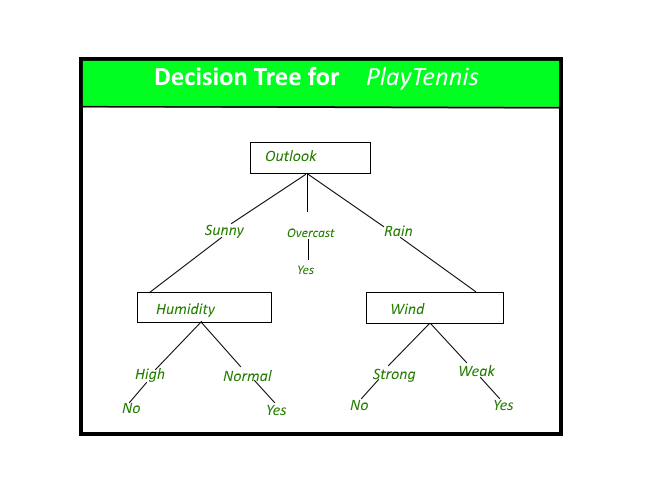
**EXPERIMENT 6**

**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:

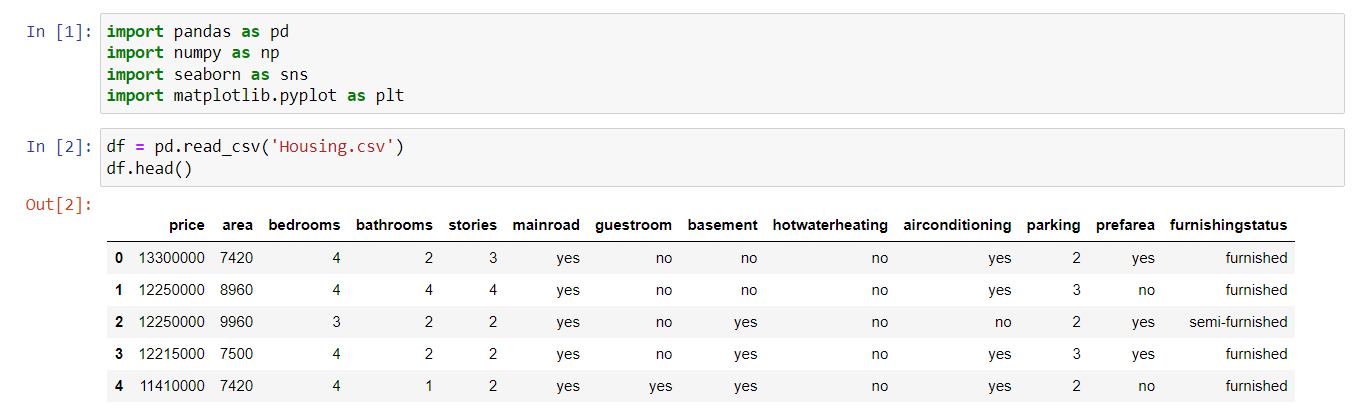
1. Decision trees are able to generate understandable rules.
2. Decision trees perform classification without requiring much computation.
3. Decision trees are able to handle both continuous and categorical variables.
4. Decision trees provide a clear indication of which fields are most important for 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.

