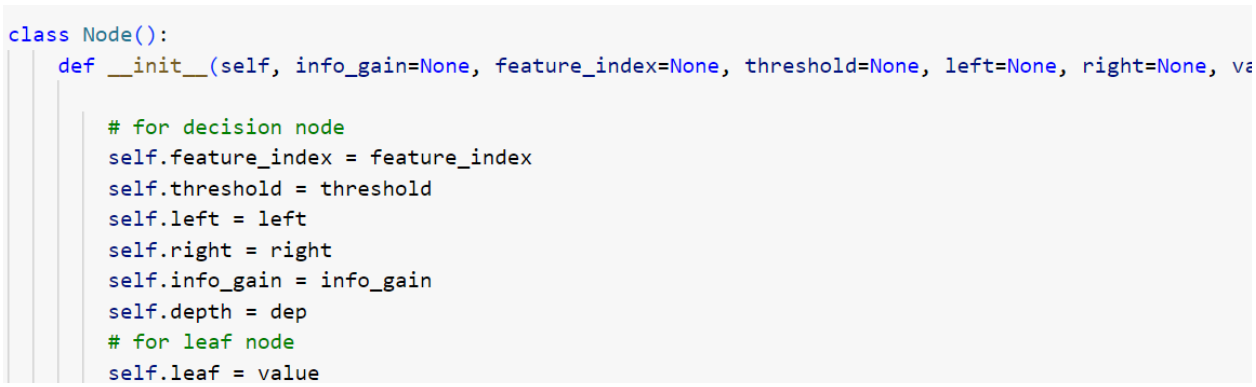
```
Checking for CV-fold 6
Best Parameters obtained are : {'max_features': 1, 'min_samples_split': 20, 'n_estimators': 200} for cv5
Accuracy: 0.8360655737704918
              precision    recall  f1-score   support

     class 0       0.80      0.94      0.86        34
     class 1       0.90      0.70      0.79        27

    accuracy                           0.84        61
   macro avg       0.85      0.82      0.83        61
weighted avg       0.85      0.84      0.83        61
```

Section - C

1.

a)Self Implementation

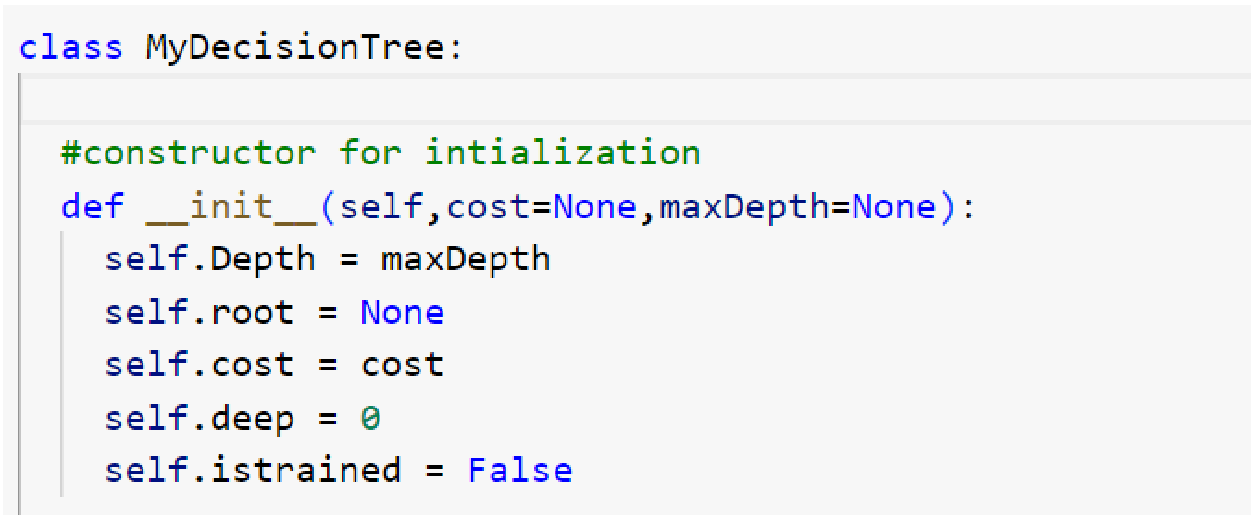
The purpose was to implement a decision tree using a class. Like all tree algorithms it needs a node class to take care of the splits and its relevant data.



```
class Node():
    def __init__(self, info_gain=None, feature_index=None, threshold=None, left=None, right=None, va

        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        self.depth = dep
        # for leaf node
        self.leaf = value
```

The Node Class Manages the following Data, which feature was chosen for the split, whats its left and right children, what was the associated info gain, at what depth was it located and if it is a leaf then the corresponding value handled by it.

