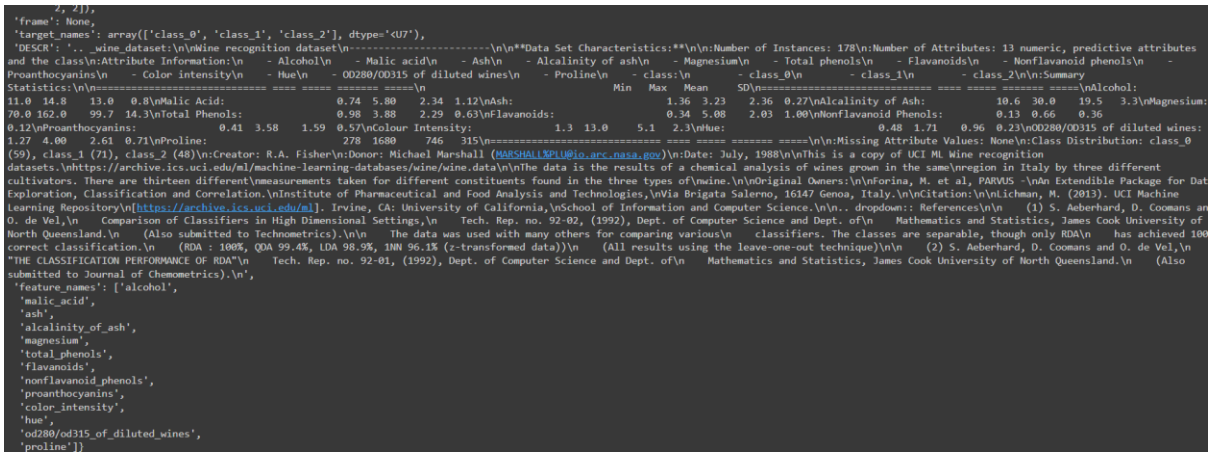
Abhishek Murthy
21BDS0064
Fall Sem 2024-2025
DA-5
Machine Learning Lab
23-10-2024

Random Forest

Dataset:



Code:

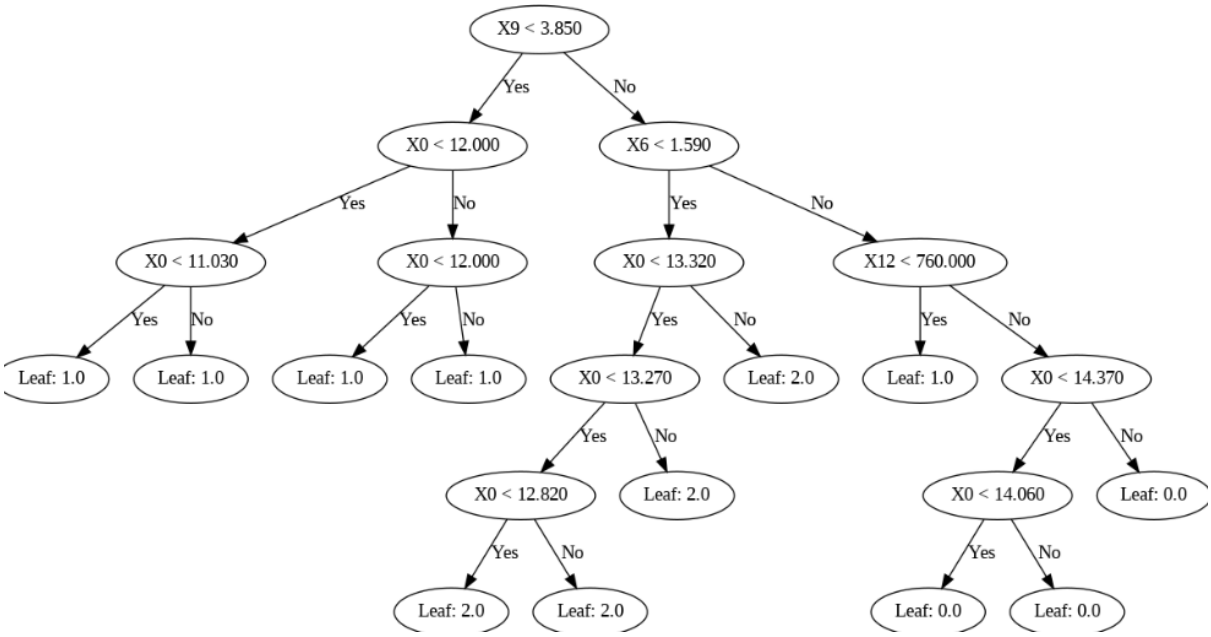
```
import numpy as np
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from graphviz import Digraph

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()
```

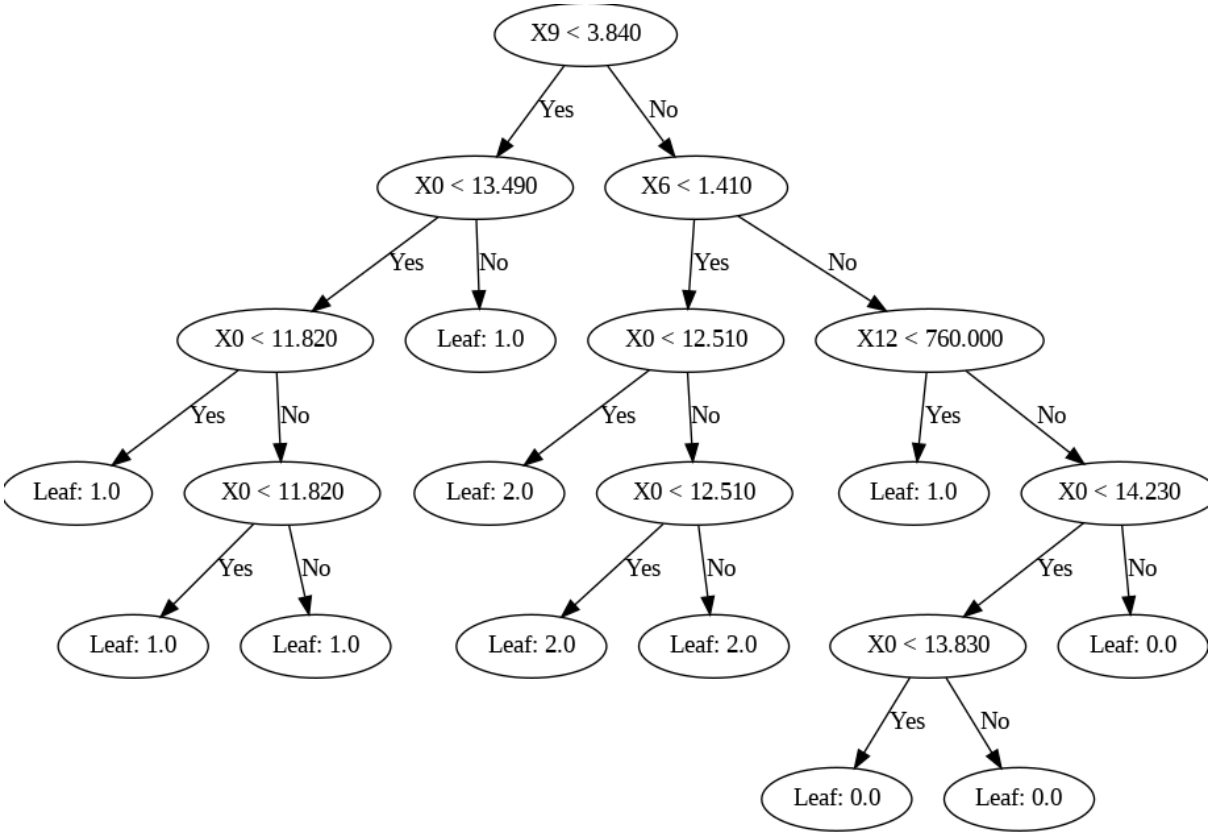
```
    if isinstance(node, dict):
        feature_index = node['index']
        threshold = node['value']
