BITS F464 - Semester 1 - MACHINE LEARNING

ASSIGNMENT 2 – DECISION TREES AND SUPPORT VECTOR MACHINES

-Please rename the file as "TeamXX_Assignment2.ipynb"

Team number: 12

(*In Title case, separated with commas*) ***Full names of all students in the team:*** Sai Karthik C, Aditya Abhiram, Paidisetty Sai Aditya, Ishan Harsh, Kamal Aditya M

(*Separated by commas*) ***Id number of all students in the team:*** 2020B5A70762H, 2020B5A70937H, 2020B5A70987H, 2021A7PS2854H, 2020A7PS0054H

This assignment aims to identify the differences between three Machine Learning models.

1. Preprocess and perform exploratory data analysis of the dataset obtained

```
#importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
df=pd.read_csv("./communities.csv",header=None)
df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	118	119	120
0	8	?	?	Lakewoodcity	1	0.19	0.33	0.02	0.90	0.12	...	0.12	0.26	0.20
1	53	?	?	Tukwilacity	1	0.00	0.16	0.12	0.74	0.45	...	0.02	0.12	0.45
2	24	?	?	Aberdeentown	1	0.00	0.42	0.49	0.56	0.17	...	0.01	0.21	0.02
3	34	5	81440	Willingborotownship	1	0.04	0.77	1.00	0.08	0.12	...	0.02	0.39	0.28
4	42	95	6096	Bethlehemtownship	1	0.01	0.55	0.02	0.95	0.09	...	0.04	0.09	0.02

5 rows × 128 columns

```
#add column names
with open('column_names.txt', 'r') as file:
    lines = file.read().splitlines()

# Extract column names from the lines
column_names = [line.split()[1] for line in lines]

# Assign column names to the DataFrame
df.columns = column_names

# Verify the changes
print(df.head())
```

```
   state county community        communityname  fold  population  \
0      8      ?         ?         Lakewoodcity     1        0.19
1     53      ?         ?          Tukwilacity     1        0.00
2     24      ?         ?         Aberdeentown     1        0.00
3     34      5     81440  Willingborotownship     1        0.04
4     42     95      6096    Bethlehemtownship     1        0.01

   householdsize  racepctblack  racePctWhite  racePctAsian  ...  LandArea  \
0           0.33          0.02          0.90          0.12  ...      0.12
1           0.16          0.12          0.74          0.45  ...      0.02
2           0.42          0.49          0.56          0.17  ...      0.01
3           0.77          1.00          0.08          0.12  ...      0.02
4           0.55          0.02          0.95          0.09  ...      0.04

   PopDens  PctUsePubTrans  PolicCars  PolicOperBudg  LemasPctPolicOnPatr  \
0     0.26            0.20       0.06           0.04                  0.9
1     0.12            0.45          ?              ?                    ?
2     0.21            0.02          ?              ?                    ?
3     0.39            0.28          ?              ?                    ?
4     0.09            0.02          ?              ?                    ?

   LemasGangUnitDeploy  LemasPctOfficDrugUn  PolicBudgPerPop  \
0                  0.5                 0.32             0.14
1                    ?                 0.00                ?
2                    ?                 0.00                ?
3                    ?                 0.00                ?
4                    ?                 0.00                ?

   ViolentCrimesPerPop
0                 0.20
1                 0.67
2                 0.43
3                 0.12
4                 0.03

[5 rows x 128 columns]
```

```
df.dtypes
```

```
state                    int64
county                  object
community               object
communityname           object
fold                     int64
                        ...
LemasPctPolicOnPatr     object
LemasGangUnitDeploy     object
LemasPctOfficDrugUn    float64
PolicBudgPerPop         object
ViolentCrimesPerPop    float64
Length: 128, dtype: object
```

```
df.drop(columns='communityname', inplace=True)
df.head()
```

	state	county	community	fold	population	householdsize	racepctblack	racePc
0	8	?	?	1	0.19	0.33	0.02	
1	53	?	?	1	0.00	0.16	0.12	
2	24	?	?	1	0.00	0.42	0.49	
3	34	5	81440	1	0.04	0.77	1.00	
4	42	95	6096	1	0.01	0.55	0.02	

5 rows × 127 columns

```
df = df.apply(pd.to_numeric, errors='coerce')
print(df.dtypes)
```

```
state                    int64
county                 float64
community              float64
fold                     int64
population             float64
                         ...
LemasPctPolicOnPatr    float64
LemasGangUnitDeploy    float64
LemasPctOfficDrugUn    float64
PolicBudgPerPop        float64
ViolentCrimesPerPop    float64
Length: 127, dtype: object
```

```
# '?' to nan
df = df.replace('?',np.nan)
print(df.isnull().sum())
num_rows=df.shape[0]
print(num_rows)
total_missing_values = df.isna().sum().sum()
print("Total number of missing values in the DataFrame:", total_missing_values)
```

```
state                     0
county                 1174
community              1177
fold                      0
population                0
                       ...
LemasPctPolicOnPatr    1675
LemasGangUnitDeploy    1675
LemasPctOfficDrugUn       0
PolicBudgPerPop        1675
ViolentCrimesPerPop       0
Length: 127, dtype: int64
1994
Total number of missing values in the DataFrame: 39202
```

```
#removing all the columns which have more than 1000 missing values
threshold = 1000
df.dropna(axis=1, thresh=df.shape[0] - threshold, inplace=True)
```