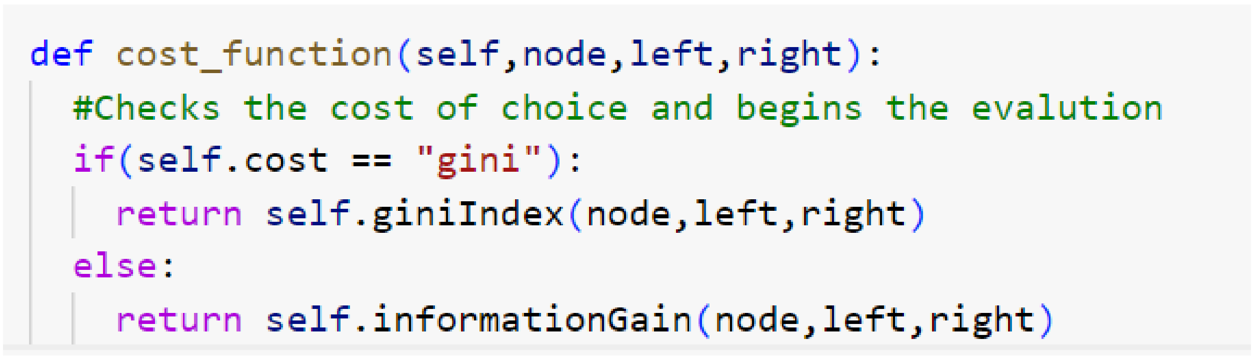
Constructor():



```
class MyDecisionTree:

    #constructor for intialization
    def __init__(self,cost=None,maxDepth=None):
        self.Depth = maxDepth
        self.root = None
        self.cost = cost
        self.deep = 0
        self.istrained = False
```

Self.Depth holds the value of the max tree depth
Self.root holds the root node of the tree
Self.cost keeps track of the choice of cost function
Self.deep keeps track of the tree’s depth
Self.istrained tracks if the model has been trained or not

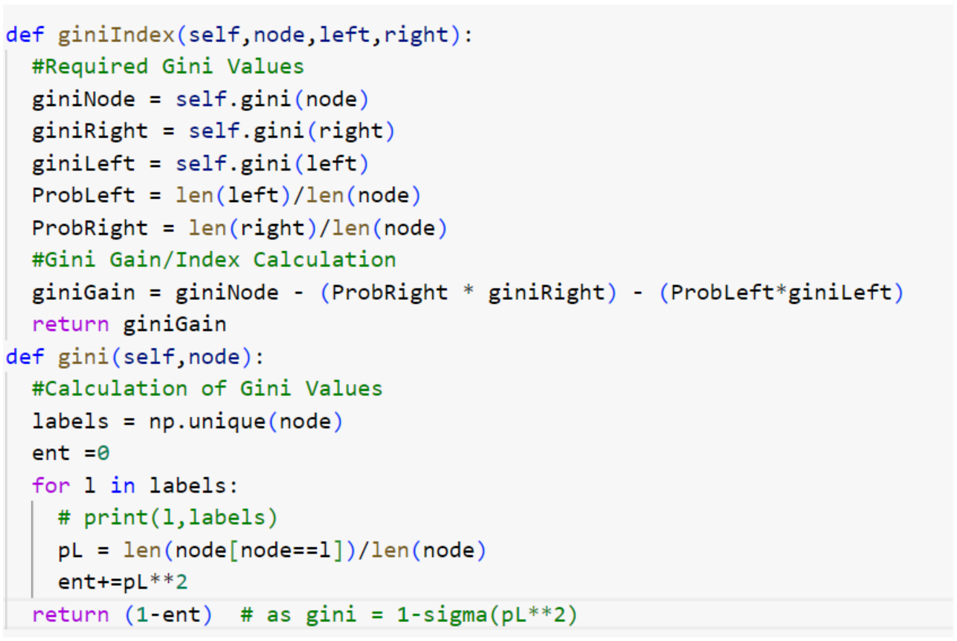
a)cost_function()



```
def cost_function(self,node,left,right):
  #Checks the cost of choice and begins the evalution
  if(self.cost == "gini"):
    return self.giniIndex(node,left,right)
  else:
    return self.informationGain(node,left,right)
```