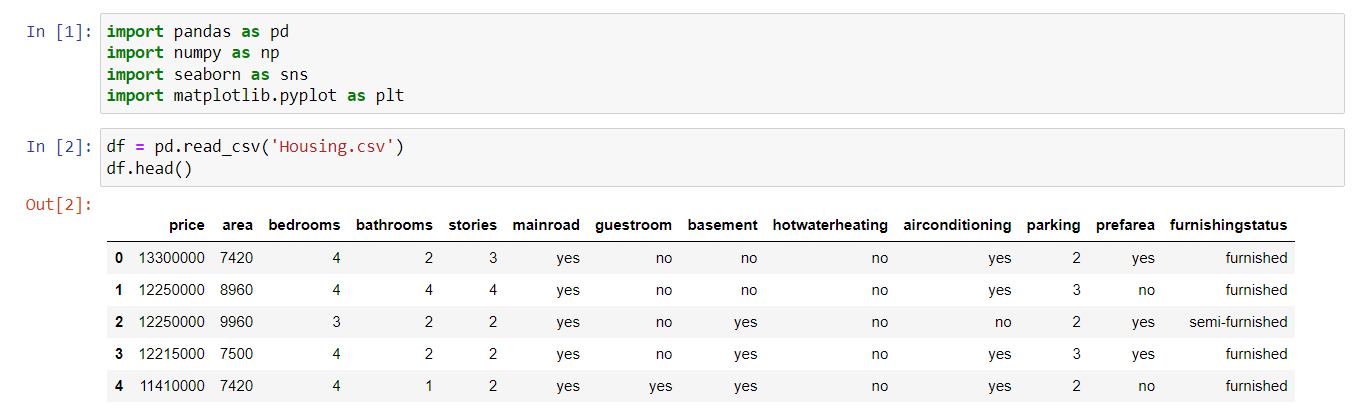
The weaknesses of decision tree methods :

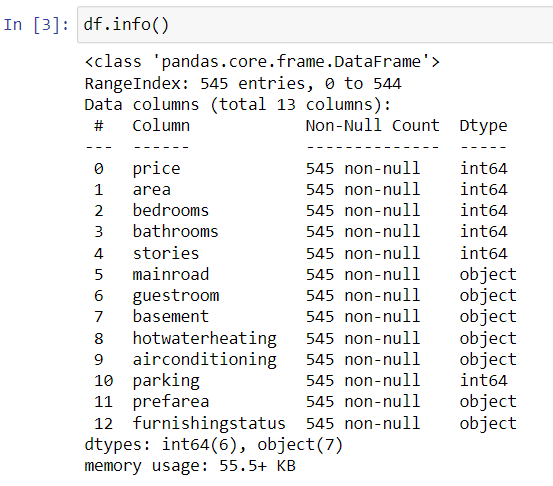
1. Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
2. Decision trees are prone to errors in classification problems with many classes and a relatively small number of training examples.
3. Decision trees can be computationally 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.