```
total_missing_values = df.isna().sum().sum()
print("Total number of missing values in the DataFrame:", total_missing_values)
```

```
Total number of missing values in the DataFrame: 1
```

```
df = df.fillna(df.mean())
```

```
total_missing_values = df.isna().sum().sum()
print("Total number of missing values in the DataFrame:", total_missing_values)
```

```
Total number of missing values in the DataFrame: 0
```

```
df1=df.copy()
```

```
intervals = [0.0,0.2,0.6,1.000]

# Create the categorical bins for the target variable
df1['target_bins'] = pd.cut(df1['ViolentCrimesPerPop'], bins=intervals, labels=False, include_lowest=T
print(df1.shape)
```

```
(1994, 104)
```

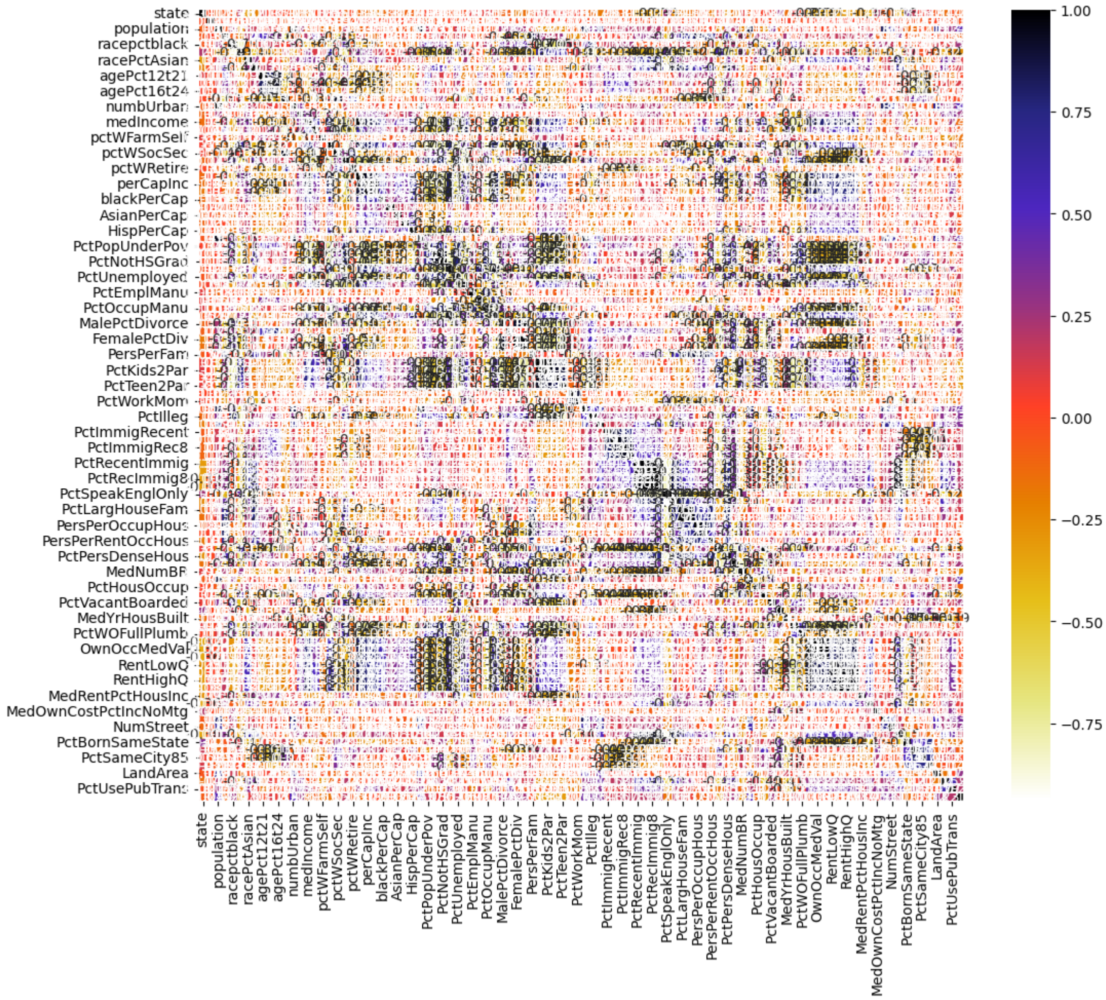
```
df.head()
```

	state	fold	population	householdsize	racepctblack	racePctWhite	racePctAsia
0	8	1	0.19	0.33	0.02	0.90	0.1
1	53	1	0.00	0.16	0.12	0.74	0.4
2	24	1	0.00	0.42	0.49	0.56	0.1
3	34	1	0.04	0.77	1.00	0.08	0.1
4	42	1	0.01	0.55	0.02	0.95	0.0

5 rows × 103 columns

```
X1=df1.drop(["ViolentCrimesPerPop","target_bins"],axis=1)
y1=df1["target_bins"]
X2=X1.copy()
```

```
import seaborn as sns
#Using Pearson Correlation
plt.figure(figsize=(12,10))
cor = X1.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.CMRmap_r)
plt.show()
```



```
def correlation(dataset, threshold):
    col_corr = set()  # Set of all the names of correlated columns
    corr_matrix = dataset.corr()
    for i in range(len(corr_matrix.columns)):
        for j in range(i):
            if (corr_matrix.iloc[i, j]) > threshold: # we are interested in absolute coeff value
                colname = corr_matrix.columns[i]  # getting the name of column
                col_corr.add(colname)
    return col_corr
```

