BITS F464 - Semester 1 - MACHINE LEARNING

ASSIGNMENT 2 – DECISION TREES AND SUPPORT VECTOR MACHINES

Team number: 13

Full names of all students in the team: ANIRUDH BAGALKOTKER, KARTIK PANDEY, ADWAIT KULKARNI, JOY SINHA

Id number of all students in the team: 2021A7PS2682H, 2021A7PS2574H, 2021A7PS2995H, 2021A8PS1606H

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

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

Import Dependencies and Load Dataset

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.graph_objs as go
```

In [ ]:
```
# Load the dataset

# Add column names
column_names = [
    "state", "county", "community", "communityname", "fold", "population"
    "racepctblack", "racePctWhite", "racePctAsian", "racePctHisp", "agePc
    "agePct12t29", "agePct16t24", "agePct65up", "numbUrban", "pctUrban",
    "pctWWage", "pctWFarmSelf", "pctWInvInc", "pctWSocSec", "pctWPubAsst"
    "medFamInc", "perCapInc", "whitePerCap", "blackPerCap", "indianPerCap
    "OtherPerCap", "HispPerCap", "NumUnderPov", "PctPopUnderPov", "PctLes
    "PctNotHSGrad", "PctBSorMore", "PctUnemployed", "PctEmploy", "PctEmpl
    "PctEmplProfServ", "PctOccupManu", "PctOccupMgmtProf", "MalePctDivorc
    "MalePctNevMarr", "FemalePctDiv", "TotalPctDiv", "PersPerFam", "PctFa
    "PctKids2Par", "PctYoungKids2Par", "PctTeen2Par", "PctWorkMomYoungKid
    "NumIlleg", "PctIlleg", "NumImmig", "PctImmigRecent", "PctImmigRec5",
    "PctImmigRec10", "PctRecentImmig", "PctRecImmig5", "PctRecImmig8", "P
    "PctSpeakEnglOnly", "PctNotSpeakEnglWell", "PctLargHouseFam", "PctLar
    "PersPerOccupHous", "PersPerOwnOccHous", "PersPerRentOccHous", "PctPe
    "PctPersDenseHous", "PctHousLess3BR", "MedNumBR", "HousVacant", "PctH
    "PctHousOwnOcc", "PctVacantBoarded", "PctVacMore6Mos", "MedYrHousBuil
```

```
    "PctHousNoPhone", "PctWOFullPlumb", "OwnOccLowQuart", "OwnOccMedVal",
    "RentLowQ", "RentMedian", "RentHighQ", "MedRent", "MedRentPctHousInc"
    "MedOwnCostPctInc", "MedOwnCostPctIncNoMtg", "NumInShelters", "NumStr
    "PctForeignBorn", "PctBornSameState", "PctSameHouse85", "PctSameCity8
    "PctSameState85", "LemasSwornFT", "LemasSwFTPerPop", "LemasSwFTFieldO
    "LemasSwFTFieldPerPop", "LemasTotalReq", "LemasTotReqPerPop", "PolicR
    "PolicPerPop", "RacialMatchCommPol", "PctPolicWhite", "PctPolicBlack"
    "PctPolicHisp", "PctPolicAsian", "PctPolicMinor", "OfficAssgnDrugUnit
    "NumKindsDrugsSeiz", "PolicAveOTWorked", "LandArea", "PopDens", "PctU
    "PolicCars", "PolicOperBudg", "LemasPctPolicOnPatr", "LemasGangUnitDe
    "LemasPctOfficDrugUn", "PolicBudgPerPop", "ViolentCrimesPerPop"
]

# Define missing values
missing_values = ['?']

# Read the dataset
data = pd.read_csv('communities.data', header=None, names=column_names, n

data.head()
```

Out[ ]:

	state	county	community	communityname	fold	population	householdsize	race
0	8	NaN	NaN	Lakewoodcity	1	0.19	0.33	
1	53	NaN	NaN	Tukwilacity	1	0.00	0.16	
2	24	NaN	NaN	Aberdeentown	1	0.00	0.42	
3	34	5.0	81440.0	Willingborotownship	1	0.04	0.77	
4	42	95.0	6096.0	Bethlehemtownship	1	0.01	0.55	

5 rows × 128 columns

In [ ]:

```
# Remove non-predictive columns
df = data.drop(["state", "county", "community", "communityname", "fold"],
df.head()
```

Out[ ]:

	population	householdsize	racepctblack	racePctWhite	racePctAsian	racePctHisp	ag
0	0.19	0.33	0.02	0.90	0.12	0.17	
1	0.00	0.16	0.12	0.74	0.45	0.07	
2	0.00	0.42	0.49	0.56	0.17	0.04	
3	0.04	0.77	1.00	0.08	0.12	0.10	
4	0.01	0.55	0.02	0.95	0.09	0.05	

5 rows × 123 columns

Exploratory Data Analysis

In [ ]:

```
samples, features = np.shape(df)
df.shape
```

Out[ ]: (1994, 123)

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1994 entries, 0 to 1993
Columns: 123 entries, population to ViolentCrimesPerPop
dtypes: float64(123)
memory usage: 1.9 MB
```

In [ ]:
```
df.describe()
```

Out[ ]:

	population	householdsize	racepctblack	racePctWhite	racePctAsian	racePct
count	1994.000000	1994.000000	1994.000000	1994.000000	1994.000000	1994.00C
mean	0.057593	0.463395	0.179629	0.753716	0.153681	0.14
std	0.126906	0.163717	0.253442	0.244039	0.208877	0.23
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.00C
25%	0.010000	0.350000	0.020000	0.630000	0.040000	0.01C
50%	0.020000	0.440000	0.060000	0.850000	0.070000	0.04C
75%	0.050000	0.540000	0.230000	0.940000	0.170000	0.16C
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.00C

8 rows × 123 columns

In [ ]:
```
df['ViolentCrimesPerPop'].value_counts(normalize=True)
```

Out[ ]:

```
ViolentCrimesPerPop
0.03    0.052156
0.04    0.046138
0.06    0.043129
0.05    0.040120
0.02    0.037111
          ...
0.79    0.001003
0.77    0.000502
0.89    0.000502
0.94    0.000502
0.96    0.000502
Name: proportion, Length: 98, dtype: float64
```

In [ ]:
```
df.drop('ViolentCrimesPerPop', axis=1).skew()
```

```
Out[ ]: population              5.063957
        householdsize           0.981300
        racepctblack            1.863340
        racePctWhite           -1.300489
        racePctAsian            2.604395
                                  ...
        PolicOperBudg           4.153317
        LemasPctPolicOnPatr    -1.589877
        LemasGangUnitDeploy     0.221361
        LemasPctOfficDrugUn     2.554246
        PolicBudgPerPop         3.222583
        Length: 122, dtype: float64
```

```
In [ ]: df.corr()
```

Out[ ]:

	population	householdsize	racepctblack	racePctWhite	racePctAsi
population	1.000000	-0.046148	0.231178	-0.300845	0.1816
householdsize	-0.046148	1.000000	-0.067109	-0.235907	0.2019
racepctblack	0.231178	-0.067109	1.000000	-0.794389	-0.1067
racePctWhite	-0.300845	-0.235907	-0.794389	1.000000	-0.2702
racePctAsian	0.181603	0.201996	-0.106738	-0.270266	1.0000
...	...	...	...	...	
LemasPctPolicOnPatr	-0.080482	-0.017972	-0.168434	0.125223	0.0690
LemasGangUnitDeploy	0.100012	-0.000784	0.022388	-0.078552	0.1395
LemasPctOfficDrugUn	0.466352	-0.094368	0.260793	-0.276234	0.1018
PolicBudgPerPop	-0.046494	-0.152603	0.045311	-0.014957	-0.0247
ViolentCrimesPerPop	0.367157	-0.034923	0.631264	-0.684770	0.0376

123 rows × 123 columns

```
In [ ]: fig = go.Figure(go.Heatmap(z=df.corr(), x=df.corr().columns.tolist(), y=d
        fig.show()
```

Data Preprocessing

```
In [ ]: # Check if there are any missing values
        missing_values = df.isnull().sum()
        columns_with_missing_values = missing_values[missing_values > (samples //

        # Print the number of missing values for each column
        for column in columns_with_missing_values:
            print(f"{column}: {missing_values[column]} missing values")

        # Drop columns with too many missing values
        df = df.drop(columns=columns_with_missing_values)
```

```
LemasSwornFT: 1675 missing values
LemasSwFTPerPop: 1675 missing values
LemasSwFTFieldOps: 1675 missing values
LemasSwFTFieldPerPop: 1675 missing values
LemasTotalReq: 1675 missing values
LemasTotReqPerPop: 1675 missing values
PolicReqPerOffic: 1675 missing values
PolicPerPop: 1675 missing values
RacialMatchCommPol: 1675 missing values
PctPolicWhite: 1675 missing values
PctPolicBlack: 1675 missing values
PctPolicHisp: 1675 missing values
PctPolicAsian: 1675 missing values
PctPolicMinor: 1675 missing values
OfficAssgnDrugUnits: 1675 missing values
NumKindsDrugsSeiz: 1675 missing values
PolicAveOTWorked: 1675 missing values
PolicCars: 1675 missing values
PolicOperBudg: 1675 missing values
LemasPctPolicOnPatr: 1675 missing values
LemasGangUnitDeploy: 1675 missing values
PolicBudgPerPop: 1675 missing values
```