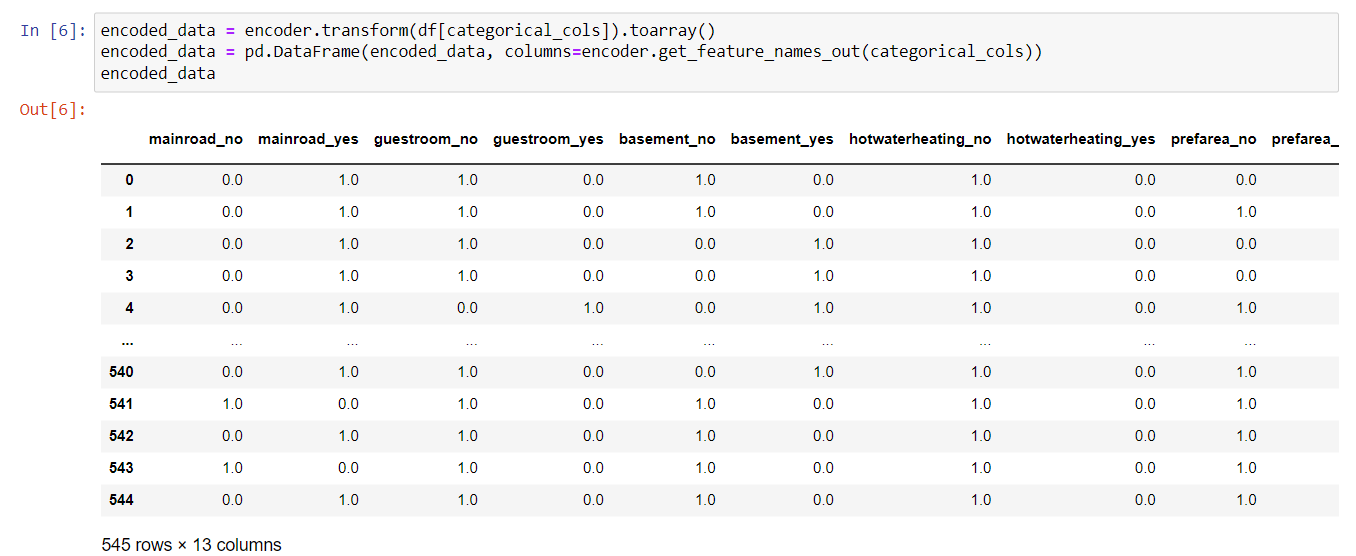


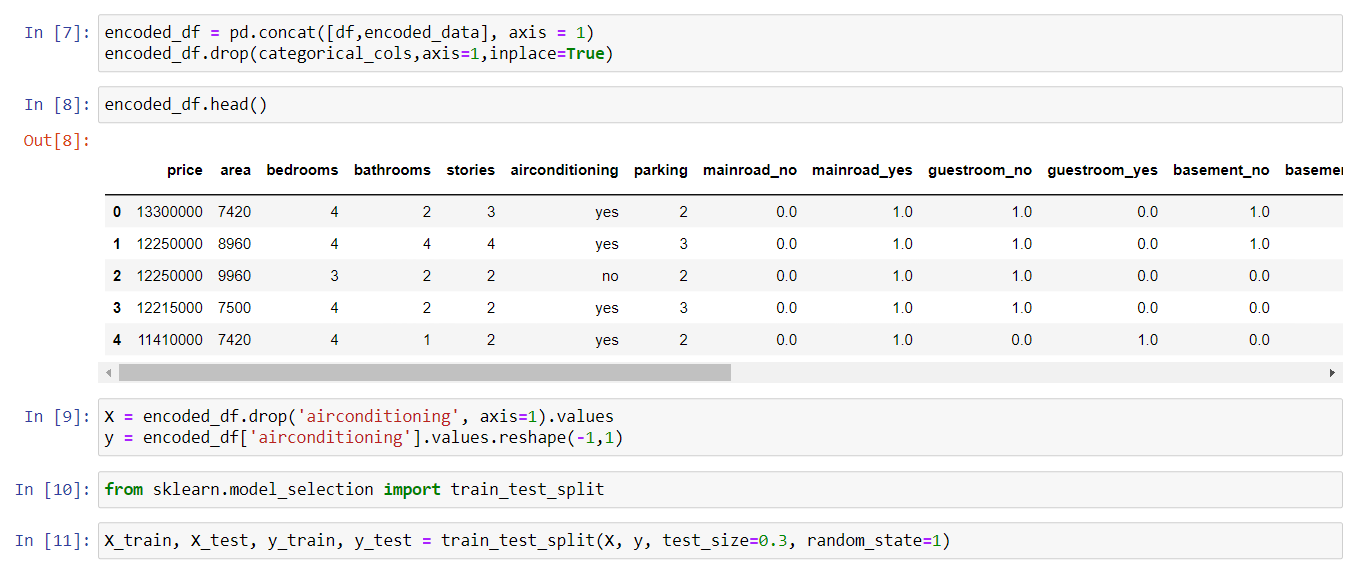




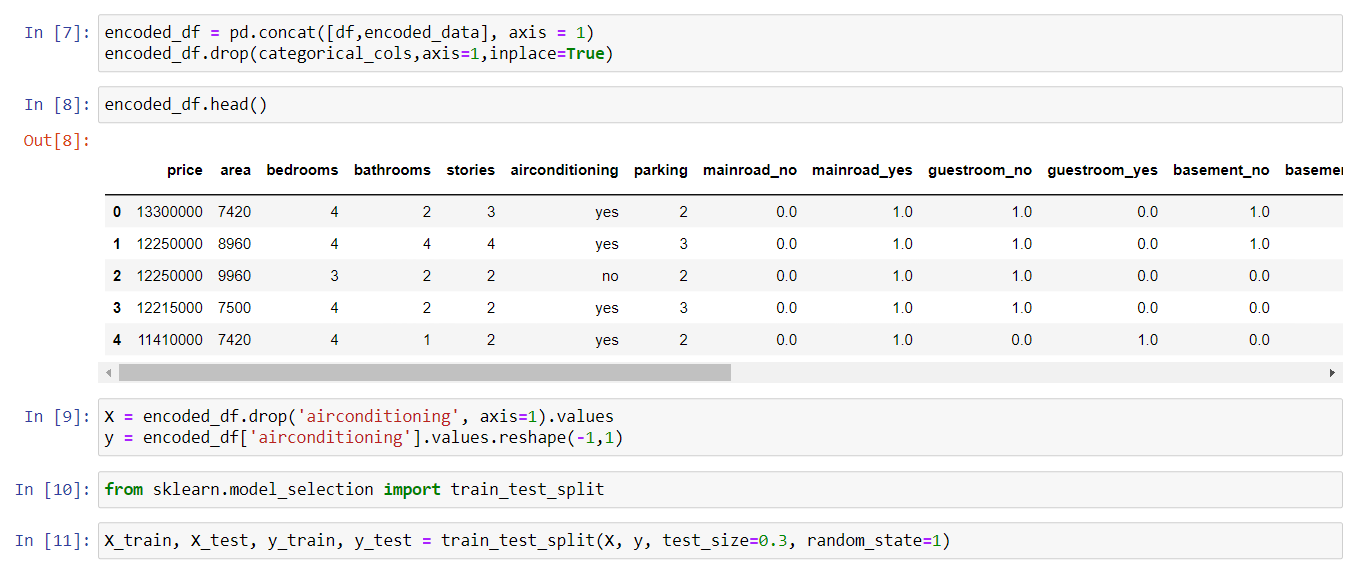
1. Converting categorical values to numerical values.







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



1. 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]

