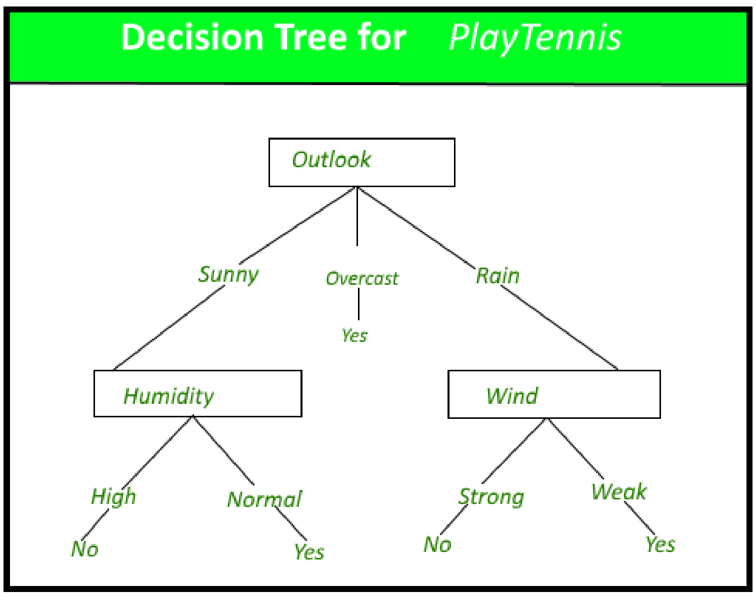
EXPERIMENT 6

Aim:
To implement any one of the classification algorithms(Decision tree/Naive Bayes) /Technique using python.

Theory:
Decision Tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart-like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.



Strengths and Weaknesses of the Decision Tree approach -

The strengths of decision tree methods are:

1. Decision trees are able to generate understandable rules.
2. Decision trees perform classification without requiring much computation.
3. Decision trees are able to handle both continuous and categorical variables.
4. Decision trees provide a clear indication of which fields are most important for prediction or classification.
5. Ease of use: Decision trees are simple to use and don't require a lot of technical expertise, making them accessible to a wide range of users.
6. Scalability: Decision trees can handle large datasets and can be easily parallelized to improve processing time.
7. Missing value tolerance: Decision trees are able to handle missing values in the data, making them a suitable choice for datasets with missing or incomplete data.

8. Handling non-linear relationships: Decision trees can handle non-linear relationships between variables, making them a suitable choice for complex datasets.
9. Ability to handle imbalanced data: Decision trees can handle imbalanced datasets, where one class is heavily represented compared to the others, by weighting the importance of individual nodes based on the class distribution.

The weaknesses of decision tree methods :

1. Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
2. Decision trees are prone to errors in classification problems with many classes and a relatively small number of training examples.
3. Decision trees can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.

Implementation:

1. Importing libraries and loading the dataset.

```
In [1]: import pandas as pd
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
```

```
In [2]: df = pd.read_csv('Housing.csv')
        df.head()
```

Out[2]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
1	12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
2	12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
3	12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished
4	11410000	7420	4	1	2	yes	yes	yes	no	yes	2	no	furnished