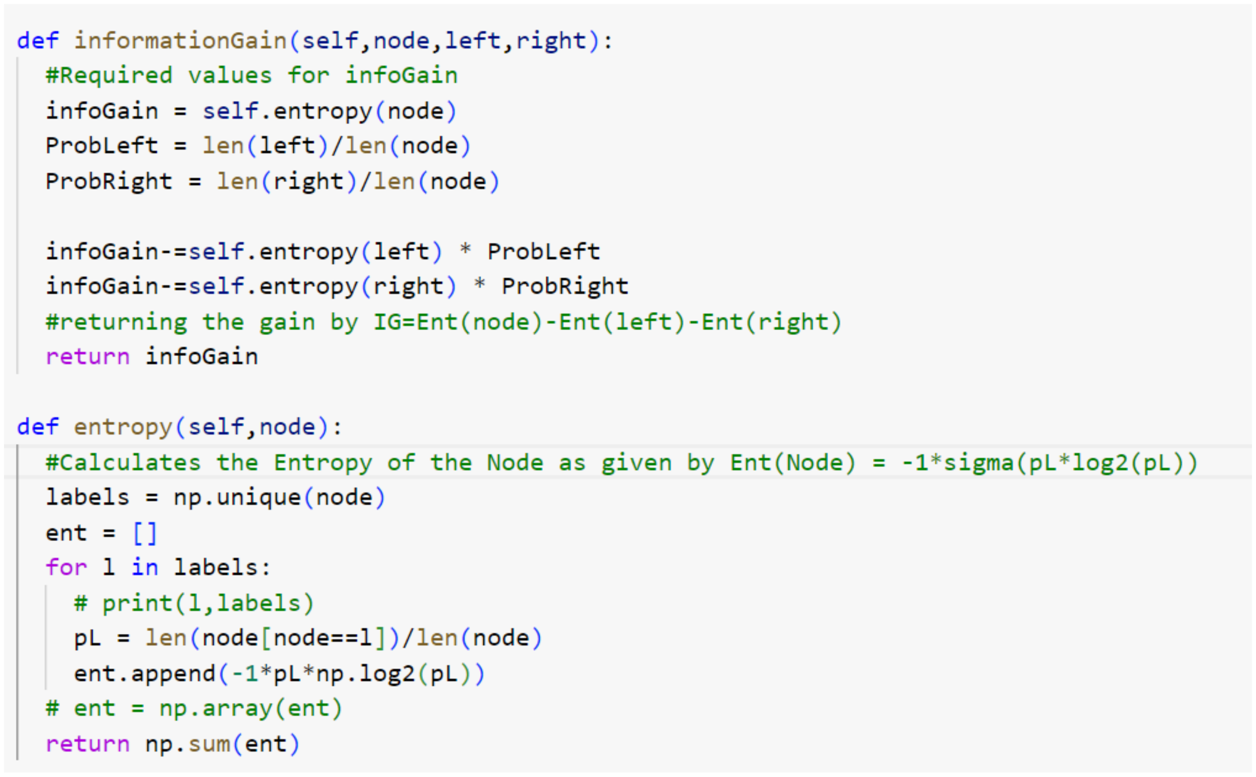
By default, entropy is the costFunction of choice

For Gini Index the Following is Used



```
def giniIndex(self,node,left,right):
  #Required Gini Values
  giniNode = self.gini(node)
  giniRight = self.gini(right)
  giniLeft = self.gini(left)
  ProbLeft = len(left)/len(node)
  ProbRight = len(right)/len(node)
  #Gini Gain/Index Calculation
  giniGain = giniNode - (ProbRight * giniRight) - (ProbLeft*giniLeft)
  return giniGain
def gini(self,node):
  #Calculation of Gini Values
  labels = np.unique(node)
  ent =0
  for l in labels:
    # print(l,labels)
    pL = len(node[node==l])/len(node)
    ent+=pL**2
  return (1-ent)  # as gini = 1-sigma(pL**2)
```

