**Abhishek Murthy**

**21BDS0064**

**Fall Sem 2024-2025**

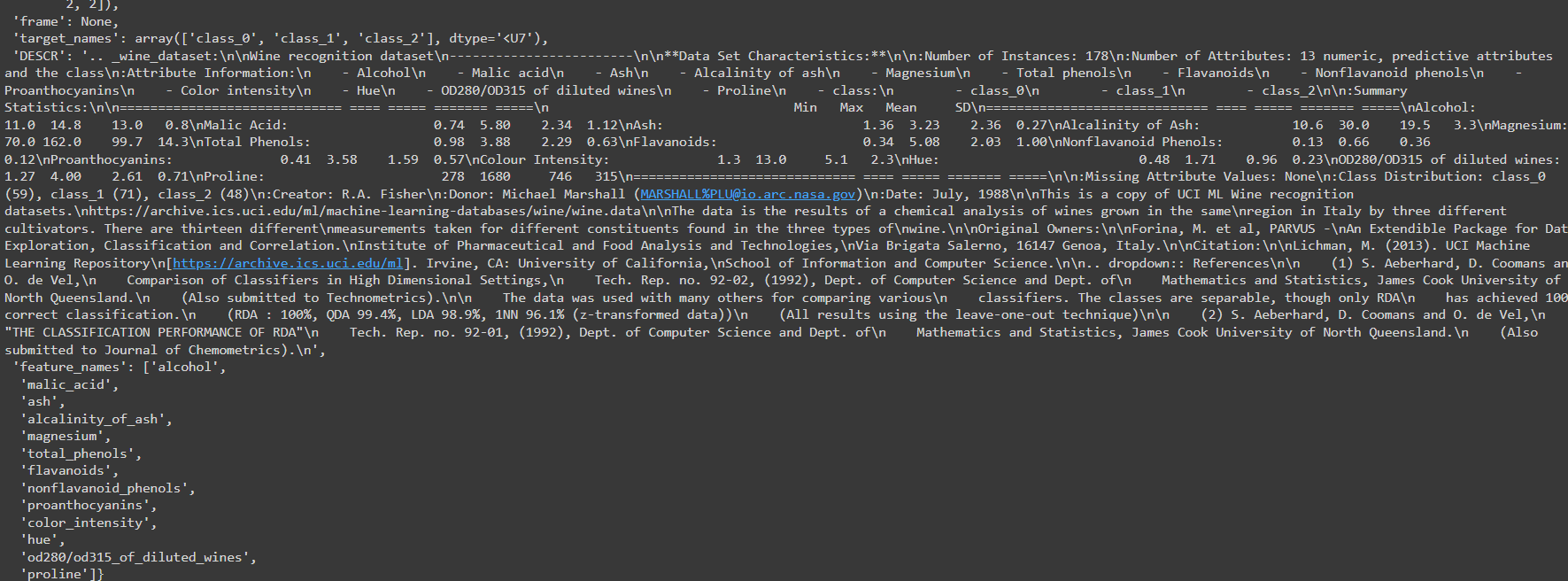
**DA-5**

**Machine Learning Lab**

**23-10-2024**

**Random Forest**

**Dataset:**

****

**Code:**

import numpy as np

from sklearn.datasets import load\_wine

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from graphviz import Digraph

def gini\_index(groups, classes):

n\_instances = float(sum([len(group) for group in groups]))

gini = 0.0

for group in groups:

size = float(len(group))

if size == 0:

continue

score = 0.0

for class\_val in classes:

p = [row[-1] for row in group].count(class\_val) / size

score += p \* p

gini += (1.0 - score) \* (size / n\_instances)

return gini

def test\_split(index, value, dataset):

left, right = list(), list()

for row in dataset:

if row[index] < value:

left.append(row)

else:

right.append(row)

return left, right

def get\_best\_split(dataset):

class\_values = list(set(row[-1] for row in dataset))

best\_index, best\_value, best\_score, best\_groups = None, None, float('inf'), None

for index in range(len(dataset[0]) - 1):

for row in dataset:

groups = test\_split(index, row[index], dataset)

gini = gini\_index(groups, class\_values)

if gini < best\_score:

best\_index, best\_value, best\_score, best\_groups = index, row[index], gini, groups

return {'index': best\_index, 'value': best\_value, 'groups': best\_groups}

def to\_terminal(group):

outcomes = [row[-1] for row in group]

return max(set(outcomes), key=outcomes.count)

def split(node, max\_depth, min\_size, depth):

left, right = node['groups']

del(node['groups'])

if not left or not right:

node['left'] = node['right'] = to\_terminal(left + right)

return

if depth >= max\_depth:

node['left'], node['right'] = to\_terminal(left), to\_terminal(right)

return

if len(left) <= min\_size:

node['left'] = to\_terminal(left)

else:

node['left'] = get\_best\_split(left)

split(node['left'], max\_depth, min\_size, depth + 1)

if len(right) <= min\_size:

node['right'] = to\_terminal(right)

else:

node['right'] = get\_best\_split(right)

split(node['right'], max\_depth, min\_size, depth + 1)

def build\_tree(train, max\_depth, min\_size):

root = get\_best\_split(train)

split(root, max\_depth, min\_size, 1)

return root

def predict(node, row):

if row[node['index']] < node['value']:

if isinstance(node['left'], dict):

return predict(node['left'], row)

else:

return node['left']

else:

if isinstance(node['right'], dict):

return predict(node['right'], row)

else:

return node['right']

def subsample(dataset, ratio=1.0):

sample = list()

n\_sample = round(len(dataset) \* ratio)

while len(sample) < n\_sample:

index = np.random.randint(len(dataset))

sample.append(dataset[index])

return sample

def export\_tree\_to\_dot(node, graph=None, node\_id=0):

if graph is None:

graph = Digraph()

if isinstance(node, dict):

feature\_index = node['index']

threshold = node['value']

left\_id = node\_id \* 2 + 1

right\_id = node\_id \* 2 + 2

graph.node(str(node\_id), f"X{feature\_index} < {threshold:.3f}")

graph = export\_tree\_to\_dot(node['left'], graph, left\_id)

graph.edge(str(node\_id), str(left\_id), label="Yes")

graph = export\_tree\_to\_dot(node['right'], graph, right\_id)

graph.edge(str(node\_id), str(right\_id), label="No")

else:

graph.node(str(node\_id), f"Leaf: {node}", shape="ellipse")

return graph

class RandomForest:

def \_\_init\_\_(self, n\_trees, max\_depth, min\_size, sample\_size):

self.n\_trees = n\_trees

self.max\_depth = max\_depth

self.min\_size = min\_size

self.sample\_size = sample\_size

self.trees = []

def fit(self, train):

self.trees = []

for i in range(self.n\_trees):

sample = subsample(train, self.sample\_size)

tree = build\_tree(sample, self.max\_depth, self.min\_size)

self.trees.append(tree)

print(f"Tree {i + 1} trained.")

def bagging\_predict(self, row):

predictions = [predict(tree, row) for tree in self.trees]

return max(set(predictions), key=predictions.count)

def predict(self, test):

predictions = [self.bagging\_predict(row) for row in test]

return predictions

def visualize\_tree(self, tree\_index):

if tree\_index < len(self.trees):

tree = self.trees[tree\_index]

dot = export\_tree\_to\_dot(tree)

dot.render(f"tree\_{tree\_index}", format="png", cleanup=False)

return dot

else:

print("Invalid tree index.")

def visualize\_all\_trees(self):

