DecisionTree

June 19, 2021

[ ]:
```
import sys
!{sys.executable} -m pip install ipynb
```

[2]:
```
%%capture
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ipynb.fs.full.CleaningData import getDataset
from ipynb.fs.full.CleaningData import getCovarianceVector

import warnings
warnings.filterwarnings('ignore')

pd.set_option("display.max_rows", 100)
pd.set_option("display.max_columns", None)
```

[ ]:
```
df = getDataset(500)

# df.head(100)
```

[ ]:
```
covVec = getCovarianceVector(df)
# print(covVec.head(100))
```

[ ]:
```
# print(covVec)
```

[6]:
```
class Node():
    def __init__(self, featureIndex=None,LeftNode=None, RightNode=None,␣
↪InformationGain=None,splitPoint= None, value=None):
        self.featureIndex = featureIndex ##to help define the conditions with␣
↪splitPoint
        self.RightNode = RightNode # right child node
        self.LeftNode = LeftNode # lefct child node
        self.InformationGain = InformationGain ## stored by info gained for␣
↪split by node
        self.splitPoint = splitPoint # to store the splitpoint
        self.value = value #value (determinite if it is a leaf node or not)
```

```
[7]: class DecisionTree():
    def __init__(self, MinDatapoints = 4, MaxDepth = 4):
        self.root = None

        #hyper parameters (to stop the tree)
        self.MinDatapoints = MinDatapoints #if the number of samples becomes␣
↪less than the minimum samples we won't split that node futher
        self.MaxDepth = MaxDepth # if the depth reaches the maximum depth

    def CreateTree(self, Data, currDepth = 0):
        X = Data[:,:-1]  #splitting the data into fetures and targets
        Y = Data[:,-1]
        NumofDatapoints, NumofFeatures = np.shape(X)

        # split until stopping conditions are met
        RequiredMinData = self.MinDatapoints
        RequiredDepth = self.MaxDepth
        if NumofDatapoints>= RequiredMinData and currDepth<=RequiredDepth:␣
↪#conditions
            # compute the optimal split

            BestSplit = self.GetBestSplit(Data, NumofDatapoints, NumofFeatures)
            if BestSplit["InfoGain"] > 0:
                LeftTree = self.CreateTree(BestSplit["LeftData"], currDepth+1)
                RightTree = self.CreateTree(BestSplit["RightData"], currDepth+1)
                # Returning the decision node
                return Node(BestSplit["FeatureIndex"],LeftTree,␣
↪RightTree,BestSplit["InfoGain"],BestSplit["Split"])

        #calculate leaf node value
        LeafNode = self.GetLeafValue(Y)
        # return leaf node
        return Node(value=LeafNode)

    def GetBestSplit(self, Data, NumofDatapoints, NumofFeatures):
        BestSplit = {}
        MaxGain = -float("inf") #must be negative infinity

        # loop over all the features
        for FIndex in range(NumofFeatures):
            Fvalue = Data[:, FIndex]
            Splits = np.unique(Fvalue)


            for Split in Splits:
                # get current split
                Data_Left, Data_Right = self.GetSplit(Data, FIndex, Split)
```

```
                # check if the neither of the two array splits are null
                if len(Data_Left)>0 and len(Data_Right)>0:
                    Y = Data[:,-1]
                    LeftSplitY = Data_Left[:,-1]
                    RightSplitY = Data_Right[:,-1]
                    # compute information gain
                    CurrentInfoGain = self.InfoGain(Y, LeftSplitY, RightSplitY)
                    # update the best split if needed to find maximum split
                    if CurrentInfoGain>MaxGain:
                        BestSplit["FeatureIndex"] = FIndex
                        BestSplit["LeftData"] = Data_Left
                        BestSplit["Split"] = Split
                        BestSplit["RightData"] = Data_Right
                        BestSplit["InfoGain"] = CurrentInfoGain
                        MaxGain = CurrentInfoGain

        # return best split
        return BestSplit

    def GetSplit(self, Data,FIndex, Split):
         #split the dataset into two using splitpoint
        DataRight =np.array([row for row in Data if row[FIndex]>Split])
        DataLeft = np.array([row for row in Data if row[FIndex]<=Split])
        return DataLeft, DataRight

    def InfoGain(self, ParentNode, LeftChildNode, RightChildNode): #calculate information gained
        WeightL = len(LeftChildNode) / len(ParentNode)
        WeightR = len(RightChildNode) / len(ParentNode)
        gain = self.gini_index(ParentNode) -(WeightL*self.gini_index(LeftChildNode) + WeightR*self.gini_index(RightChildNode))
        return gain #gone with gini index

    def gini_index(self, Y): #use gini index
        class_labels = np.unique(Y)
        gini = 0
        for cls in class_labels:
            p_cls = len(Y[Y == cls]) / len(Y)
            gini += p_cls**2
        return 1 - gini

    def GetLeafValue(self, Y):
        Y = list(Y)
        return max(Y, key=Y.count)


    def fit(self, X, Y):
```

```
        Data = np.concatenate((X, Y), axis=1)
        self.root = self.CreateTree(Data)

    def predict(self, X):
        preditions = [self.makePrediction(x, self.root) for x in X]
        return preditions

    def makePrediction(self, x, tree):
        if tree.value!=None:
            return tree.value
        FeatureValue = x[tree.featureIndex]
        if (FeatureValue>tree.splitPoint):
            return self.makePrediction(x, tree.RightNode)
        else:
            return self.makePrediction(x, tree.LeftNode)
```

[8]:

```
#splitting the dataset into training and testing sets
Training_set = df.sample(frac = 0.7,random_state = 25)
Test_set = df.drop(Training_set.index)




X_train = Training_set.iloc[:, :-2].values
Y_train = Training_set.iloc[:, -2].values.reshape(-1,1)

X_test = Test_set.iloc[:, :-2].values
Y_test = Test_set.iloc[:, -2].values.reshape(-1,1)
```

[9]:

```
%%time
#train model

Model = DecisionTree(25,20)

Model.fit(X_train,Y_train)
```

```
CPU times: user 19min 30s, sys: 5.9 s, total: 19min 36s
Wall time: 19min 46s
```

[ ]:

```
```

[10]:

```
#caluclate accuracy and confusion matrx
Y_pred = Model.predict(X_test)

Totalcorrect0 = 0
Totalcorrect1 = 0
Totalincorrect0 = 0
Totalincorrect1 = 0
for i in range(len(Y_pred)):
```

```
    if(Y_test[i] == 0 and Y_pred[i] == 0):
        Totalcorrect0 = Totalcorrect0 + 1
    elif(Y_test[i] == 1 and Y_pred[i] == 1):
        Totalcorrect1 = Totalcorrect1 +1
    elif(Y_test[i] == 1 and Y_pred[i] == 0):
        Totalincorrect0 = Totalincorrect0 +1
    elif(Y_test[i] == 0 and Y_pred[i] == 1):
        Totalincorrect1 = Totalincorrect1 +1

print("=================================")
print("Confusion Matrix \n")

print("Classes :  0      1")
print("     0    ",Totalcorrect0,"   ",Totalincorrect0)
print("     1    ",Totalincorrect1,"   ",Totalcorrect1)
print("")
print("=================================")
print("accuracy: " , (Totalcorrect0 + Totalcorrect1)/len(Y_pred))
print("==================================")
print("\n")
```

```
=================================
Confusion Matrix

Classes :  0      1
     0     1623     120
     1     68      1633

=================================
accuracy:  0.9454123112659698
==================================
```