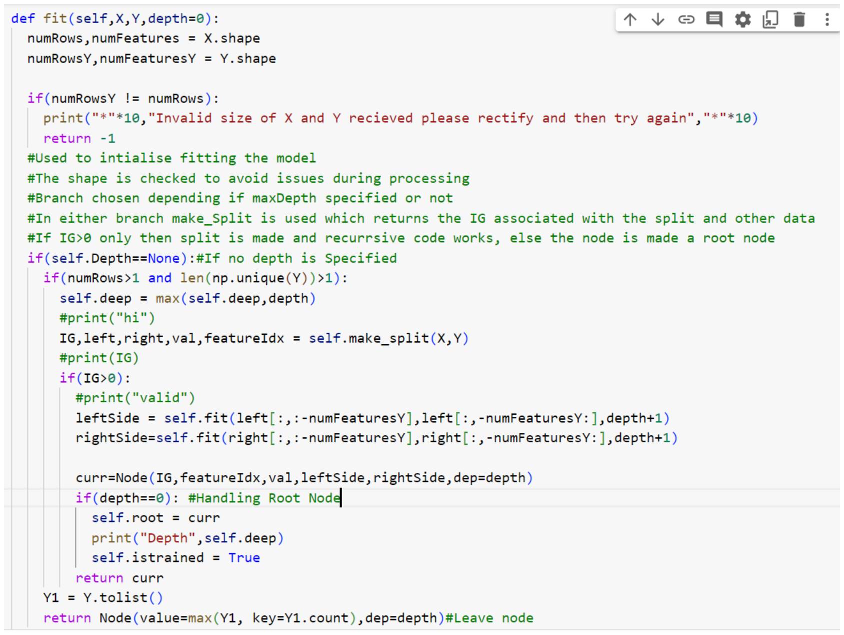
For entropy the following function is enlisted,



```python
def informationGain(self,node,left,right):
  #Required values for infoGain
  infoGain = self.entropy(node)
  ProbLeft = len(left)/len(node)
  ProbRight = len(right)/len(node)

  infoGain-=self.entropy(left) * ProbLeft
  infoGain-=self.entropy(right) * ProbRight
  #returning the gain by IG=Ent(node)-Ent(left)-Ent(right)
  return infoGain

def entropy(self,node):
  #Calculates the Entropy of the Node as given by Ent(Node) = -1*sigma(pL*log2(pL))
  labels = np.unique(node)
  ent = []
  for l in labels:
    # print(l,labels)
    pL = len(node[node==l])/len(node)
    ent.append(-1*pL*np.log2(pL))
  # ent = np.array(ent)
  return np.sum(ent)
```