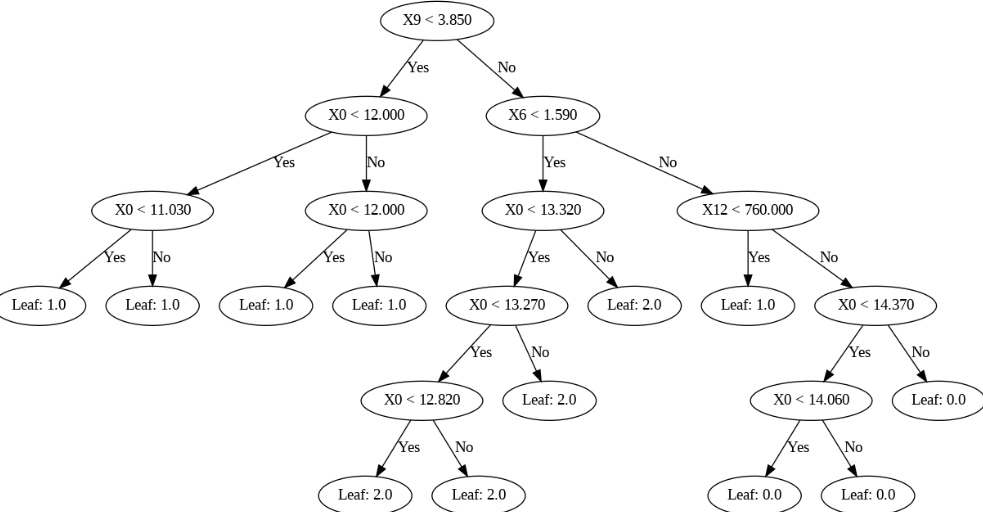
for i in range(len(self.trees)):

print(f"Visualizing Tree {i + 1}")

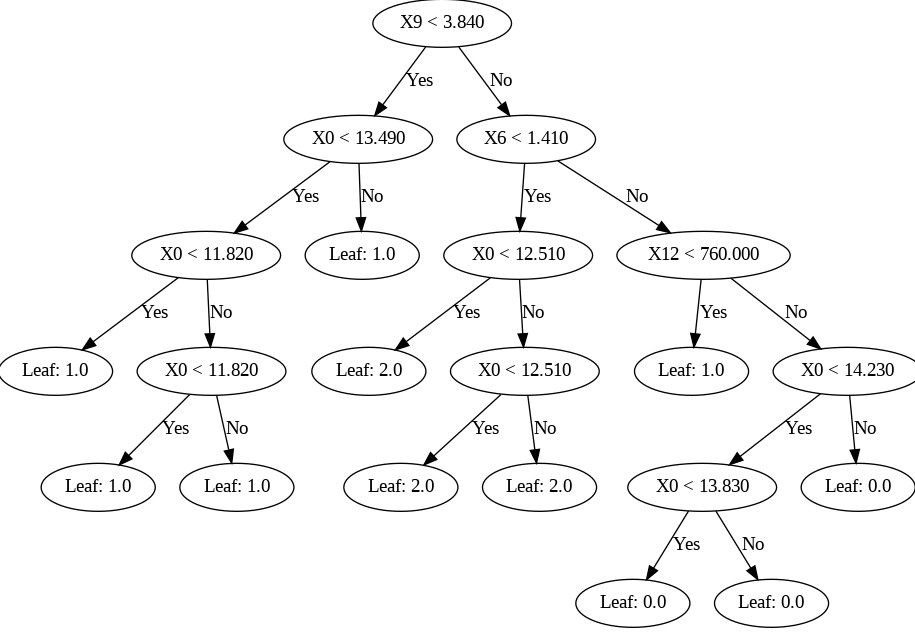
self.visualize\_tree(i)