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 545 entries, 0 to 544
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   price             545 non-null    int64 
 1   area              545 non-null    int64 
 2   bedrooms          545 non-null    int64 
 3   bathrooms         545 non-null    int64 
 4   stories           545 non-null    int64 
 5   mainroad          545 non-null    object
 6   guestroom         545 non-null    object
 7   basement          545 non-null    object
 8   hotwaterheating   545 non-null    object
 9   airconditioning   545 non-null    object
 10  parking           545 non-null    int64 
 11  prefarea          545 non-null    object
 12  furnishingstatus  545 non-null    object
dtypes: int64(6), object(7)
memory usage: 55.5+ KB
```

2. Converting categorical values to numerical values.

In [4]:
```
from sklearn.preprocessing import OneHotEncoder
```

In [5]:
```
categorical_cols = ['mainroad', 'guestroom', 'basement',
                    'hotwaterheating', 'prefarea', 'furnishingstatus']
encoder = OneHotEncoder()
encoder.fit(df[categorical_cols])
```

Out[5]:
```
▾ OneHotEncoder
OneHotEncoder()
```

In [6]:
```
encoded_data = encoder.transform(df[categorical_cols]).toarray()
encoded_data = pd.DataFrame(encoded_data, columns=encoder.get_feature_names_out(categorical_cols))
encoded_data
```

Out[6]:

	mainroad_no	mainroad_yes	guestroom_no	guestroom_yes	basement_no	basement_yes	hotwaterheating_no	hotwaterheating_yes	prefarea_no	prefarea_
0	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	
1	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	
2	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	
3	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	
4	0.0	1.0	0.0	1.0	0.0	1.0	1.0	0.0	1.0	
...	...	...	...	...	...	...	...	...	...	
540	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0	1.0	
541	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	
542	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	
543	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	
544	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	

545 rows × 13 columns

In [7]:
```
encoded_df = pd.concat([df,encoded_data], axis = 1)
encoded_df.drop(categorical_cols,axis=1,inplace=True)
```

In [8]:
```
encoded_df.head()
```

Out[8]:

	price	area	bedrooms	bathrooms	stories	airconditioning	parking	mainroad_no	mainroad_yes	guestroom_no	guestroom_yes	basement_no	basemei
0	13300000	7420	4	2	3	yes	2	0.0	1.0	1.0	0.0	1.0	
1	12250000	8960	4	4	4	yes	3	0.0	1.0	1.0	0.0	1.0	
2	12250000	9960	3	2	2	no	2	0.0	1.0	1.0	0.0	0.0	
3	12215000	7500	4	2	2	yes	3	0.0	1.0	1.0	0.0	0.0	
4	11410000	7420	4	1	2	yes	2	0.0	1.0	0.0	1.0	0.0	

In [9]:
```
X = encoded_df.drop('airconditioning', axis=1).values
y = encoded_df['airconditioning'].values.reshape(-1,1)
```

3. Splitting the dataset into training dataset and testing dataset.

In [10]:
```
from sklearn.model_selection import train_test_split
```

In [11]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
```