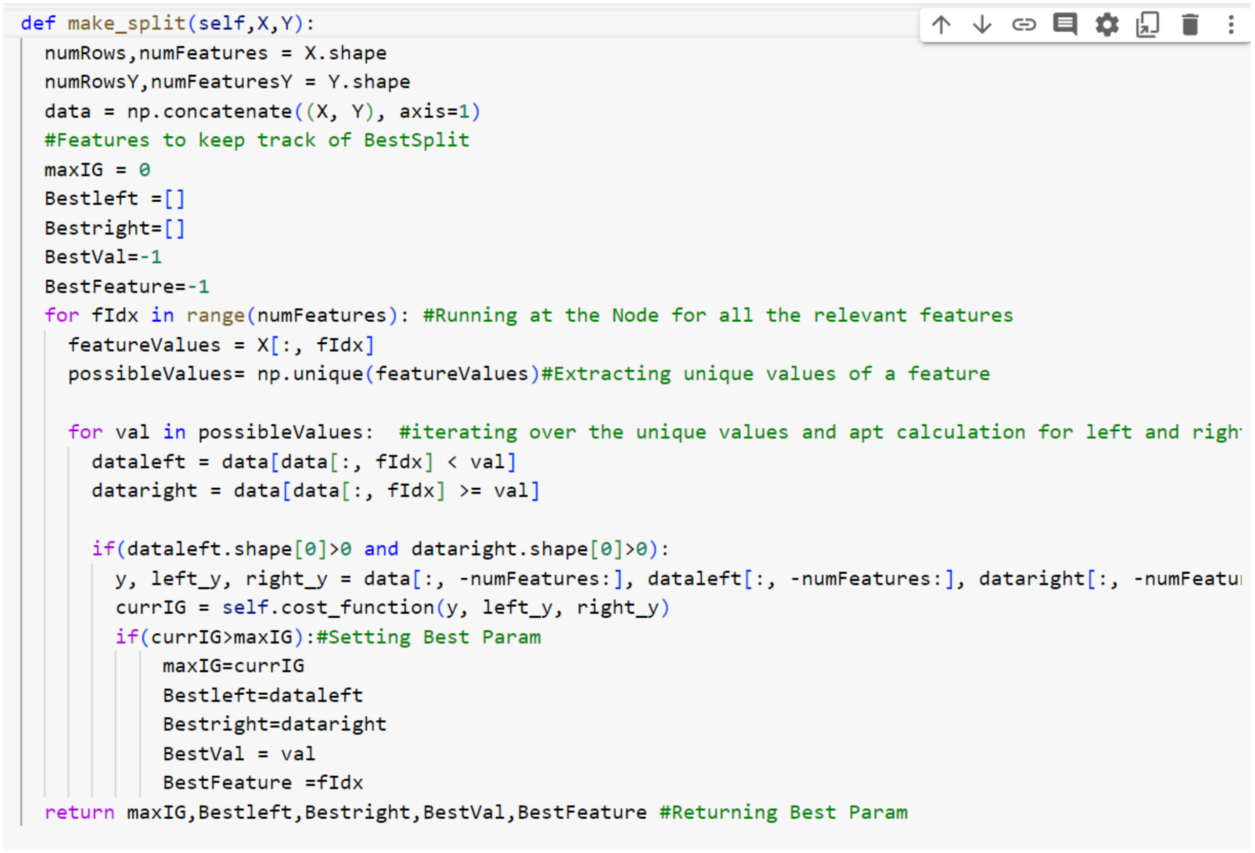
Both of the function calls is handled by cost function

Fit Function



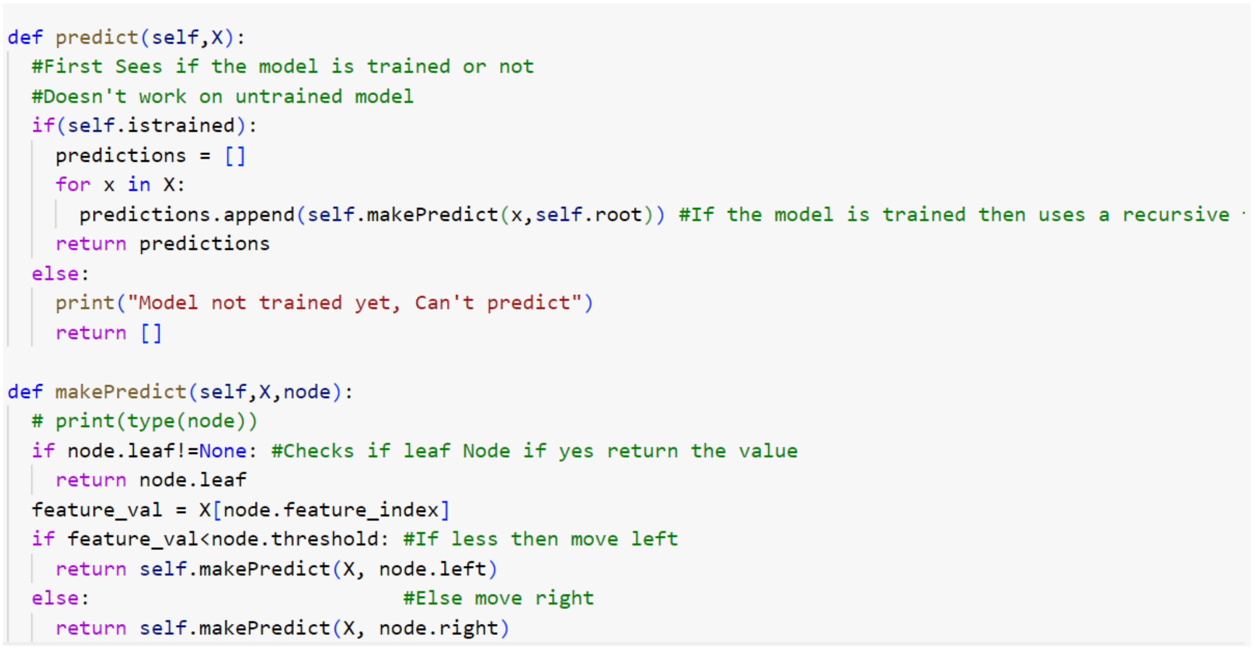
```python
def fit(self,X,Y,depth=0):
  numRows,numFeatures = X.shape
  numRowsY,numFeaturesY = Y.shape

  if(numRowsY != numRows):
    print("*"*10,"Invalid size of X and Y recieved please rectify and then try again","*"*10)
    return -1
  #Used to intialise fitting the model
  #The shape is checked to avoid issues during processing
  #Branch chosen depending if maxDepth specified or not
  #In either branch make_Split is used which returns the IG associated with the split and other data
  #If IG>0 only then split is made and recurrsive code works, else the node is made a root node
  if(self.Depth==None):#If no depth is Specified
    if(numRows>1 and len(np.unique(Y))>1):
      self.deep = max(self.deep,depth)
      #print("hi")
      IG,left,right,val,featureIdx = self.make_split(X,Y)
      #print(IG)
      if(IG>0):
        #print("valid")
        leftSide = self.fit(left[:,:-numFeaturesY],left[:,-numFeaturesY:],depth+1)
        rightSide=self.fit(right[:,:-numFeaturesY],right[:,-numFeaturesY:],depth+1)

        curr=Node(IG,featureIdx,val,leftSide,rightSide,dep=depth)
        if(depth==0): #Handling Root Node
          self.root = curr
          print("Depth",self.deep)
          self.istrained = True
        return curr
    Y1 = Y.tolist()
    return Node(value=max(Y1, key=Y1.count),dep=depth)#Leave node
```