**Output:**

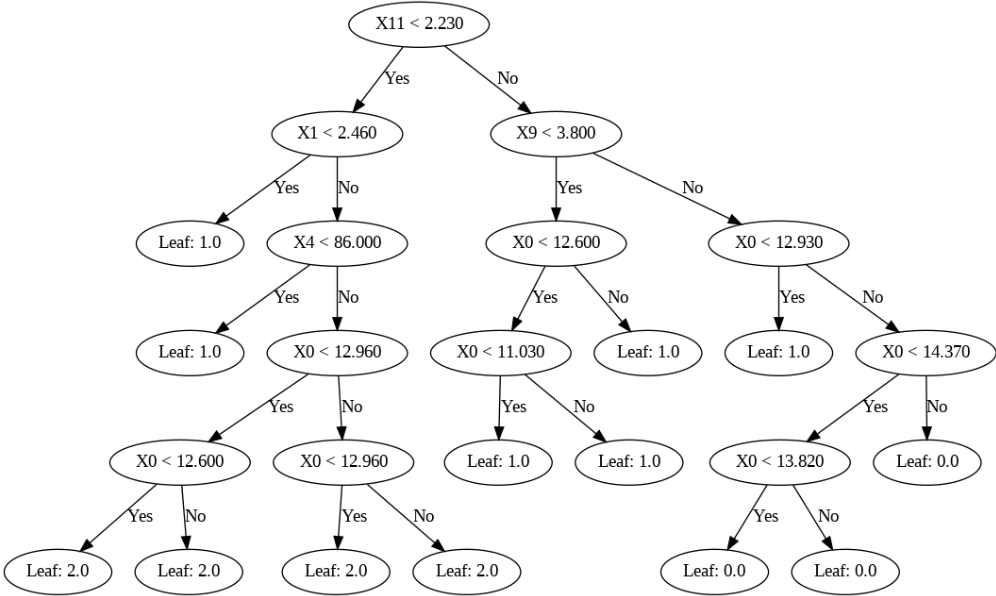
Tree 1

****

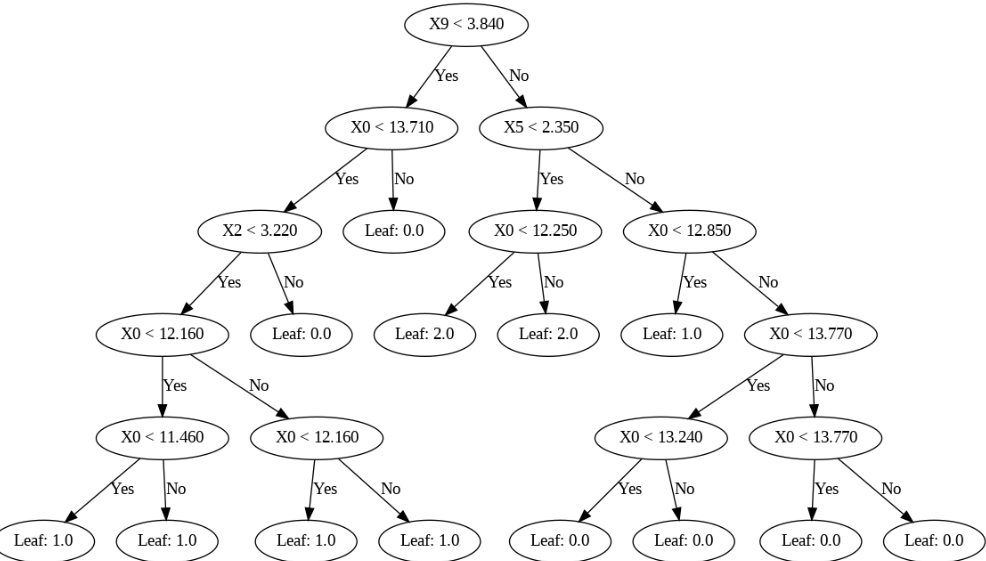
Tree 2

****

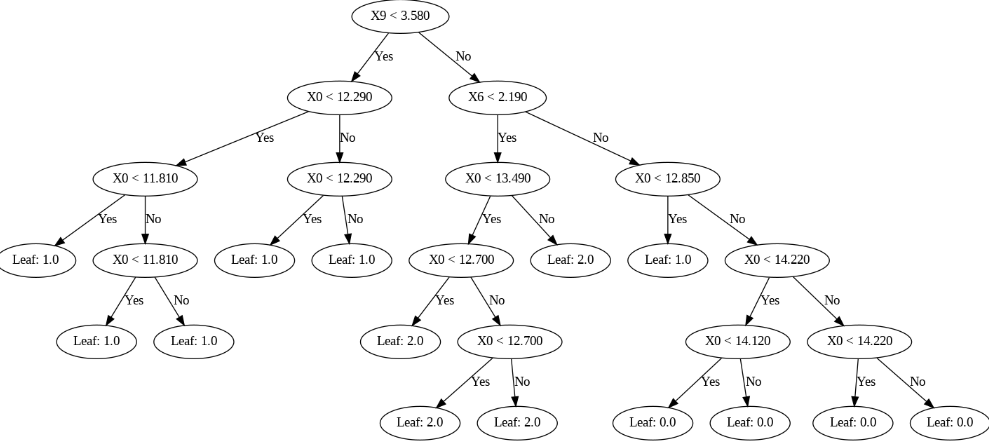
Tree 3



Tree 4



Tree 5



**Metrics**

**Naïve Bayes**

**Dataset**

****

**Code**

import pandas as pd

from collections import Counter

data = {

    'Color': ['Red', 'Blue', 'Blue', 'Red', 'Blue', 'Red', 'Blue', 'Red'],

    'Engine Type': ['Hybrid', 'Electric', 'Hybrid', 'Hybrid', 'Electric',

                    'Hybrid', 'Electric', 'Hybrid'],

    'Top Speed': ['Fast', 'Medium', 'Fast', 'Medium', 'Fast',

                  'Medium', 'Fast', 'Fast'],

    'Aerodynamics': ['Good', 'Excellent', 'Fair', 'Good', 'Fair',

                     'Excellent', 'Good', 'Excellent'],

    'Team': ['Red Bull', 'Mercedes', 'Red Bull', 'Red Bull',

             'Mercedes', 'Mercedes', 'Red Bull', 'Red Bull']

}

df = pd.DataFrame(data)

def class\_probs(df, target):

    total = len(df)

    class\_counts = Counter(df[target])

    class\_probs = {i: ct / total for i, ct in class\_counts.items()}

    return class\_counts, class\_probs

def feature\_probs(df, feature, target):

    feature\_dict = {}

    for class\_ in df[target].unique():

        mini\_df = df[df[target] == class\_]

        feature\_counts = Counter(mini\_df[feature].astype(str))

        tot\_count = len(mini\_df)

        feature\_dict[class\_] = {val: count / tot\_count for val, count in feature\_counts.items()}

    return feature\_dict

def calc\_probs(instance, feat\_probs, class\_probs):

    inst\_probs = {}

    for class\_, class\_prob in class\_probs.items():

        probs = class\_prob

        for i, feature\_val in enumerate(instance):