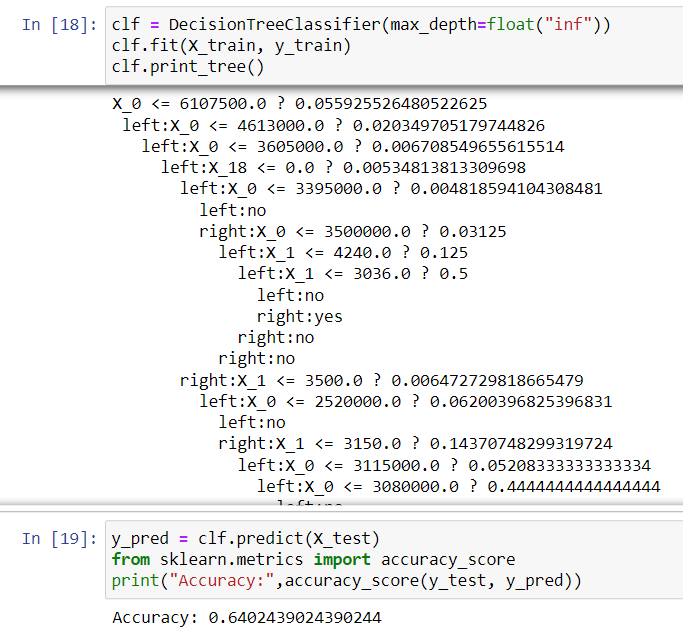
if feature\_value <= tree.threshold:

return self.make\_predictions(x, tree.left)

else:

return self.make\_predictions(x, tree.left)

1. Fitting the model and calculating the accuracy.



**Conclusion:**

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