```
corr_features=correlation(X1,0.90)
print(len(set(corr_features)))
print(X1.shape)
set(corr_features)
X1=X1.drop(columns=corr_features)
X1.shape
```

```
31
(1994, 102)
(1994, 71)
```

```
X = X1.values
col_no=X1.shape[1]
mean_ = X.mean(axis =0)
std_ = X.std(axis=0)

X_scaled = (X-mean_)/std_
print(f'Shape of X_scaled: {X_scaled.shape}')
features = X_scaled.T
print(f'Shape of features: {features.shape}')
cov_matrix = np.cov(features)
print(f'Shape of cov_matrix: {cov_matrix.shape}')
print('\nSneak Peak of the covariance matrix:\n')
cov_matrix[0:5, 0:5]
eig_values, eig_vectors = np.linalg.eig(cov_matrix)



print(f'First 10 eigenvalues: {eig_values[:10]}')
print(f'\n\nLast 10 eigenvalues: {eig_values[-10:]}')
plt.figure(figsize=(5,3))
plt.stem(eig_values[:102], use_line_collection = True)
plt.xlabel('Eigen value index')
plt.ylabel('Eigen value')
plt.show()
for i in range(col_no):
    exp_var = np.sum(eig_values[:i+1])*100 / np.sum(eig_values)
    print(f'Eigenvectors upto {i+1} expresses {exp_var} % variance')
result = np.dot(X_scaled.reshape((-1, col_no)), eig_vectors[:,:50].reshape((col_no, -1)))
result.shape
X_new = pd.DataFrame()
for i in range(50):
    projected = X_scaled.dot(eig_vectors.T[i])
    X_new['PC'+str(i)] = projected
```

```
Eigenvectors upto 13 expresses 78.03830647214453 % variance
Eigenvectors upto 14 expresses 79.36497636565765 % variance
Eigenvectors upto 15 expresses 80.62221390663616 % variance
Eigenvectors upto 16 expresses 81.83186659973276 % variance
Eigenvectors upto 17 expresses 82.97923781721288 % variance
Eigenvectors upto 18 expresses 84.01572089397234 % variance
Eigenvectors upto 19 expresses 84.98000803853024 % variance
Eigenvectors upto 20 expresses 85.89053425605748 % variance
Eigenvectors upto 21 expresses 86.73889497387425 % variance
Eigenvectors upto 22 expresses 87.5430580026014 % variance
Eigenvectors upto 23 expresses 88.28373022168475 % variance
Eigenvectors upto 24 expresses 88.9964404049252 % variance
Eigenvectors upto 25 expresses 89.68134141986853 % variance
Eigenvectors upto 26 expresses 90.35540291895923 % variance
Eigenvectors upto 27 expresses 90.96113971605314 % variance
Eigenvectors upto 28 expresses 91.55887969667846 % variance
Eigenvectors upto 29 expresses 92.11136322319796 % variance
Eigenvectors upto 30 expresses 92.64977814276278 % variance
Eigenvectors upto 31 expresses 93.17362158665918 % variance
Eigenvectors upto 32 expresses 93.64818073319059 % variance
Eigenvectors upto 33 expresses 94.11283552090426 % variance
Eigenvectors upto 34 expresses 94.54040636065956 % variance
Eigenvectors upto 35 expresses 94.93144207974714 % variance
Eigenvectors upto 36 expresses 95.31778558642435 % variance
Eigenvectors upto 37 expresses 95.67354668657788 % variance
Eigenvectors upto 38 expresses 95.9944867564876 % variance
Eigenvectors upto 39 expresses 96.30119137040298 % variance
Eigenvectors upto 40 expresses 96.59415179331498 % variance
Eigenvectors upto 41 expresses 96.87610389150299 % variance
Eigenvectors upto 42 expresses 97.12971004466604 % variance
Eigenvectors upto 43 expresses 97.35648004954078 % variance
Eigenvectors upto 44 expresses 97.57162522197451 % variance
Eigenvectors upto 45 expresses 97.77753977603598 % variance
Eigenvectors upto 46 expresses 97.9698681713905 % variance
Eigenvectors upto 47 expresses 98.14255512352531 % variance
Eigenvectors upto 48 expresses 98.3023243617003 % variance
Eigenvectors upto 49 expresses 98.44431567114125 % variance
Eigenvectors upto 50 expresses 98.58157381915163 % variance
Eigenvectors upto 51 expresses 98.7090449107571 % variance
Eigenvectors upto 52 expresses 98.82818445711268 % variance
Eigenvectors upto 53 expresses 98.94642329775277 % variance
Eigenvectors upto 54 expresses 99.05846060664744 % variance
Eigenvectors upto 55 expresses 99.15996212779991 % variance
Eigenvectors upto 56 expresses 99.1804435404189 % variance
Eigenvectors upto 57 expresses 99.26890138486479 % variance
Eigenvectors upto 58 expresses 99.35480181083182 % variance
Eigenvectors upto 59 expresses 99.43442046341413 % variance
Eigenvectors upto 60 expresses 99.5080091142434 % variance
Eigenvectors upto 61 expresses 99.5348375718669 % variance
Eigenvectors upto 62 expresses 99.59886808562918 % variance
Eigenvectors upto 63 expresses 99.65861788909987 % variance
Eigenvectors upto 64 expresses 99.6902126772966 % variance
Eigenvectors upto 65 expresses 99.72443294506097 % variance
Eigenvectors upto 66 expresses 99.761900707446 % variance
Eigenvectors upto 67 expresses 99.8013255428829 % variance
Eigenvectors upto 68 expresses 99.85738830805492 % variance
```