4. Writing a custom python function for building a decision tree.

```
class Node():
    def __init__(self, threshold=None, left=None, right=None, info_gain=None,
feature_index=None, value=None):
        # for decision node
        self.right = right
        self.left = left
        self.threshold = threshold
```

```
        self.feature_index = feature_index
        self.info_gain = info_gain
        # for leaf node
        self.value = value


class DecisionTreeClassifier():
    def __init__(self, max_depth=2, min_samples_split=2):
        # initialize the root of tree
        self.root = None
        # stopping conditions
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split

    def build_tree(self, dataset, curr_depth=0):
        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_depth:
            #find the best split
            best_split = self.get_best_split(dataset, num_features, num_samples)
            # check if information gain is positive
            if best_split['info_gain'] > 0:
                # left recursive function
                left_subtree = self.build_tree(
                    best_split["dataset_left"], curr_depth+1)
                # right recursive function
                right_subtree = self.build_tree(
                    best_split["dataset_right"], curr_depth+1)
                # return decision node
                return Node(best_split["threshold"], left_subtree, right_subtree,
best_split["info_gain"], best_split["feature_index"])
        # return leaf node as stopping conditions are met
        leaf_value = self.calculate_leaf_value(y)
        return Node(value=leaf_value)
```

```
def get_best_split(self, dataset, num_features, num_samples):
    # dictionary to store values
    best_split = {}
    max_info_gain = -float("inf")
    #loop over all features values present in dataset
    for feature_index in range(num_features):
        feature_values = dataset[:,feature_index]
        possible_thresholds = np.unique(feature_values)
        # loop over all feature values
        for threshold in possible_thresholds:
            dataset_left, dataset_right = self.split(dataset, feature_index, threshold)
            # check if split/child are not empty
            if len(dataset_left)>0 and len(dataset_right)>0:
                y, left_y, right_y = dataset[:,-1], dataset_left[:,-1], dataset_right[:,-1]
                # compute information gain
                curr_info_gain = self.information_gain(y, left_y, right_y, "gini")
                if curr_info_gain>max_info_gain:
                    best_split["info_gain"] = curr_info_gain
                    best_split["feature_index"] = feature_index
                    best_split["dataset_left"] = dataset_left
                    best_split["dataset_right"] = dataset_right
                    best_split["threshold"] = threshold
                    max_info_gain = curr_info_gain
    return best_split

def split(self, dataset, feature_index, threshold):
    dataset_left = np.array([row for row in dataset if row[feature_index]<= threshold])
    dataset_right = np.array([row for row in dataset if row[feature_index] > threshold])
    return dataset_left, dataset_right

def information_gain(self, parent, left_child, right_child, mode="entropy"):
    weight_l = len(left_child)/len(parent)
    weight_r = len(right_child)/len(parent)
```

```
        if mode=="gini":
            gain = self.gini_index(parent) - (weight_l * self.gini_index(left_child) + weight_r
* self.gini_index(right_child))
        else:
            gain = self.entropy(parent) - (weight_l * self.entropy(left_child) + weight_r *
self.entropy(right_child))
        return gain

    def gini_index(self, y):
        class_labels = np.unique(y)
        gini = 0
        for label in class_labels:
            prob = len(y[y == label]) / len(y)
            gini += prob**2
        return 1 - gini

    def entropy(self, y):
        class_labels = np.unique(y)
        entropy = 0
        for label in class_labels:
            prob = len(y[y == label]) / len(y)
            entropy += -prob * np.log2(prob)
        return entropy

    def calculate_leaf_value(self, y):
        y = list(y)
        return max(y, key=y.count)

    def print_tree(self, tree=None, indent=" "):
        if not tree:
            tree = self.root
        if tree.value is not None:
            print(tree.value)
        else:
            print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
```

```
            print("%sleft:" % (indent), end="")
            self.print_tree(tree.left, indent + "  ")
            print("%sright:" % (indent), end="")
            self.print_tree(tree.right, indent + "  ")

    def fit(self, X, y):
        dataset = np.concatenate((X,y),axis=1)
        self.root = self.build_tree(dataset)

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

    def make_predictions(self, x, tree):
        if tree.value != None:
            return tree.value
        feature_value = x[tree.feature_index]
        if feature_value <= tree.threshold:
            return self.make_predictions(x, tree.left)
        else:
            return self.make_predictions(x, tree.left)
```

5. Fitting the model and calculating the accuracy.

In [18]:
```python
clf = DecisionTreeClassifier(max_depth=float("inf"))
clf.fit(X_train, y_train)
clf.print_tree()
```

```
X_0 <= 6107500.0 ? 0.055925526480522625
 left:X_0 <= 4613000.0 ? 0.020349705179744826
   left:X_0 <= 3605000.0 ? 0.006708549655615514
     left:X_18 <= 0.0 ? 0.005348138133096 98
       left:X_0 <= 3395000.0 ? 0.004818594104308481
         left:no
         right:X_0 <= 3500000.0 ? 0.03125
           left:X_1 <= 4240.0 ? 0.125
             left:X_1 <= 3036.0 ? 0.5
               left:no
               right:yes
             right:no
           right:no
       right:X_1 <= 3500.0 ? 0.006472729818665479
         left:X_0 <= 2520000.0 ? 0.06200396825396831
           left:no
           right:X_1 <= 3150.0 ? 0.14370748299319724
             left:X_0 <= 3115000.0 ? 0.05208333333333334
               left:X_0 <= 3080000.0 ? 0.4444444444444444
                 left:no
```

In [19]:
```python
y_pred = clf.predict(X_test)
from sklearn.metrics import accuracy_score
print("Accuracy:",accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6402439024390244
```

Conclusion:

Hence, we have created a custom function for creating a decision tree using python.