Artificial Intelligence and Machine Learning Lab

1. Write a program to implement BFS and DFS Traversal.

BFS:-

Source Code:-

```
graph={
    '5':['2','3'],
    '2':['4','8'],
    '3':['6'],
    '4':[],
    '8':['7'],
    '6':[],
    '7':[]
}
visited=[]
queue=[]
def bfs(visited,graph,node):
    queue.append(node)
    visited.append(node)
    while queue:
        m=queue.pop(0)
        print(m,end=" ")
        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

print("BFS Nodes are:")
bfs(visited,graph,'5')
```

Out Put:-

```
BFS Nodes are:
5 2 3 4 8 6 7
```

DFS:-

Source Code:-

```
graph={
    '5':['2','3'],
```

```
    '2':['4','8'],
    '3':['6'],
    '4':[],
    '8':['7'],
    '6':[],
    '7':[]
}
visited=[]
stack=[]
def dfs(visited,graph,node):
    if node not in visited:
        visited.append(node)
        stack.append(node)
        n=stack.pop(0)#5
        print(n,end=" ")
        for neighbour in graph[node]:#2,3
            dfs(visited,graph,neighbour)

print("DFS nodes are:")
dfs(visited,graph,'5')
```

Out Put:-

```
DFS nodes are:
5 2 4 8 7 3 6
```

2. Write a program to implement A* Search.

Source Code:-

```
def aStarAlgo(start_node,stop_node):#A,G
    open_set=set(start_node)# C G
    closed_set=set()#A B E D
    g={}
    parents={}
    g[start_node]=0#g[A]=0
    parents[start_node]=start_node#A->A
    while len(open_set)>0:#2>0
        n=None
        for v in open_set:#G C
            if n==None or g[v]+heuristic(v)<g[n]+heuristic(n):#C
                n=v#G
                #3+99=102<11+0
        if n==stop_node or Graph_nodes[n]==None:
            pass
        else:
            for (m,weight) in get_neighbour(n):#('G',1)
                if m not in open_set and m not in closed_set:#B
                    open_set.add(m)#A B E C G D
                    parents[m]=n#B->A,E->A,C->B,G->D,D->E
                    g[m]=g[n]+weight#g[B]=0+2=2,g[E]=0+3=3
                    #g[c]=2+1=3,g[g]=2+9=11,g[d]=3+6=9
                else:
                    if g[m]>g[n]+weight:#11>9+1=10
                        g[m]=g[n]+weight#g[g]=10
                        parents[m]=n#G->D
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n==None:
            print('path does not exist!')
            return None
        if  n==stop_node:#A->A,B->A,E->A,C->B,G->D,D->E
            path=[]
            while parents[n]!=n:A!=A
                path.append(n)#G D E
                n=parents[n]#A
            path.append(start_node)#G D E A
```

```
                path.reverse()#A E D G
                print('path found: {}'.format(path))#A E D G
                return path
            open_set.remove(n)#  G C
            closed_set.add(n)#A B E D

        print('path does not exit!')
        return None



    def get_neighbour(v):#A
        if v in Graph_nodes:
            return Graph_nodes[v]
        else:
            return None

    def heuristic(n):
        H_dist={
            'A':11,
            'B':6,
            'C':99,
            'D':1,
            'E':7,
            'G':0
        }
        return H_dist[n]
    Graph_nodes={
        'A':[('B',2),('E',3)],
        'B':[('C',1),('G',9)],
        'C':None,
        'D':[('G',1)],
        'E':[('D',6)],
    }
    aStarAlgo('A','G')
```

Out Put:-

path found: ['A', 'E', 'D', 'G']

[1]:

['A', 'E', 'D', 'G']

3. **Write a program to implement Travelling Salesman Problem and Graph Coloring Problem**