```
X_new.head()
```

	PC0	PC1	PC2	PC3	PC4	PC5	PC6	PC7	
0	2.302278	0.387429	2.653746	0.586517	-0.949278	2.108505	0.501496	0.099017	-0.30
1	0.221924	0.324474	2.767893	0.920026	1.240554	2.749155	3.190882	0.779405	0.13
2	-0.922689	-1.754886	0.085793	0.535151	0.024471	1.517617	1.247109	-1.560448	-1.53
3	1.384694	2.033427	-2.264064	0.153777	-3.031430	-1.037113	1.841230	-4.181932	-2.18
4	4.941237	-0.904792	-2.124714	-0.082720	-1.315343	0.726998	-0.323037	0.435373	-0.66

5 rows × 50 columns

```
from sklearn.model_selection import train_test_split
X_train1, X_test1, y_train1, y_test1 = train_test_split(
    X_new,
    y1,
    test_size=0.3,
    random_state=42)
```

```
from sklearn.model_selection import train_test_split
X_train2, X_test2, y_train2, y_test2 = train_test_split(
    X1,
    y1,
    test_size=0.3,
    random_state=42)
```

2. Decision tree model with entropy implementation

2.1 Implementation of the Model

```
class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y, sample_weights=None):
        self.tree = self._build_tree(X, y, sample_weights, depth=0)

    def _build_tree(self, X, y, sample_weights, depth):
```

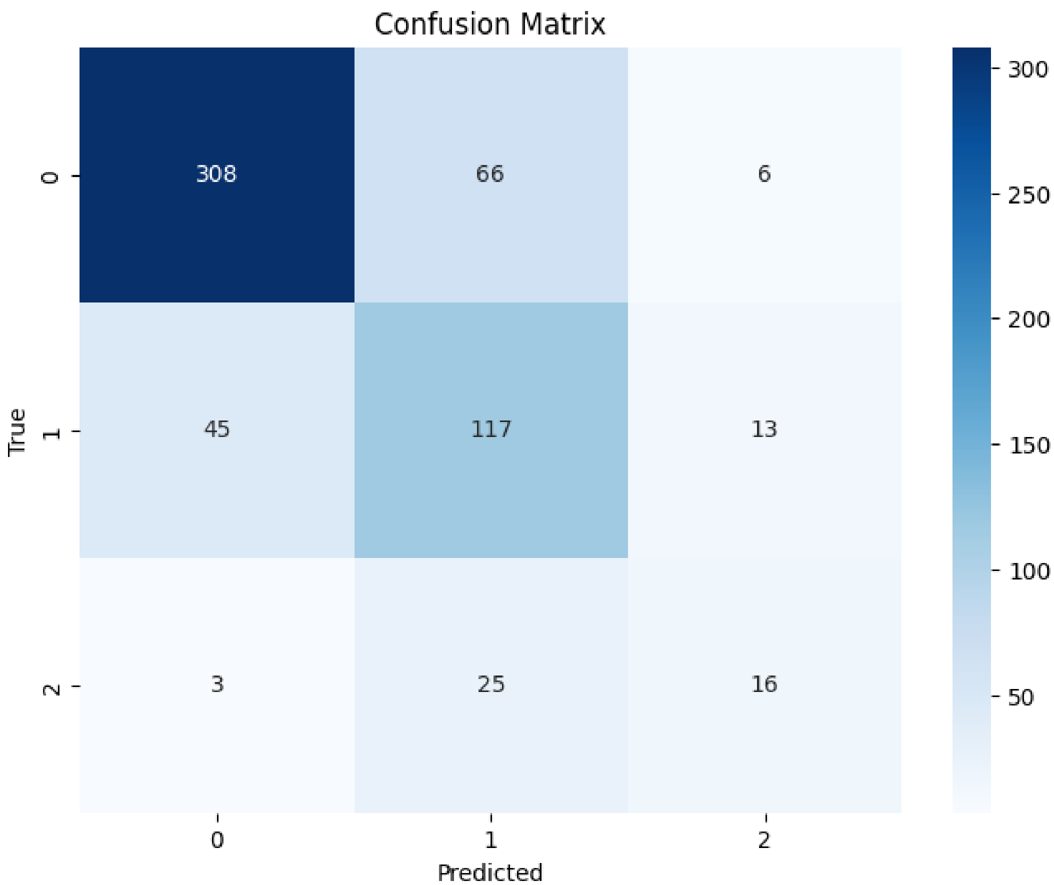
```
        num_samples, num_features = X.shape
        unique_classes, class_counts = np.unique(y, return_counts=True)
        default_class = unique_classes[np.argmax(class_counts)]

        # Base cases
        if depth == self.max_depth or len(np.unique(y)) == 1:
            return {'class': default_class, 'count': len(y)}