b)make_split



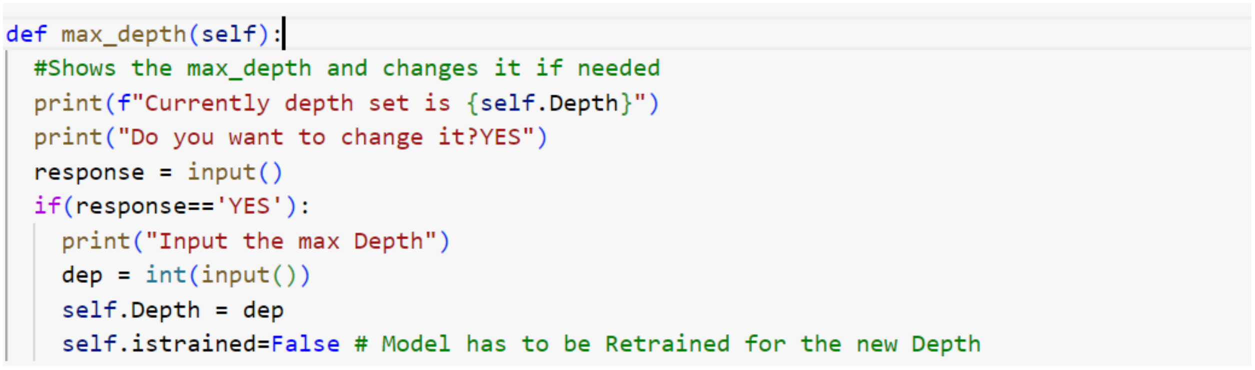
```
def make_split(self,X,Y):
  numRows,numFeatures = X.shape
  numRowsY,numFeaturesY = Y.shape
  data = np.concatenate((X, Y), axis=1)
  #Features to keep track of BestSplit
  maxIG = 0
  Bestleft =[]
  Bestright=[]
  BestVal=-1
  BestFeature=-1
  for fIdx in range(numFeatures): #Running at the Node for all the relevant features
    featureValues = X[:, fIdx]
    possibleValues= np.unique(featureValues)#Extracting unique values of a feature

    for val in possibleValues:  #iterating over the unique values and apt calculation for left and righ
      dataleft = data[data[:, fIdx] < val]
      dataright = data[data[:, fIdx] >= val]

      if(dataleft.shape[0]>0 and dataright.shape[0]>0):
        y, left_y, right_y = data[:, -numFeatures:], dataleft[:, -numFeatures:], dataright[:, -numFeatu
        currIG = self.cost_function(y, left_y, right_y)
        if(currIG>maxIG):#Setting Best Param
          maxIG=currIG
          Bestleft=dataleft
          Bestright=dataright
          BestVal = val
          BestFeature =fIdx
  return maxIG,Bestleft,Bestright,BestVal,BestFeature #Returning Best Param
```