        left_id = node_id * 2 + 1
        right_id = node_id * 2 + 2
        graph.node(str(node_id), f"X{feature_index} < {threshold:.3f}")
        graph = export_tree_to_dot(node['left'], graph, 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:

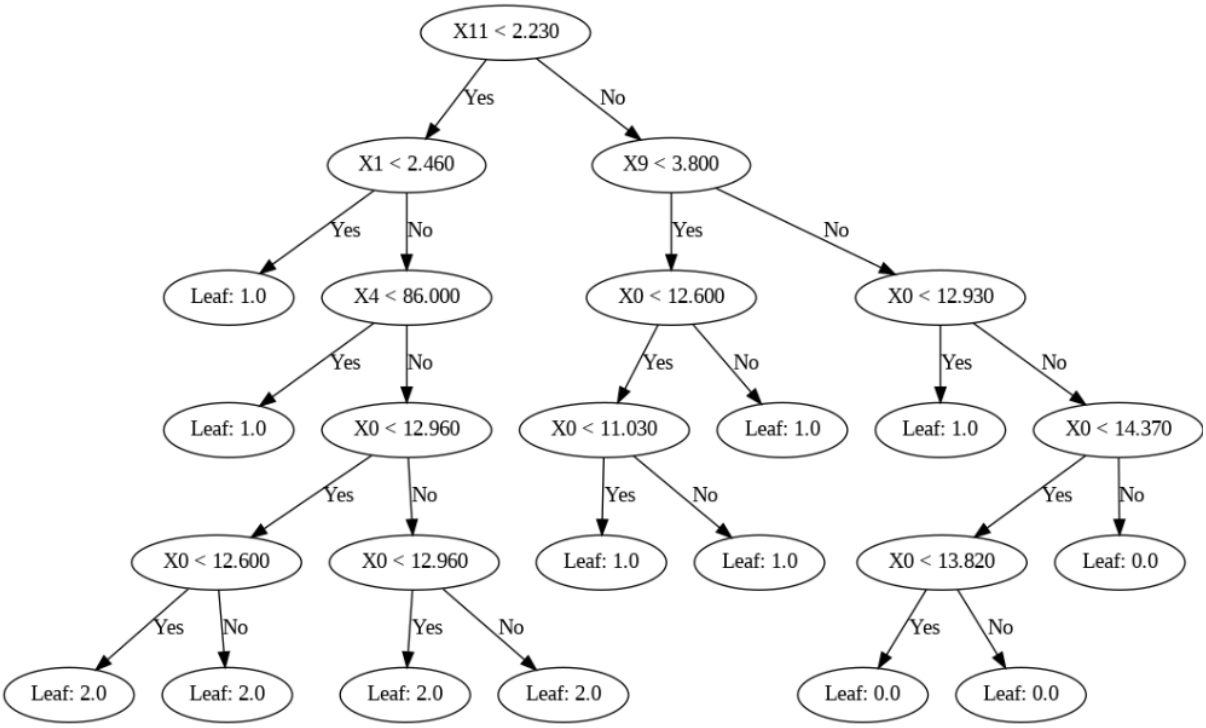
Tree 1



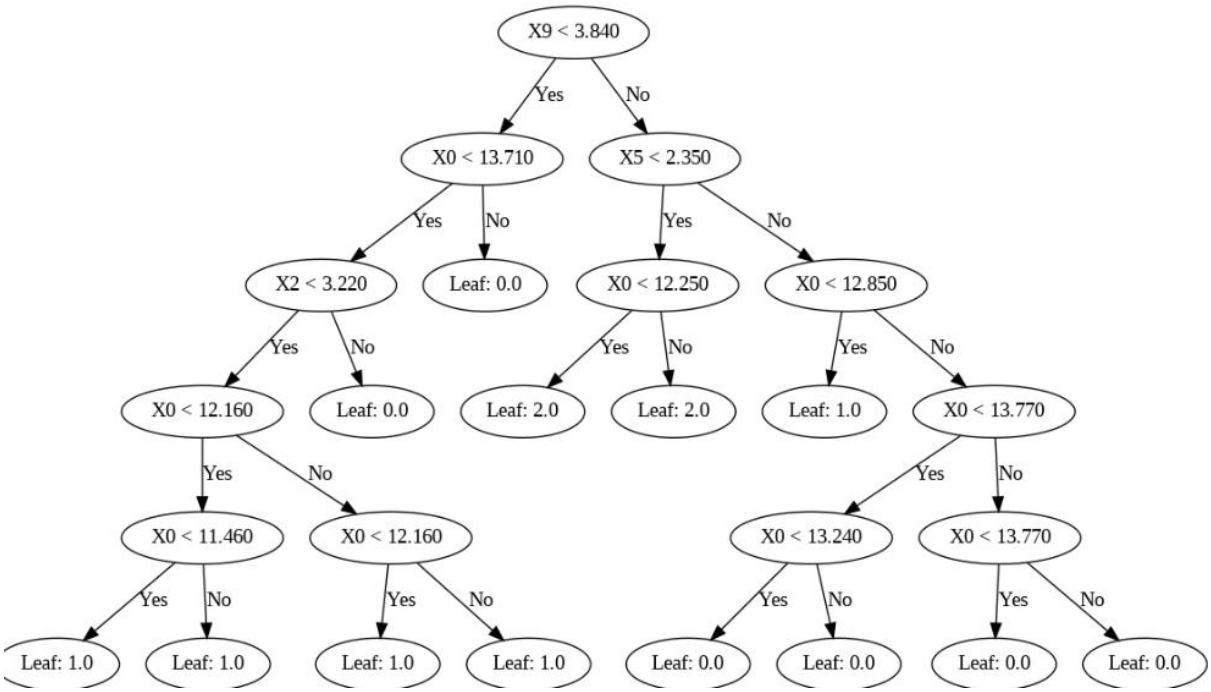
Tree 2



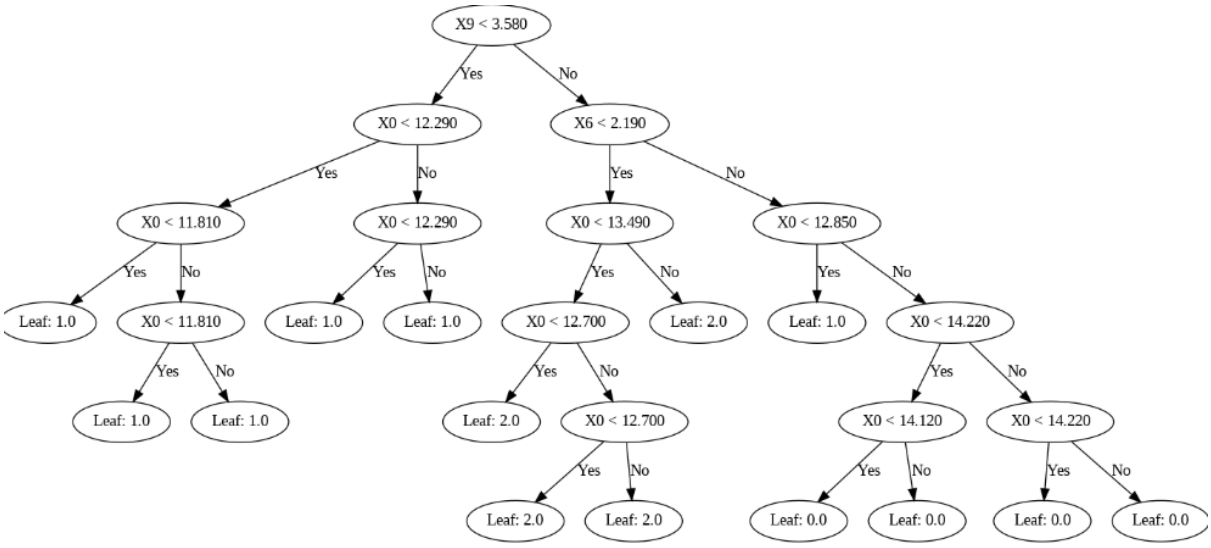
Tree 3


X11 < 2.230
Yes
No
X1 < 2.460
X9 < 3.800
Yes
No
Yes
No
Leaf: 1.0
X4 < 86.000
X0 < 12.600
X0 < 12.930
Yes
No