In [ ]:
```
# Handle missing values (if any) by replacing them with the mean
df.fillna(df.mean(), inplace=True)
df.head()
```

Out[ ]:

	population	householdsize	racepctblack	racePctWhite	racePctAsian	racePctHisp	ag
0	0.19	0.33	0.02	0.90	0.12	0.17	
1	0.00	0.16	0.12	0.74	0.45	0.07	
2	0.00	0.42	0.49	0.56	0.17	0.04	
3	0.04	0.77	1.00	0.08	0.12	0.10	
4	0.01	0.55	0.02	0.95	0.09	0.05	

5 rows × 101 columns

In [ ]:
```
df.describe()
```

Out[ ]:

	population	householdsize	racepctblack	racePctWhite	racePctAsian	racePct
count	1994.000000	1994.000000	1994.000000	1994.000000	1994.000000	1994.000
mean	0.057593	0.463395	0.179629	0.753716	0.153681	0.14
std	0.126906	0.163717	0.253442	0.244039	0.208877	0.23
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	0.010000	0.350000	0.020000	0.630000	0.040000	0.010
50%	0.020000	0.440000	0.060000	0.850000	0.070000	0.040
75%	0.050000	0.540000	0.230000	0.940000	0.170000	0.160
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000

8 rows × 101 columns

In [ ]:

```
# Performing PCA on the dataset
target_variable = df['ViolentCrimesPerPop']
features = df.drop(['ViolentCrimesPerPop'], axis=1)

# Standardize the features
standardized_features = (features - features.mean()) / features.std()

# Calculate the covariance matrix
cov_matrix = np.cov(standardized_features, rowvar=False)

# Calculate the eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

# Sort eigenvalues and corresponding eigenvectors in descending order
sorted_indices = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

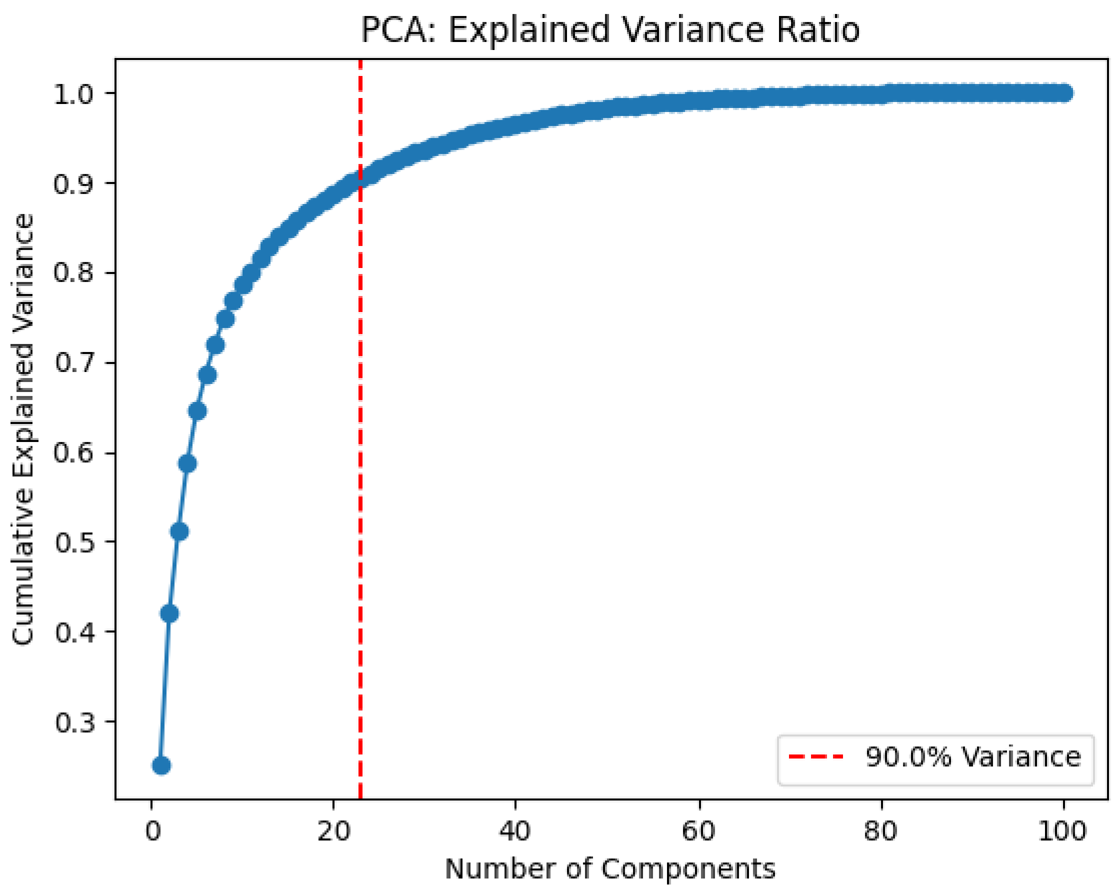
In [ ]:

```
# Calculate the explained variance ratio
explained_variance_ratio = eigenvalues / np.sum(eigenvalues)

# Cumulative explained variance
cumulative_explained_variance = np.cumsum(explained_variance_ratio)

# Find the number of components that explain at least 90% of the variance
desired_explained_variance = 0.90
num_components = np.argmax(cumulative_explained_variance >= desired_expla

# Plot the explained variance ratio
plt.plot(range(1, len(explained_variance_ratio) + 1), cumulative_explaine
plt.axvline(x=num_components, color='r', linestyle='--', label=f'{desired
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA: Explained Variance Ratio')
plt.legend()
plt.show()
```



In [ ]:
```
# Project the original data onto the selected number of components
selected_eigenvectors = eigenvectors[:, :num_components]
pca_result = np.dot(standardized_features, selected_eigenvectors)

# Create a DataFrame with the principal components and the target variabl
df_pca = pd.DataFrame(pca_result, columns=[f'PC{i + 1}' for i in range(nu
df_pca['ViolentCrimesPerPop'] = target_variable
df_pca.head()
df_pca.to_csv('crimes.csv', index=False)
```

Generate Random Test and Train Splits

In [ ]:
```
seed = 420
train_fraction = 0.8
train = df_pca.sample(frac=train_fraction, random_state=seed)
test = df_pca.drop(train.index)
```

In [ ]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1595 entries, 493 to 1405
Data columns (total 24 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   PC1                 1595 non-null   float64
 1   PC2                 1595 non-null   float64
 2   PC3                 1595 non-null   float64
 3   PC4                 1595 non-null   float64
 4   PC5                 1595 non-null   float64
 5   PC6                 1595 non-null   float64
 6   PC7                 1595 non-null   float64
 7   PC8                 1595 non-null   float64
 8   PC9                 1595 non-null   float64
 9   PC10                1595 non-null   float64
 10  PC11                1595 non-null   float64
 11  PC12                1595 non-null   float64
 12  PC13                1595 non-null   float64
 13  PC14                1595 non-null   float64
 14  PC15                1595 non-null   float64
 15  PC16                1595 non-null   float64
 16  PC17                1595 non-null   float64
 17  PC18                1595 non-null   float64
 18  PC19                1595 non-null   float64
 19  PC20                1595 non-null   float64
 20  PC21                1595 non-null   float64
 21  PC22                1595 non-null   float64
 22  PC23                1595 non-null   float64
 23  ViolentCrimesPerPop 1595 non-null   float64
dtypes: float64(24)
memory usage: 311.5 KB
```

In [ ]:

```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 399 entries, 2 to 1989
Data columns (total 24 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   PC1                 399 non-null    float64
 1   PC2                 399 non-null    float64
 2   PC3                 399 non-null    float64
 3   PC4                 399 non-null    float64
 4   PC5                 399 non-null    float64
 5   PC6                 399 non-null    float64
 6   PC7                 399 non-null    float64
 7   PC8                 399 non-null    float64
 8   PC9                 399 non-null    float64
 9   PC10                399 non-null    float64
 10  PC11                399 non-null    float64
 11  PC12                399 non-null    float64
 12  PC13                399 non-null    float64
 13  PC14                399 non-null    float64
 14  PC15                399 non-null    float64
 15  PC16                399 non-null    float64
 16  PC17                399 non-null    float64
 17  PC18                399 non-null    float64
 18  PC19                399 non-null    float64
 19  PC20                399 non-null    float64
 20  PC21                399 non-null    float64
 21  PC22                399 non-null    float64
 22  PC23                399 non-null    float64
 23  ViolentCrimesPerPop  399 non-null    float64
dtypes: float64(24)
memory usage: 77.9 KB
```