e)predict():



```
def predict(self,X):
  #First Sees if the model is trained or not
  #Doesn't work on untrained model
  if(self.istrained):
    predictions = []
    for x in X:
      predictions.append(self.makePredict(x,self.root)) #If the model is trained then uses a recursive
    return predictions
  else:
    print("Model not trained yet, Can't predict")
    return []

def makePredict(self,X,node):
  # print(type(node))
  if node.leaf!=None: #Checks if leaf Node if yes return the value
    return node.leaf
  feature_val = X[node.feature_index]
  if feature_val<node.threshold: #If less then move left
    return self.makePredict(X, node.left)
  else:                           #Else move right
    return self.makePredict(X, node.right)
```

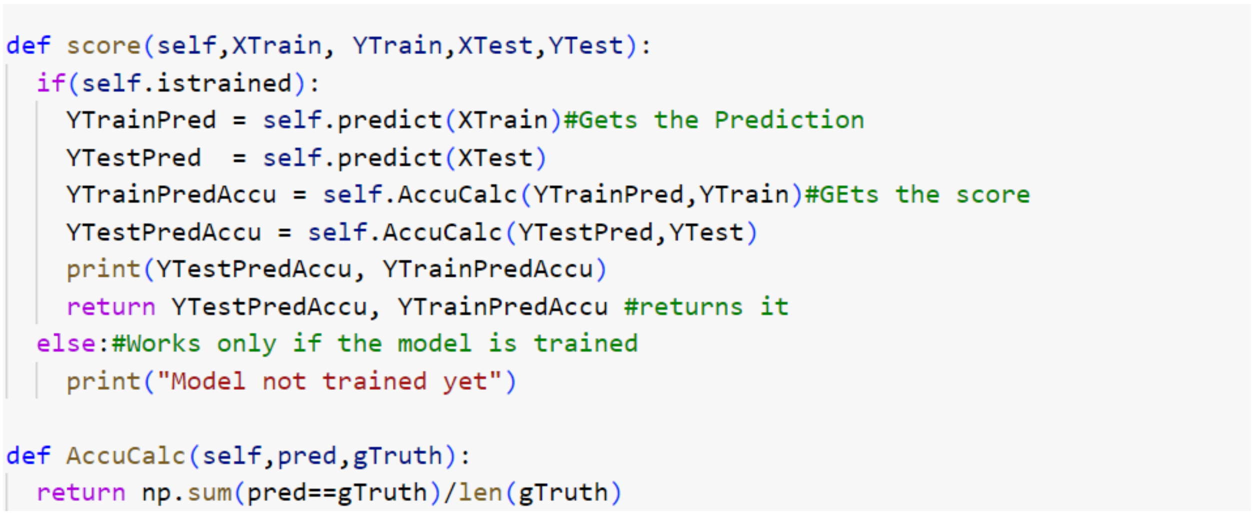
c)max_depth():



```python
def max_depth(self):
  #Shows the max_depth and changes it if needed
  print(f"Currently depth set is {self.Depth}")
  print("Do you want to change it?YES")
  response = input()
  if(response=='YES'):
    print("Input the max Depth")
    dep = int(input())
    self.Depth = dep
    self.istrained=False # Model has to be Retrained for the new Depth
```

Model has to be retrained on changing depth as is done by the variable istrained

f)score():



```python
def score(self,XTrain, YTrain,XTest,YTest):
  if(self.istrained):
    YTrainPred = self.predict(XTrain)#Gets the Prediction
    YTestPred  = self.predict(XTest)
    YTrainPredAccu = self.AccuCalc(YTrainPred,YTrain)#GEts the score
    YTestPredAccu = self.AccuCalc(YTestPred,YTest)
    print(YTestPredAccu, YTrainPredAccu)
    return YTestPredAccu, YTrainPredAccu #returns it
  else:#Works only if the model is trained
    print("Model not trained yet")

def AccuCalc(self,pred,gTruth):
  return np.sum(pred==gTruth)/len(gTruth)
```

b) Evaluation of Model

Entropy Model

```
dtE = MyDecisionTree()
```

```
dtE.fit(X_train1, y_train1)
```

```
Depth 11
<__main__.Node at 0x7e8c08110280>
```

```
pred=dtE.predict(X_test1)
```

```
accuracy = accuracy_score(y_test1, pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.9832089552238806
```

```
Train,Test=dtE.score(X_train1,y_train1,X_test1,y_test1)
```