Travelling Salesman Problem

Source Code:-

```
#Travelling Salesman Problem
from sys import maxsize
from itertools import permutations
v=4
def travellingSalesmanProblem(graph,s):
    vertex=[]#[1,2,3]
    for i in range(v):#(0,4)0,1,2,3
        if i!=s:#1!=0
            vertex.append(i)#1 2 3
    min_path=maxsize
    next_permutation=permutations(vertex)#[1,2,3][1,3,2][2,1,3][2,3,1]
                                     #[3,1,2][3,2,1]
    for i in next_permutation:#[1,2,3][1,3,2][2,1,3][2,3,1]
                                     #[3,1,2][3,2,1]
        current_pathweight=0
        k=s#00][
        for j in i:#[1,2,3] #3
            current_pathweight+=graph[k][j]#75
            k=j#3
        current_pathweight+=graph[k][s]#75+20=95
        min_path=min(min_path,current_pathweight)#95
    return min_path#80

 graph=[[0,10,15,20],[10,0,35,25],
           [15,35,0,30],[20,25,30,0]]
s=0
print(travellingSalesmanProblem(graph,s))#80
```

Out Put:-

80

Graph Coloring Problem

Source Code:-

```
colors=['Red','Blue','Green']
```

```
states=['a','b','c','d']
neighbors={}
neighbors['a']=['b','c','d']
neighbors['b']=['a','d']
neighbors['c']=['a','d']
neighbors['d']=['c','b','a']

colors_of_states={}

def promising(state,color):#d,green
    for neighbor in neighbors.get(state):#c,b,a
        color_of_neighbor=colors_of_states.get(neighbor)#blue
        if color_of_neighbor==color:#b==b
            return False
    return True

def get_color_for_state(state):#d
    for color in colors:#Red,Blue,Green
        if promising(state,color):#d,Red
            return color

def main():
    for state in states:#c,d
        colors_of_states[state]=get_color_for_state(state)#a:Red,b:blue,c:blue,d:green

    print(colors_of_states)

main()
```

Out Put:-

{'a': 'Red', 'b': 'Blue', 'c': 'Blue', 'd': 'Green'}

4. Write a program to implement Knowledge Representation.

Source Code:-

```
from sympy import symbols, Not, Implies, Xor, And

# Define propositional variables
rained = symbols('rained')
visited_hagrid = symbols('visited_hagrid')
visited_dumbledore = symbols('visited_dumbledore')

# Define the logical expressions based on the given statements
statement1 = Implies(Not(rained), visited_hagrid)  # If it didn't rain, then Harry visited
Hagrid today
statement2 = Xor(visited_hagrid, visited_dumbledore)  # Harry visited either Hagrid or
Dumbledore, but not both
statement3 = visited_dumbledore  # Harry visited Dumbledore today

# Combine the statements into a single formula
combined_formula = And(statement1, statement2, statement3)

# Function to check consistency of the combined formula and print evaluations
def check_combined_consistency():
    # Evaluate all possible scenarios for rained and visited_hagrid
    possible_values = [True, False]
    consistent_scenarios = []

    for rained_value in possible_values:#true,false
        for visited_hagrid_value in possible_values:#true,false
            # Substitute the variable values into the combined formula
            results = {
                rained: rained_value,#t
                visited_hagrid: visited_hagrid_value,#f
                visited_dumbledore: True  # We know that Harry visited Dumbledore today
            }

            # Evaluate the logical statements individually
            eval_statement1  =  statement1.subs(results)#t
            eval_statement2  =  statement2.subs(results)#t
            eval_statement3 = statement3.subs(results)#t
```

```
        # Evaluate the combined formula
        eval_combined_formula = combined_formula.subs(results)#t

        # Print the evaluation of each statement and the combined formula
        print(f"rained={rained_value}, visited_hagrid={visited_hagrid_value},
visited_dumbledore=True")
        print(f"  Statement 1 (¬R → H) evaluates to: {eval_statement1}")
        print(f"  Statement 2 (H ⊕ D) evaluates to: {eval_statement2}")
        print(f"  Statement 3 (D) evaluates to: {eval_statement3}")
        print(f"  Combined Formula evaluates to: {eval_combined_formula}\n")

        # Append to consistent scenarios if the combined formula is true
        if eval_combined_formula:
            consistent_scenarios.append((rained_value, visited_hagrid_value))#t,f

    return consistent_scenarios

# Find consistent scenarios
consistent_scenarios = check_combined_consistency()#t,f

# Output consistent scenarios
print("Consistent scenarios based on the combined formula:")
if consistent_scenarios:#t,f
    for scenario in consistent_scenarios:
        rained_value, visited_hagrid_value = scenario#t,f
        print(f"rained={rained_value}, visited_hagrid={visited_hagrid_value},
visited_dumbledore=True")
else:
    print("No consistent scenarios found.")

# Output the combined formula for reference
print("\nCombined logical formula:")
print(combined_formula)
```