Yes
No
Yes
No
Leaf: 1.0
X0 < 12.960
X0 < 11.030
Leaf: 1.0
Leaf: 1.0
X0 < 14.370
Yes
No
Yes
No
Yes
No
X0 < 12.600
X0 < 12.960
Leaf: 1.0
Leaf: 1.0
X0 < 13.820
Leaf: 0.0
Yes
No
Yes
No
Yes
No
Leaf: 2.0
Leaf: 2.0
Leaf: 2.0
Leaf: 2.0
Leaf: 0.0
Leaf: 0.0

Tree 4


X9 < 3.840
Yes
No
X0 < 13.710
X5 < 2.350
Yes
No
Yes
No
X2 < 3.220
Leaf: 0.0
X0 < 12.250
X0 < 12.850
Yes
No
Yes
No
Yes
No
X0 < 12.160
Leaf: 0.0
Leaf: 2.0
Leaf: 2.0
Leaf: 1.0
X0 < 13.770
Yes
No
Yes
No
X0 < 11.460
X0 < 12.160
X0 < 13.240
X0 < 13.770
Yes
No
Yes
No
Yes
No
Yes
No
Leaf: 1.0
Leaf: 1.0
Leaf: 1.0
Leaf: 1.0
Leaf: 0.0
Leaf: 0.0
Leaf: 0.0
Leaf: 0.0

Tree 5


X9 < 3.580
Yes
No
X0 < 12.290
X6 < 2.190
Yes
No
Yes
No
X0 < 11.810
X0 < 12.290
X0 < 13.490
X0 < 12.850
Yes
No
Yes
No
Yes
No
Yes
No
Leaf: 1.0
X0 < 11.810
Leaf: 1.0
Leaf: 1.0
X0 < 12.700
Leaf: 2.0
Leaf: 1.0
X0 < 14.220
Yes
No
Yes
No
Yes
No
Leaf: 1.0
Leaf: 1.0
Leaf: 2.0
X0 < 12.700
X0 < 14.120
X0 < 14.220
Yes
No
Yes
No
Yes
No
Leaf: 2.0
Leaf: 2.0
Leaf: 0.0
Leaf: 0.0
Leaf: 0.0
Leaf: 0.0

Metrics

Naïve Bayes

Dataset

Color	Engine Type	Top Speed	Aerodynamics	Team
Red	Hybrid	Fast	Good	Red Bull
Blue	Electric	Medium	Excellent	Mercedes
Blue	Hybrid	Fast	Fair	Red Bull
Red	Hybrid	Medium	Good	Red Bull
Blue	Electric	Fast	Fair	Mercedes
Red	Hybrid	Medium	Excellent	Mercedes
Blue	Electric	Fast	Good	Red Bull
Red	Hybrid	Fast	Excellent	Red Bull

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

```
TP: 5, TN: 3, FP: 0, FN: 0
Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1-Score: 1.00
TPR (Recall): 1.00
FPR: 0.00
```