        # If sample weights are provided, compute the weighted Gini impurity
        if sample_weights is not None:
            total_weight = np.sum(sample_weights)
            weighted_gini = 1.0 - np.sum((np.sum(sample_weights[y == c]) / total_weight) ** 2 for c in u
        else:
            weighted_gini = 1.0 - np.sum((np.sum(y == c) / num_samples) ** 2 for c in unique_classes)

        best_split = {'feature_index': None, 'threshold': None, 'gini': weighted_gini}

        # Iterate through each feature and find the best split
        for feature_index in range(num_features):
            unique_values = np.unique(X[:, feature_index])
            for threshold in unique_values:
                left_mask = X[:, feature_index] <= threshold
                right_mask = ~left_mask

                if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
                    continue

                left_gini = self._calculate_gini(y[left_mask], sample_weights[left_mask] if sample_weig
                right_gini = self._calculate_gini(y[right_mask], sample_weights[right_mask] if sample_we

                weighted_avg_gini = (np.sum(left_mask) * left_gini + np.sum(right_mask) * right_gini) /

                if weighted_avg_gini < best_split['gini']:
                    best_split = {
                        'feature_index': feature_index,
                        'threshold': threshold,
                        'gini': weighted_avg_gini,
                        'left_mask': left_mask,
                        'right_mask': right_mask
                    }

        if best_split['feature_index'] is None:
            return {'class': default_class, 'count': len(y)}

        left_subtree = self._build_tree(X[best_split['left_mask']], y[best_split['left_mask']], sample_w
        right_subtree = self._build_tree(X[best_split['right_mask']], y[best_split['right_mask']], samp

        return {
            'feature_index': best_split['feature_index'],
            'threshold': best_split['threshold'],
            'left': left_subtree,
            'right': right_subtree
        }

    def _calculate_gini(self, labels, weights=None):
        num_samples = len(labels)
        if weights is None:
            weights = np.ones(num_samples) / num_samples

        unique_classes, class_counts = np.unique(labels, return_counts=True)
        gini = 1.0 - np.sum((np.sum(weights[labels == c]) / np.sum(weights)) ** 2 for c in unique_classe

        return gini
```

```
        return gini

    def predict(self, X):
        if self.tree is None:
            raise ValueError("Tree not fitted. Call fit() first.")

        return np.array([self._predict_tree(x, self.tree) for x in X])

    def _predict_tree(self, x, node):
        if 'class' in node:
            return node['class']

        if x[node['feature_index']] <= node['threshold']:
            return self._predict_tree(x, node['left'])
        else:
            return self._predict_tree(x, node['right'])



DecisionTree = DecisionTree(max_depth=7)
X_train_array = X_train1.values
y_train_array = y_train1.values
X_test_array = X_test1.values
y_test_array = y_test1.values
DecisionTree.fit(X_train_array,y_train_array)

y_pred = DecisionTree.predict(X_test_array)
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test_array,y_pred)
print(f'Accuracy: {accuracy}')
```

```
/var/folders/8y/cssm2pjn5fq74m2wgx9cslsr0000gn/T/ipykernel_77663/2350801644.py:23: DeprecationWarn
  weighted_gini = 1.0 - np.sum((np.sum(y == c) / num_samples) ** 2 for c in unique_classes)
/var/folders/8y/cssm2pjn5fq74m2wgx9cslsr0000gn/T/ipykernel_77663/2350801644.py:70: DeprecationWarn
  gini = 1.0 - np.sum((np.sum(weights[labels == c]) / np.sum(weights)) ** 2 for c in unique_classe
Accuracy: 0.7362270450751253
```

2.2 Insights drawn (plots, markdown explanations)

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_test_array, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=np.unique(y_test_array),
            yticklabels=np.unique(y_test_array))
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()


def plot_tree_structure(tree, depth=0, feature_names=None, class_names=None):
    if depth == 0:
        print("Tree Structure:")
    indent = "  " * depth
    if 'class' in tree:
        print(f"{indent}Class: {tree['class']}, Count: {tree['count']}")
    else:
        print(f"{indent}Feature {feature_names[tree['feature_index']]} <= {tree['threshold']}")
        plot_tree_structure(tree['left'], depth + 1, feature_names, class_names)
        plot_tree_structure(tree['right'], depth + 1, feature_names, class_names)

# Assuming X_train1.columns contains feature names
plot_tree_structure(DecisionTree.tree, feature_names=X_train1.columns)


def plot_feature_importance(tree, feature_names):
    feature_importance = np.zeros(len(feature_names))

    def traverse_tree(node, importance):
        if 'feature_index' in node:
            importance[node['feature_index']] += 1  # Increment the count for each split
            traverse_tree(node['left'], importance)
            traverse_tree(node['right'], importance)

    traverse_tree(tree, feature_importance)

    plt.bar(range(len(feature_names)), feature_importance, tick_label=feature_names)
    plt.title("Feature Importance")
    plt.xlabel("Feature")
    plt.ylabel("Importance")
    plt.show()

# Assuming X_train1.columns contains feature names
plot_feature_importance(DecisionTree.tree, feature_names=X_train1.columns)
```