Out Put:-

```
rained=True, visited_hagrid=True, visited_dumbledore=True
  Statement 1 (¬R → H) evaluates to: True
  Statement 2 (H ⊕ D) evaluates to: False
  Statement 3 (D) evaluates to: True
  Combined Formula evaluates to: False
```

rained=True, visited_hagrid=False, visited_dumbledore=True
  Statement 1 (¬R → H) evaluates to: True
  Statement 2 (H ⊕ D) evaluates to: True
  Statement 3 (D) evaluates to: True
  Combined Formula evaluates to: True

rained=False, visited_hagrid=True, visited_dumbledore=True
  Statement 1 (¬R → H) evaluates to: True
  Statement 2 (H ⊕ D) evaluates to: False
  Statement 3 (D) evaluates to: True
  Combined Formula evaluates to: False

rained=False, visited_hagrid=False, visited_dumbledore=True
  Statement 1 (¬R → H) evaluates to: False
  Statement 2 (H ⊕ D) evaluates to: True
  Statement 3 (D) evaluates to: True
  Combined Formula evaluates to: False

Consistent scenarios based on the combined formula:
rained=True, visited_hagrid=False, visited_dumbledore=True

Combined logical formula:
visited_dumbledore & (visited_dumbledore ^ visited_hagrid) & (Implies(~rained, visited_hagrid))

5. Write a program to implement Bayesian Network.

Source Code:-

```
# Define conditional probability tables (CPTs)
P_burglary = 0.002#t
P_earthquake = 0.001#t

# Probability of alarm given burglary and earthquake
P_alarm_given_burglary_and_earthquake = 0.94
P_alarm_given_burglary_and_no_earthquake = 0.95
P_alarm_given_no_burglary_and_earthquake = 0.31
P_alarm_given_no_burglary_and_no_earthquake = 0.001

# Probability of David calling given alarm
P_david_calls_given_alarm = 0.91#t
P_david_does_not_call_given_alarm = 0.09
P_david_calls_given_no_alarm = 0.05#t
P_david_does_not_call_given_no_alarm = 0.95

# Probability of Sophia calling given alarm
P_sophia_calls_given_alarm = 0.75
P_sophia_does_not_call_given_alarm = 0.25
P_sophia_calls_given_no_alarm = 0.02
P_sophia_does_not_call_given_no_alarm = 0.98

# Calculate joint probability
def joint_probability(alarm, burglary, earthquake, david_calls, sophia_calls):#(t,f,f,t,t)
    if alarm:
        if burglary and earthquake:
            P_alarm = P_alarm_given_burglary_and_earthquake
        elif burglary:
            P_alarm = P_alarm_given_burglary_and_no_earthquake
        elif earthquake:
            P_alarm = P_alarm_given_no_burglary_and_earthquake
        else:
            P_alarm = P_alarm_given_no_burglary_and_no_earthquake#0.001
    else:
if burglary and earthquake:
            P_alarm = 1 - P_alarm_given_burglary_and_earthquake
        elif burglary:
```

```
        P_alarm = 1 - P_alarm_given_burglary_and_no_earthquake
    elif earthquake:
        P_alarm = 1 - P_alarm_given_no_burglary_and_earthquake
    else:
        P_alarm = 1 - P_alarm_given_no_burglary_and_no_earthquake

    P_david = (P_david_calls_given_alarm if david_calls else
P_david_does_not_call_given_alarm) if alarm else (P_david_calls_given_no_alarm if
david_calls else P_david_does_not_call_given_no_alarm)#0.91

    P_sophia = (P_sophia_calls_given_alarm if sophia_calls else
P_sophia_does_not_call_given_alarm) if alarm else (P_sophia_calls_given_no_alarm if
sophia_calls else P_sophia_does_not_call_given_no_alarm)#0.75

    return (P_burglary if burglary else 1 - P_burglary) * (P_earthquake if earthquake else 1
- P_earthquake) * P_alarm * P_david * P_sophia#0.75*0.91*0.001*0.998*0.999

# Calculate the probability for the given scenario
result = joint_probability(
    alarm=True,
    burglary=False,
    earthquake=False,
    david_calls=True,
    sophia_calls=True
)


# Print the result
print(f'The probability that the alarm has sounded, there is neither a burglary nor an
earthquake, and both David and Sophia called Harry is: {result:.8f}')
```