In [ ]:

```
# Assuming 'ViolentCrimesPerPop' is the column you want to predict
X_train = train.drop('ViolentCrimesPerPop', axis=1)  # Features for train
y_train = train['ViolentCrimesPerPop']  # Target for training

X_test = test.drop('ViolentCrimesPerPop', axis=1)  # Features for testing
y_test = test['ViolentCrimesPerPop']  # Target for testing

# Convert labels to numpy array for applying ML Models
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()
```

In [ ]:

```
def accuracy(pred, y_test, threshold=0.5):
    # Calculate the standard deviation of the true values
    y_std = np.std(y_test)

    # Check if the absolute difference is below the threshold (multiple o
    correct_predictions = np.abs(pred - y_test) < threshold * y_std

    # Calculate accuracy as the percentage of correct predictions
    accu = 100 * correct_predictions.mean()

    return accu
```

2. Decision tree model with entropy implementation

2.1 Implementation of the Model

In [ ]:
```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None, rig
        ''' constructor '''

        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain

        # for leaf node
        self.value = value

class DecisionTreeClassifier():
    def __init__(self, min_samples_split=2, max_depth=2):
        ''' constructor '''

        # initialize the root of the tree
        self.root = None

        # stopping conditions
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        ''' recursive function to build the tree '''

        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)

        # split until stopping conditions are met
        if num_samples>=self.min_samples_split and curr_depth<=self.max_d
            # find the best split
            best_split = self.get_best_split(dataset, num_samples, num_fe
            # check if information gain is positive
            if best_split["info_gain"]>0:
                # recur left
                left_subtree = self.build_tree(best_split["dataset_left"]
                # recur right
                right_subtree = self.build_tree(best_split["dataset_right
                # return decision node
                return Node(best_split["feature_index"], best_split["thre
                            left_subtree, right_subtree, best_split["info

        # compute leaf node
        leaf_value = self.calculate_leaf_value(Y)
        # return leaf node
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        ''' function to find the best split '''

        # dictionary to store the best split
        best_split = {}
        max_info_gain = -float("inf")
```

```
        # loop over all the features
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            # loop over all the feature values present in the data
            for threshold in possible_thresholds:
                # get current split
                dataset_left, dataset_right = self.split(dataset, feature
                # check if childs are not null
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:,
                    # compute information gain
                    curr_info_gain = self.information_gain(y, left_y, rig
                    # update the best split if needed
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain

        # return best split
        return best_split

    def split(self, dataset, feature_index, threshold):
        ''' function to split the data '''

        dataset_left = np.array([row for row in dataset if row[feature_in
        dataset_right = np.array([row for row in dataset if row[feature_i
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
        ''' function to compute information gain '''

        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        if mode=="gini":
            gain = self.gini_index(parent) - (weight_l*self.gini_index(l_
        else:
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child)
        return gain

    def entropy(self, y):
        ''' function to compute entropy '''

        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        ''' function to compute 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 calculate_leaf_value(self, Y):
        ''' function to compute leaf node '''

        Y = list(Y)
        return max(Y, key=Y.count)

    def print_tree(self, tree=None, indent=""):
        ''' function to print the tree '''

        if not tree:
            tree = self.root

        if tree.value is not None:
            print(f"{indent}Leaf Node: Class {tree.value}")

        else:
            print(f"{indent}Node: PC{tree.feature_index} <= {tree.thresho
            print(f"{indent}left:")
            self.print_tree(tree.left, indent + "  ")
            print(f"{indent}right:")
            self.print_tree(tree.right, indent + "  ")

    def fit(self, X, Y):
        ''' function to train the tree '''

        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)

    def predict(self, X):
        ''' function to predict new dataset '''

        preditions = [self.make_prediction(x, self.root) for x in X.value
        return preditions

    def make_prediction(self, x, tree):
        ''' function to predict a single data point '''

        if (tree.value != None):
            return tree.value
        feature_val = x[tree.feature_index]
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)

classifier = DecisionTreeClassifier(min_samples_split=2, max_depth=num_co
```

In [ ]:
```
classifier.fit(X_train,pd.DataFrame(y_train))
```

In [ ]:
```
Y_pred = np.array(classifier.predict(X_test))
acc = accuracy(y_test, Y_pred)
acc
```

Out[ ]: 57.89473684210527

2.2 Insights drawn (plots, markdown explanations)

Decision Tree Structure

Lets take a look at the decision tree structure.