            if feature\_val in feat\_probs[i][class\_]:

                probs \*= feat\_probs[i][class\_][feature\_val]

            else:

                probs \*= 0

        inst\_probs[class\_] = probs

    return inst\_probs

target = 'Team'

class\_counts, class\_prob = class\_probs(df, target)

feature\_probs\_list = []

for feature in df.columns:

    if feature == target:

        continue

    feature\_probs\_list.append(feature\_probs(df, feature, target))

y\_true = df[target].apply(lambda x: 1 if x == 'Red Bull' else 0).tolist()

y\_pred = []

for i in range(len(df)):

    instance = df.iloc[i, :-1].tolist()

    val = calc\_probs(instance, feature\_probs\_list, class\_prob)

    predicted\_class = max(val, key=val.get)

    y\_pred.append(1 if predicted\_class == 'Red Bull' else 0)

tp = sum((1 for yt, yp in zip(y\_true, y\_pred) if yt == 1 and yp == 1))

tn = sum((1 for yt, yp in zip(y\_true, y\_pred) if yt == 0 and yp == 0))

fp = sum((1 for yt, yp in zip(y\_true, y\_pred) if yt == 0 and yp == 1))

fn = sum((1 for yt, yp in zip(y\_true, y\_pred) if yt == 1 and yp == 0))

accuracy = (tp + tn) / len(y\_true)

precision = tp / (tp + fp) if (tp + fp) != 0 else 0

recall = tp / (tp + fn) if (tp + fn) != 0 else 0

f1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) != 0 else 0

tpr = recall

fpr = fp / (fp + tn) if (fp + tn) != 0 else 0

print(f"TP: {tp}, TN: {tn}, FP: {fp}, FN: {fn}")

print(f"Accuracy: {accuracy:.2f}")

print(f"Precision: {precision:.2f}")

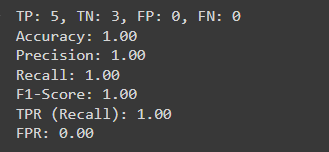
print(f"Recall: {recall:.2f}")

print(f"F1-Score: {f1\_score:.2f}")

print(f"TPR (Recall): {tpr:.2f}")

print(f"FPR: {fpr:.2f}")

**Output**

****