Out Put:-

The probability that the alarm has sounded, there is neither a burglary nor an earthquake, and bo
th David and Sophia called Harry is: 0.00068045

6. Write a program to implement Hidden Markov Model.

Source Code:-

```
import numpy as np


class HMM:
    def __init__(self, states, observations):#['Sunny', 'Cloudy', 'Rainy'],['Umbrella', 'Normal', 
'Raincoat']
        self.states = states#['Sunny', 'Cloudy', 'Rainy']
        self.n_states = len(states)#3
        self.n_obs = len(observations)#3
        self.state_index = {state: i for i, state in enumerate(states)}#{'Sunny': 0, 'Cloudy': 1, 
'Rainy': 2}
        self.obs_index = {obs: i for i, obs in enumerate(observations)}#{'Umbrella': 0, 'Normal': 1, 
'Raincoat': 2}


        # Transition probability matrix (A)
        self.A = np.array([
            [0.6, 0.3, 0.1],  # Sunny -> Sunny, Cloudy, Rainy
            [0.2, 0.5, 0.3],  # Cloudy -> Sunny, Cloudy, Rainy
            [0.1, 0.4, 0.5]   # Rainy -> Sunny, Cloudy, Rainy
        ])


        # Emission probability matrix (B)
        self.B = np.array([
            [0.8, 0.15, 0.05],  # Sunny: Umbrella, Normal, Raincoat
            [0.3, 0.4, 0.3],    # Cloudy: Umbrella, Normal, Raincoat
            [0.1, 0.2, 0.7]     # Rainy: Umbrella, Normal, Raincoat
        ])
```

```
        # Initial state probabilities (pi)
        self.pi = np.array([0.5, 0.3, 0.2])  # Sunny, Cloudy, Rainy

    def forward(self, obs_seq):#[0,1,0,2]
        n = len(obs_seq)#4
        alpha = np.zeros((n, self.n_states))#(4,3)
        '''
         [[0. 0. 0.]
          [0. 0. 0.]
          [0. 0. 0.]
          [0. 0. 0.]]
          '''

        # Initialize alpha
        alpha[0] = self.pi * self.B[:, obs_seq[0]]#[0.5 0.3 0.2]*[0.8 0.3 0.1]=[0.4  0.09 0.02]
        # Recursion
        for t in range(1, n):#1,2,3
            for j in range(self.n_states):#0,1,2
                alpha[t, j] = (alpha[t-1] @ self.A[:, j]) * self.B[j, obs_seq[t]]#alpha[1]
                #alpha[0]*A[:,0]*b[0,1]
        #([0.4  0.09 0.02]*[0.6 0.2 0.1])*0.15
#[0.039,0.0692,0.0154]
        #[0.031024,0.015738,0.003236]
        #[0.00110428,0.00554118,0.00660926]
        # Probability of the observation sequence
        return alpha.sum(axis=1)[-1]#0.0133.
```

```
# Define states and observations
states = ['Sunny', 'Cloudy', 'Rainy']
observations = ['Umbrella', 'Normal', 'Raincoat']#0,1,2

# Initialize the HMM
hmm = HMM(states, observations)

# Define an observation sequence
obs_seq = ['Umbrella', 'Normal', 'Umbrella','Raincoat']  # Convert this to indices for
computation
obs_seq_indices = [hmm.obs_index[obs] for obs in obs_seq]#[0,1,0,2]

# Evaluate the probability of the observation sequence
prob = hmm.forward(obs_seq_indices)
print(f"Probability of the observation sequence '{obs_seq}': {prob:.4f}")#0.0133.
```

Out Put:-

Probability of the observation sequence '['Umbrella', 'Normal', 'Umbrella', 'Raincoat']': 0.0133