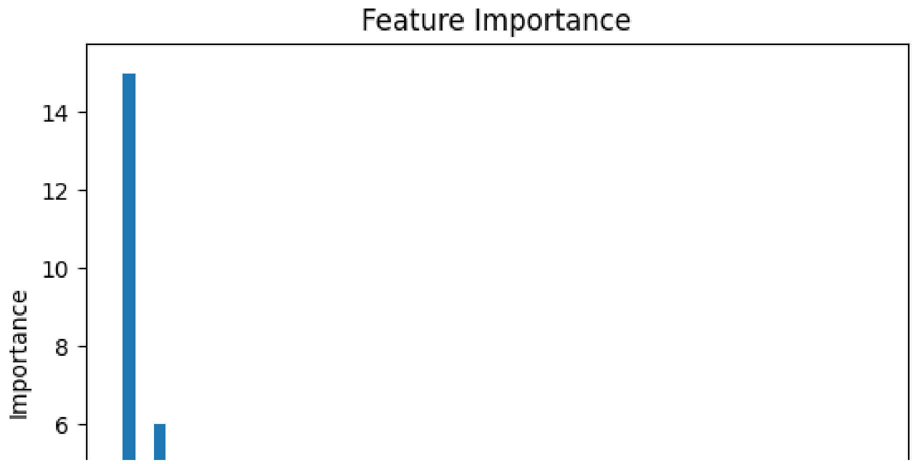



```
Tree Structure:
Feature PC0 <= -0.20352227283318539
  Feature PC0 <= -3.120088799719813
    Feature PC2 <= -0.043469163012343146
      Feature PC35 <= -0.5673786742637349
        Feature PC36 <= 0.3656600456177286
          Feature PC6 <= -1.867830483992641
            Feature PC0 <= -3.6857171869356113
              Class: 0, Count: 1
              Class: 1, Count: 1
            Class: 2, Count: 9
          Feature PC24 <= -0.15240262605721516
            Feature PC0 <= -6.753307063834892
              Class: 2, Count: 1
              Class: 0, Count: 2
            Class: 1, Count: 6
        Feature PC36 <= -1.097704034645209
          Class: 0, Count: 4
          Feature PC6 <= -3.2342399483407758
            Feature PC49 <= -0.17126743340501518
              Class: 0, Count: 11
              Class: 1, Count: 6
            Feature PC33 <= 1.0792433512475477
              Class: 1, Count: 110
              Class: 2, Count: 3
      Feature PC0 <= -4.927804494796778
        Feature PC17 <= 0.598213808764557
          Feature PC21 <= -1.254285549714082
            Class: 1, Count: 3
            Feature PC49 <= -0.23864888893244657
              Class: 1, Count: 5
              Class: 2, Count: 45
          Feature PC40 <= -0.007055363433057346
            Feature PC6 <= -1.2527353068865965
              Class: 1, Count: 1
              Class: 2, Count: 5
            Feature PC0 <= -5.276772613507291
```

```
          Feature PC0 <= -5.276772613507291
            Class: 1, Count: 7
            Class: 2, Count: 1
      Feature PC24 <= 0.0896944496834895
        Feature PC36 <= 0.025139757439457868
          Feature PC35 <= -1.5261344940509427
            Class: 1, Count: 3
            Class: 2, Count: 22
          Feature PC2 <= 1.7775507800908183
            Class: 1, Count: 9
            Class: 2, Count: 4
        Feature PC24 <= 0.7264186970487115
          Feature PC27 <= 0.7632274062415988
            Class: 1, Count: 20
            Class: 2, Count: 2
          Feature PC4 <= -0.7030932024655769
            Class: 1, Count: 2
            Class: 0, Count: 3
  Feature PC7 <= 0.7831886582068653
    Feature PC40 <= -0.49075029020470495
      Feature PC33 <= -0.9093319154360692
        Class: 1, Count: 3
        Feature PC10 <= -0.05668607848173586
          Class: 0, Count: 13
          Feature PC43 <= -0.27928293631930645
            Class: 0, Count: 5
            Class: 1, Count: 8
      Feature PC48 <= -0.6385253954881361
        Feature PC0 <= -2.0397517285492084
          Class: 2, Count: 3
          Class: 0, Count: 2
        Feature PC26 <= -1.1171531472289598
          Feature PC3 <= 2.328145865955763
            Class: 0, Count: 5
            Class: 1, Count: 2
          Feature PC18 <= -1.293638860838211
            Class: 0, Count: 11
            Class: 1, Count: 165
    Feature PC11 <= 0.2682423977820025
      Feature PC3 <= 0.05085061225066428
        Feature PC24 <= -0.3953312184153184
          Feature PC5 <= -2.0026190781666027
            Class: 0, Count: 1
            Class: 1, Count: 8
          Feature PC5 <= 1.3754914551271284
            Class: 0, Count: 32
            Class: 1, Count: 7
        Feature PC49 <= -0.6186657578184241
          Class: 1, Count: 2
          Feature PC0 <= -2.9984278238303514
            Class: 1, Count: 1
            Class: 0, Count: 64
      Feature PC21 <= -0.930436956322043
        Feature PC12 <= -1.0354413301218213
          Feature PC0 <= -2.7638241539764996
            Class: 1, Count: 1
            Class: 2, Count: 1