In [ ]:
```
classifier.print_tree()
```

```
Node: PC0 <= -1.903406546494383 [Info Gain: 0.3068691753511583]
left:
  Node: PC0 <= -5.404072525388851 [Info Gain: 0.2795796614091701]
  left:
    Node: PC10 <= -0.622998399766284 [Info Gain: 0.5069499925013172]
    left:
      Node: PC6 <= 1.304111411122921 [Info Gain: 0.7324452692561572]
      left:
        Node: PC6 <= -0.5355053161787456 [Info Gain: 0.8429038335138657]
        left:
          Node: PC5 <= 0.5314709382419304 [Info Gain: 0.9656361333706105]
          left:
            Node: PC8 <= -0.723970264374518 [Info Gain: 1.000000000000000
4]
            left:
              Node: PC0 <= -6.139118842483037 [Info Gain: 0.59167277858232
73]
              left:
                Node: PC5 <= -0.5463861793856238 [Info Gain: 0.45914791702
72448]
                left:
                  Node: PC3 <= 1.684765404111435 [Info Gain: 0.91829583405
44896]
                  left:
                    Leaf Node: Class 0.66
                  right:
                    Leaf Node: Class 0.36
                right:
                  Leaf Node: Class 0.36
              right:
                Leaf Node: Class 0.93
            right:
              Node: PC0 <= -7.58865331960808 [Info Gain: 0.985228136034251
6]
              left:
                Node: PC0 <= -8.937900603459456 [Info Gain: 0.918295834054
4894]
                left:
                  Leaf Node: Class 0.61
                right:
                  Node: PC0 <= -7.716220490292774 [Info Gain: 1.0]
                  left:
                    Leaf Node: Class 0.45
                  right:
                    Leaf Node: Class 0.33
              right:
                Node: PC0 <= -6.510808682197184 [Info Gain: 1.0]
                left:
                  Node: PC0 <= -6.8523344043417795 [Info Gain: 1.0]
                  left:
                    Leaf Node: Class 0.72
                  right:
                    Leaf Node: Class 0.54
                right:
                  Node: PC0 <= -5.8161357646481004 [Info Gain: 1.0]
                  left:
                    Leaf Node: Class 0.51
                  right:
                    Leaf Node: Class 0.16
          right:
```

```
            Leaf Node: Class 0.0
    right:
      Node: PC1 <= 3.9984587146391304 [Info Gain: 1.0]
      left:
        Node: PC1 <= 3.7855241130784365 [Info Gain: 1.0]
        left:
          Leaf Node: Class 0.07
        right:
          Node: PC0 <= 11.788976055157395 [Info Gain: 1.0]
          left:
            Leaf Node: Class 0.05
          right:
            Leaf Node: Class 0.01
      right:
        Node: PC0 <= 8.440018853654108 [Info Gain: 1.0]
        left:
          Node: PC0 <= 7.792827973550439 [Info Gain: 1.0]
          left:
            Leaf Node: Class 0.45
          right:
            Leaf Node: Class 0.03
        right:
          Node: PC0 <= 8.885360375132475 [Info Gain: 1.0]
          left:
            Leaf Node: Class 0.17
          right:
            Leaf Node: Class 0.13
```

Decision Tree Implementation

1. `Node` Class

The `Node` class represents a node in the decision tree. It has attributes for decision nodes (`feature_index` , `threshold` , `left` , `right` , `info_gain`) and leaf nodes (`value`).

2. `DecisionTreeClassifier` Class

The Decision Tree Classifier is implemented with a recursive binary tree structure. It builds the tree by selecting the best feature and threshold for splitting based on information gain. The tree stops growing when a specified depth or minimum samples for splitting is reached. Leaf nodes represent the majority class, and the structure is printed for interpretability. The classifier is trained using the fit method and makes predictions for new datasets.

Initialization:

The class is initialized with parameters `min_samples_split` and `max_depth` to control the tree-building process.

Methods:

1. **`build_tree`** : Recursive tree construction.
2. **`get_best_split`** : Finds best split based on Entropy.

3. `split` : Divides data based on feature threshold.

4. `information_gain` : Computes information gain.

5. `calculate_leaf_value` : Determines leaf value.

6. `print_tree` : Prints tree structure.

7. `fit` : Trains the tree.

8. `predict` : Makes predictions.

9. `make_prediction` : Predicts a single data point.

Usage

- An instance of DecisionTreeClassifier is created with specified parameters.
- The fit method is called to train the tree on the training data.
- Predictions are made using the predict method on the test data.
- Accuracy is calculated using a simple accuracy calculating function.