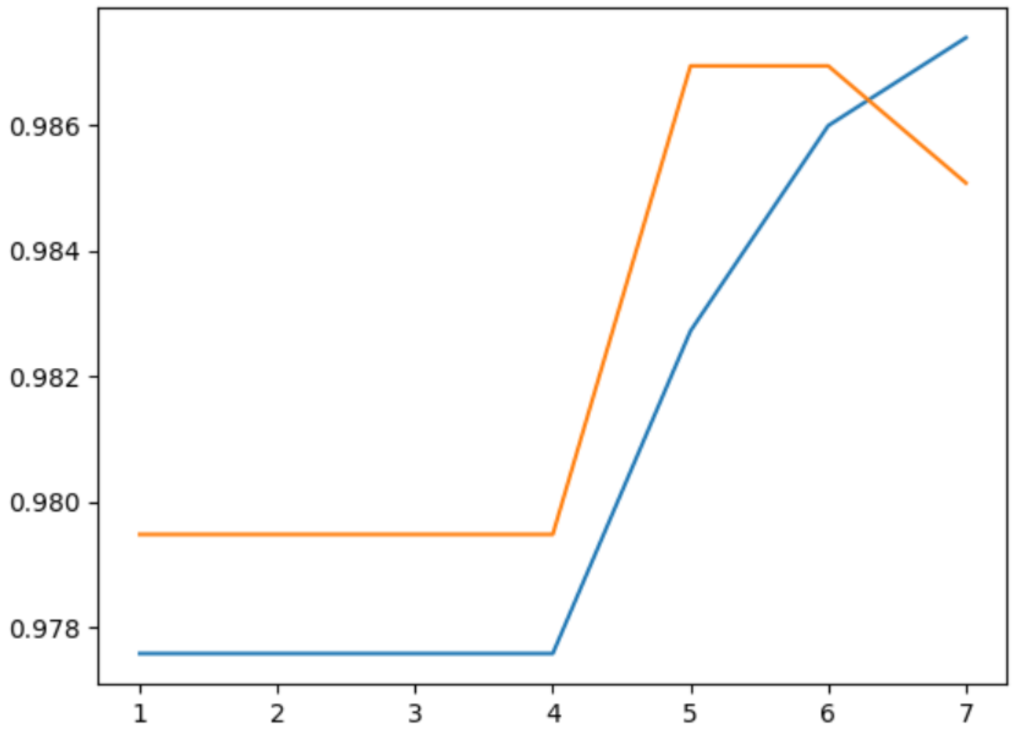
```
0.9832089552238806 1.0
```

Entropy for a range of depths

```
TrainingAccu =[]
TestingAccu = []
val = np.arange(1,8)
for i in val:
    dti = MyDecisionTree(i,i)
    dti.fit(X_train1, y_train1)
    Test,Train=dti.score(X_train1,y_train1,X_test1,y_test1)
    TrainingAccu.append(Train)
    TestingAccu.append(Test)
plt.plot(val,TrainingAccu,val,TestingAccu)
plt.show()
```

```
Depth 1
0.9794776119402985 0.9775805698271836
Depth 2
0.9794776119402985 0.9775805698271836
Depth 3
0.9794776119402985 0.9775805698271836
Depth 4
0.9794776119402985 0.9775805698271836
Depth 5
0.9869402985074627 0.982718355908454
Depth 6
0.9869402985074627 0.9859878561419897
Depth 7
0.9850746268656716 0.9873890705277908
```

Corresponding Graph



Best is at depth 7

```
Depth 7
0.9850746268656716 0.9873890705277908
```

Gini

```
dti = MyDecisionTree("gini")
dti.fit(X_train1, y_train1)
Test,Train=dti.score(X_train1,y_train1,X_test1,y_test1)
```

```
Depth 25
0.9888059701492538 1.0
```