          Class: 0, Count: 8
        Feature PC47 <= -0.13515808440443378
          Feature PC26 <= -0.0656370904581833
            Class: 1, Count: 5
            Class: 0, Count: 7
          Feature PC2 <= -1.7414974612146845
            Class: 0, Count: 1
            Class: 1, Count: 17
Feature PC0 <= 1.5749849379618615
  Feature PC7 <= 0.2780831208662224
    Feature PC15 <= -0.8004093185277821
      Feature PC41 <= -0.5835395618606524
```

```
          Class: 0, Count: 2
          Class: 1, Count: 16
        Feature PC41 <= -0.26063339003383357
          Feature PC48 <= 0.3276201191469057
            Feature PC1 <= 3.8597049022527656
              Class: 1, Count: 15
              Class: 0, Count: 2
            Class: 0, Count: 3
          Feature PC41 <= 0.06665643000839354
            Feature PC13 <= -1.4368648477234247
              Class: 1, Count: 1
              Class: 0, Count: 28
            Feature PC37 <= 0.040930048286687525
              Class: 0, Count: 22
              Class: 1, Count: 15
      Feature PC33 <= 0.5442886973512711
        Feature PC13 <= -1.0380980880134125
          Feature PC24 <= 0.11433390471976372
            Feature PC47 <= -0.07345469913638066
              Class: 0, Count: 5
              Class: 1, Count: 2
            Class: 1, Count: 4
          Feature PC18 <= -0.5547258591671509
            Feature PC41 <= 0.584562605612093
              Class: 0, Count: 27
              Class: 1, Count: 3
            Feature PC2 <= 4.453349597897949
              Class: 0, Count: 69
              Class: 0, Count: 2
        Feature PC16 <= -0.7474943834943428
          Class: 0, Count: 3
          Feature PC15 <= 1.1391643517327348
            Class: 1, Count: 6
            Class: 2, Count: 2
    Feature PC0 <= 3.2452846886553095
      Feature PC2 <= 0.4260315653856368
        Feature PC8 <= -2.3961288862004917
          Feature PC0 <= 3.178391546792424
            Class: 1, Count: 3
            Class: 0, Count: 1
          Feature PC47 <= -0.2553091550879119
            Feature PC0 <= 2.9850451535342977
              Class: 0, Count: 15
              Class: 1, Count: 5
            Feature PC7 <= 2.21044748289066
              Class: 0, Count: 108
              Class: 0, Count: 2
        Feature PC39 <= 0.4443428111152416
          Feature PC34 <= 0.6130317045875795
            Feature PC29 <= -1.390144753092316
              Class: 1, Count: 2
              Class: 0, Count: 56
            Feature PC7 <= -0.11783210590490162
              Class: 1, Count: 7
              Class: 0, Count: 3
          Feature PC2 <= 2.3676280816939577
            Class: 1, Count: 6
            Class: 0, Count: 1
      Feature PC13 <= -2.4538283738752162
        Feature PC0 <= 4.902256387568408
          Class: 1, Count: 2
          Class: 0, Count: 1
        Feature PC42 <= 1.0799883602390004
          Feature PC6 <= -3.998047387347777
            Feature PC0 <= 3.705734762715251
              Class: 1, Count: 1
              Class: 0, Count: 1
            Feature PC8 <= -2.5632193891651642
```

```
        Class: 0, Count: 3
        Class: 0, Count: 292
    Class: 1, Count: 1
```




```
from sklearn.metrics import classification_report

report = classification_report(y_test1, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.87      0.81      0.84       380
           1       0.56      0.67      0.61       175
           2       0.46      0.36      0.41        44

    accuracy                           0.74       599
   macro avg       0.63      0.61      0.62       599
weighted avg       0.75      0.74      0.74       599
```