Interpretation of Accuracy

In [ ]:
```
print(f"The Accuracy of the Decision Tree Classifier is: {acc}%")
```

```
The Accuracy of the Decision Tree Classifier is: 57.89473684210527%
```

Improvement Suggestions

- Tune hyperparameters.
- Analyze feature importance.
- Use cross-validation.
- Iteratively refine based on insights.
- Advanced Tree algorithms improve implementation and optimization

3. Adaboost

3.1 Implementation of the Model

In [ ]:
```
# Decision stump used as weak classifier
class DecisionStump():
    def __init__(self):
        self.polarity = 1
        self.feature_idx = None
        self.threshold = None
        self.alpha = None

    def predict(self, X):
        n_samples = X.shape[0]
        X_column = X[:, self.feature_idx]
        predictions = np.ones(n_samples)
```

```
        if self.polarity == 1:
            predictions[X_column < self.threshold] = -1
        else:
            predictions[X_column > self.threshold] = -1

        return predictions


class Adaboost():

    def __init__(self, n_clf=2):
        self.n_clf = n_clf

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Initialize weights to 1/N
        w = np.full(n_samples, (1 / n_samples))

        self.clfs = []
        # Iterate through classifiers
        for _ in range(self.n_clf):
            clf = DecisionStump()

            min_error = float('inf')
            # greedy search to find best threshold and feature
            for feature_i in range(n_features):
                X_column = X[:, feature_i]
                thresholds = np.unique(X_column)

                for threshold in thresholds:
                    # predict with polarity 1
                    p = 1
                    predictions = np.ones(n_samples)
                    predictions[X_column < threshold] = -1

                    # Error = sum of weights of misclassified samples
                    misclassified = w[y != predictions]
                    error = sum(misclassified)

                    if error > 0.5:
                        error = 1 - error
                        p = -1

                    # store the best configuration
                    if error < min_error:
                        clf.polarity = p
                        clf.threshold = threshold
                        clf.feature_idx = feature_i
                        min_error = error

            # calculate alpha
            EPS = 1e-10
            clf.alpha = 0.5 * np.log((1.0 - min_error + EPS) / (min_error

            # calculate predictions and update weights
            predictions = clf.predict(X)

            w *= np.exp(-clf.alpha * y * predictions)
            # Normalize to one
```

```
            w /= np.sum(w)

            # Save classifier
            self.clfs.append(clf)

    def predict(self, X):
        clf_preds = [clf.alpha * clf.predict(X) for clf in self.clfs]
        y_pred = np.sum(clf_preds, axis=0)
        y_pred = np.sign(y_pred)

        return y_pred

classifier = Adaboost()
```

In [ ]:
```
classifier.fit(X_train.values, y_train)

y_pred = classifier.predict(X_test.values)
```

In [ ]:
```
acc = accuracy(y_test, y_pred, threshold=1)
acc
```

Out[ ]:
```
56.390977443609025
```

3.2 Insights drawn (plots, markdown explanations)

AdaBoost Implementation

1. `DecisionStump` Class

The `DecisionStump` class represents a weak classifier (a decision stump). It has attributes for polarity, feature index, threshold, and alpha.

2. `Adaboost` Class

Adaboost, short for Adaptive Boosting, is an ensemble learning algorithm that combines weak classifiers to create a strong classifier. In this implementation, weak classifiers are decision stumps (simple decision trees with a single split). Adaboost iteratively trains decision stumps(`DecisionStump`), adjusting their weights based on their performance. The `Adaboost` Class iteratively selects the best feature and threshold for each weak classifier, assigning higher weights to misclassified samples. The final prediction is a weighted combination of individual weak classifiers. The algorithm adapts by adjusting weights and focuses on difficult-to-classify instances. The resulting ensemble achieves better accuracy than individual classifiers.

Initialization:

The class is initialized with the number of weak classifiers (`n_clf`).

Methods:

1. **`fit`** : Trains the AdaBoost ensemble by iteratively training weak classifiers.

2. `predict` : Makes predictions using the ensemble.

Usage

- An instance of the `AdaBoost` class is created with the specified number of weak classifiers.
- The `fit` method is called to train the AdaBoost ensemble on the training data.
- Predictions are made using the `predict` method on the test data.
- Accuracy is calculated using a simple accuracy calculating function.

Interpretation of Accuracy

In [ ]:
```
print(f"The Accuracy of the AdaBoost Classifier is: {acc}%")
```

```
The Accuracy of the AdaBoost Classifier is: 56.390977443609025%
```

Improvement Suggestions

- Experiment with different weak classifiers.
- Fine-tune hyperparameters, especially the learning rate.
- Consider increasing the number of weak classifiers.
- Evaluate performance on a variety of datasets to ensure generalization.
- Analyze misclassifications for further insights.

4. Multiclass SVM

4.1 Implementation of the Model

In [ ]:
```
class MultiClassSVM:
    def __init__(self, C=1.0, learning_rate=0.01, epochs=500):
        self.C = C  # Regularization parameter
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.classifiers = []

    def fit(self, X, y):
        unique_classes = np.unique(y)

        for cls in unique_classes:
            binary_labels = np.where(y == cls, 1, -1)
            classifier = self.train_one_class(X, binary_labels)
            self.classifiers.append((cls, classifier))

    def train_one_class(self, X, y):
        m, n = X.shape
        weights = np.zeros(n)
        bias = 0

        for epoch in range(self.epochs):
            for i in range(m):
                if y[i] * (np.dot(X[i], weights) - bias) >= 1:
```

```
                    weights -= self.learning_rate * (2 * self.C * weights
                else:
                    weights -= self.learning_rate * (2 * self.C * weights
                    bias -= self.learning_rate * y[i]

        return (weights, bias)

    def predict(self, X):
        predictions = []

        for cls, classifier in self.classifiers:
            weights, bias = classifier
            decision = np.dot(X, weights) - bias
            predictions.append((cls, decision))

        # Choose the class with the highest decision value as the predict
        return max(predictions, key=lambda x: x[1])[0]

# Convert labels to binary for each class
def to_binary_labels(y, target_class):
    return np.where(y == target_class, 1, -1)

# Train the SVM classifier
classifier = MultiClassSVM()
classifier.fit(X_train.values, y_train)

# Make predictions
predictions = [classifier.predict(x) for x in X_test.values]
```

In [ ]:

```
acc = accuracy(y_test, predictions)
acc
```

Out[ ]:

```
58.64661654135338
```

4.2 Insights drawn (plots, markdown explanations)

Multi-Class SVM Implementation

`to_binary_labels` Function

The `to_binary_labels` function converts the multi-class labels to binary labels for each class, where the target class is assigned a label of 1 and all other classes are assigned a label of -1.

`MultiClassSVM` Class

The `MultiClassSVM` class implements a multi-class Support Vector Machine (SVM) using a **ONE-VS-ALL** strategy. It has attributes for the regularization parameter (`C`), learning rate (`learning_rate`), and number of epochs (`epochs`). The trained classifiers for each class are stored in the `classifiers` attribute.

Methods:

1. `fit` **Method:** Iterates over unique classes, converts labels to binary, and trains a binary SVM for each class. The trained classifiers are stored in the `classifiers` attribute.

2. `train_one_class` **Method:** Trains a binary SVM for one class using stochastic gradient descent (SGD) with hinge loss.

3. `predict` **Method:** Makes predictions by obtaining decision values for each class and selecting the class with the highest decision value as the prediction.

Usage

- An instance of the `MultiClassSVM` class is created with specified parameters.
- The `fit` method is called to train the multi-class SVM on the training data.
- Predictions are made using the `predict` method on the test data.
- Accuracy is calculated using a simple accuracy calculating function.

Interpretation of Accuracy

In [ ]:
```
print(f"The Accuracy of the Multi-Class SVM Classifier is: {acc}%")
```

```
The Accuracy of the Multi-Class SVM Classifier is: 58.64661654135338%
```

Improvement Suggestions

- Fine-tune hyperparameters (e.g., `C`, `learning_rate`, `epochs`).
- Evaluate performance on various datasets to ensure generalization.
- Implement kernelized SVM for non-linear decision boundaries.
- Explore additional multi-class SVM strategies (e.g., one-vs-one).
- Use Cross-validation strategy to evaluate the Model using standard SVM libraries.

5. References

1. Dataset Description - https://www.hindawi.com/journals/cin/2022/9283293/

2. EDA and Data Cleaning - https://www.kaggle.com/code/charmainechiam/dealing-with-missing-values-in-data-preparation

3. PCA without sklearn: https://towardsdatascience.com/principal-component-analysis-pca-from-scratch-in-python-7f3e2a540c51

4. SVM: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/

5. AdaBoost: https://www.python-engineer.com/courses/mlfromscratch/13_adaboost/

6. Decision Trees: https://www.analyticsvidhya.com/blog/2020/10/all-about-decision-tree-from-scratch-with-python-implementation/