3. Adaboost

3.1 Implementation of the Model

```
import numpy as np
class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y, sample_weights=None):
        self.tree = self._build_tree(X, y, sample_weights, depth=0)

    def _build_tree(self, X, y, sample_weights, depth):
        num_samples, num_features = X.shape
        unique_classes, class_counts = np.unique(y, return_counts=True)
        default_class = unique_classes[np.argmax(class_counts)]

        # Base cases
        if depth == self.max_depth or len(np.unique(y)) == 1:
            return {'class': default_class, 'count': len(y)}

        # If sample weights are provided, compute the weighted Gini impurity
        if sample_weights is not None:
            total_weight = np.sum(sample_weights)
            weighted_gini = 1.0 - np.sum((np.sum(sample_weights[y == c]) / total_weight) ** 2 for c in
        else:
            weighted_gini = 1.0 - np.sum((np.sum(y == c) / num_samples) ** 2 for c in unique_classes)

        best_split = {'feature_index': None, 'threshold': None, 'gini': weighted_gini}

        # Iterate through each feature and find the best split
        for feature_index in range(num_features):
            unique_values = np.unique(X[:, feature_index])
            for threshold in unique_values:
                left_mask = X[:, feature_index] <= threshold
                right_mask = ~left_mask

                if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
                    continue

                left_gini = self._calculate_gini(y[left_mask], sample_weights[left_mask] if sample_wei
                right_gini = self._calculate_gini(y[right_mask], sample_weights[right_mask] if sample_

                weighted_avg_gini = (np.sum(left_mask) * left_gini + np.sum(right_mask) * right_gini)

                if weighted_avg_gini < best_split['gini']:
                    best_split = {
                        'feature_index': feature_index,
                        'threshold': threshold,
                        'gini': weighted_avg_gini,
                        'left_mask': left_mask,
                        'right_mask': right_mask
                    }

        if best_split['feature_index'] is None:
            return {'class': default_class, 'count': len(y)}

        left_subtree = self._build_tree(X[best_split['left_mask']], y[best_split['left_mask']], sample
        right_subtree = self._build_tree(X[best_split['right_mask']], y[best_split['right_mask']], sam

        return {
            'feature_index': best_split['feature_index'],
            'threshold': best_split['threshold'],
            'left': left_subtree,
            'right': right_subtree
```

```
        }

    def _calculate_gini(self, labels, weights=None):
        num_samples = len(labels)
        if weights is None:
            weights = np.ones(num_samples) / num_samples

        unique_classes, class_counts = np.unique(labels, return_counts=True)
        gini = 1.0 - np.sum((np.sum(weights[labels == c]) / np.sum(weights)) ** 2 for c in unique_clas

        return gini

    def predict(self, X):
        if self.tree is None:
            raise ValueError("Tree not fitted. Call fit() first.")

        return np.array([self._predict_tree(x, self.tree) for x in X])

    def _predict_tree(self, x, node):
        if 'class' in node:
            return node['class']

        if x[node['feature_index']] <= node['threshold']:
            return self._predict_tree(x, node['left'])
        else:
            return self._predict_tree(x, node['right'])
from sklearn.metrics import accuracy_score

# Convert DataFrames to numpy arrays
X_train_array = X_train2.values
y_train_array = y_train2.values
X_test_array = X_test2.values
y_test_array = y_test2.values

# Number of weak learners (decision stumps)
n_estimators = 50

# Initialize weights for data points
weights = np.ones(len(X_train_array)) / len(X_train_array)

# List to store weak learners
weak_learners = []

for _ in range(n_estimators):
    # Create a weak learner (decision stump)
    weak_learner = DecisionTree(max_depth=1)

    # Train the weak learner on the weighted data
    weak_learner.fit(X_train_array, y_train_array, sample_weights=weights)

    # Make predictions on the training set
    predictions = weak_learner.predict(X_train_array)

    # Calculate the error of the weak learner
    error = np.sum(weights * (predictions != y_train_array)) / np.sum(weights)

    # Calculate the weight of the weak learner
    alpha = 0.5 * np.log((1 - error) / error)

    # Update weights based on the correctness of predictions
    weights = weights * np.exp(-alpha * y_train_array * predictions)
    weights /= np.sum(weights)
```

```
        # Save the weak learner and its weight
        weak_learners.append((weak_learner, alpha))

    # Make predictions on the test set
    final_predictions = np.zeros(len(X_test_array))

    for learner, alpha in weak_learners:
        learner_predictions = alpha * learner.predict(X_test_array)
        final_predictions += learner_predictions

    # Convert final predictions to binary values
    final_predictions = np.sign(final_predictions)

    # Evaluate the accuracy
    accuracy1 = accuracy_score(y_test_array, final_predictions)
    print(f'Accuracy: {accuracy1}')
```

```
/var/folders/8y/cssm2pjn5fq74m2wgx9cslsr0000gn/T/ipykernel_77663/3805669546.py:22: DeprecationWarn
  weighted_gini = 1.0 - np.sum((np.sum(sample_weights[y == c]) / total_weight) ** 2 for c in uniqu
/var/folders/8y/cssm2pjn5fq74m2wgx9cslsr0000gn/T/ipykernel_77663/3805669546.py:71: DeprecationWarn
  gini = 1.0 - np.sum((np.sum(weights[labels == c]) / np.sum(weights)) ** 2 for c in unique_classe
Accuracy: 0.7212020033388982
```

3.2 Insights drawn (plots, markdown explanations)

```
import matplotlib.pyplot as plt

def calculate_error(predictions, true_labels, weights=None):
    if weights is None:
        weights = np.ones(len(true_labels)) / len(true_labels)
    return np.sum(weights * (predictions != true_labels)) / np.sum(weights)

# Training Error Plot
train_errors = []
for learner, alpha in weak_learners:
    train_predictions = sum(alpha * learner.predict(X_train_array) for learner, alpha in weak_learners
    train_error = calculate_error(train_predictions, y_train_array)
    train_errors.append(train_error)

plt.figure(figsize=(10, 5))
plt.plot(range(1, n_estimators + 1), train_errors, marker='o')
plt.title('Training Error vs. Number of Weak Learners')
plt.xlabel('Number of Weak Learners')
plt.ylabel('Training Error')
plt.show()

# Test Error Plot
test_errors = []
for num_learners in range(1, n_estimators + 1):
    test_predictions = sum(alpha * learner.predict(X_test_array) for learner, alpha in weak_learners[:
    test_error = calculate_error(test_predictions, y_test_array)
    test_errors.append(test_error)

plt.figure(figsize=(10, 5))
plt.plot(range(1, n_estimators + 1), test_errors, marker='o', color='red')
plt.title('Test Error vs. Number of Weak Learners')
plt.xlabel('Number of Weak Learners')
plt.ylabel('Test Error')
plt.show()

# Weak Learner Weight Plot
learner_weights = [alpha for learner, alpha in weak_learners]

plt.figure(figsize=(10, 5))
plt.bar(range(1, n_estimators + 1), learner_weights)
plt.title('Weak Learner Weights')
plt.xlabel('Weak Learner')
plt.ylabel('Weight')
plt.show()

from sklearn.metrics import roc_curve, auc, precision_recall_curve, confusion_matrix

# ... (Previous code)

# ROC Curve
from sklearn.preprocessing import label_binarize

def plot_roc_curve_multiclass(X_test, y_test, weak_learners):
    classes = np.unique(y_test)
    n_classes = len(classes)
    final_predictions_proba = np.zeros((len(y_test), n_classes))

    for i, class_label in enumerate(classes):
        # Binarize the labels for the current class
        y_binary = label_binarize(y_test, classes=classes)

        # Sum the weighted predictions for the current class
```