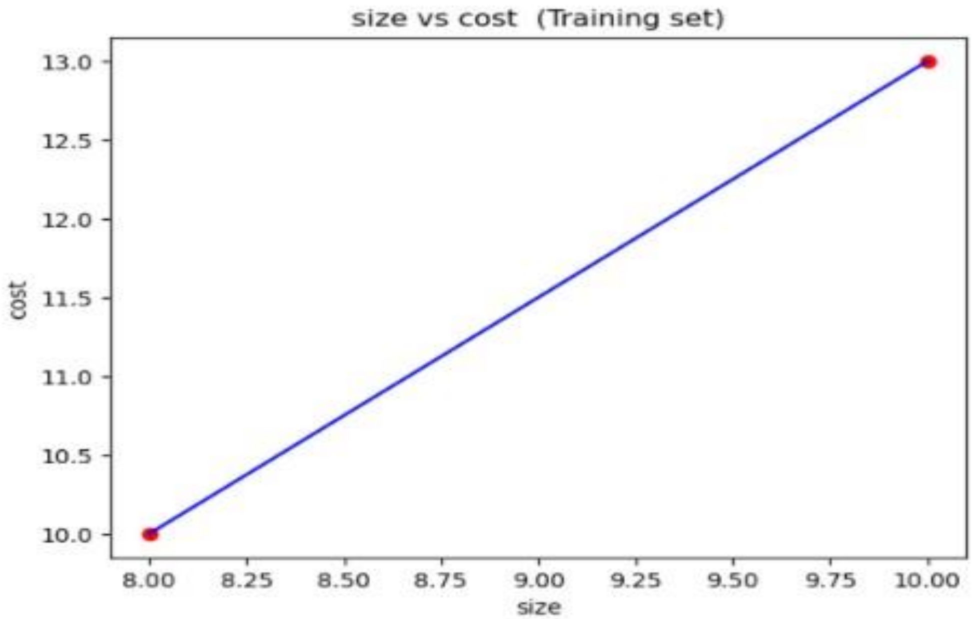
7. Write a program to implement Regression algorithm.

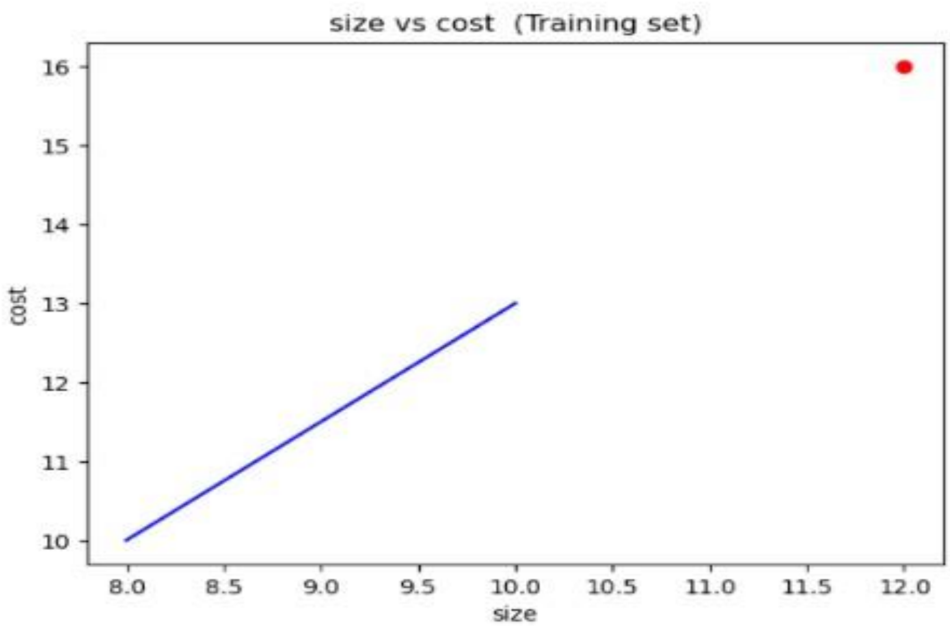
Source Code:-

- import numpy as np
- import pandas as pd
- import matplotlib.pyplot as plt
- #from google.colab import files
- #uploaded = files.upload()
- dataset = pd.read_csv('pizza.csv')
- X = dataset.iloc[:, 0:-1].values #independent variable array
- y = dataset.iloc[:,1].values #dependent variable vector
- # splitting the dataset
- from sklearn.model_selection import train_test_split
- X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1/3,random_state=0)
- # fitting the regression model
- from sklearn.linear_model import LinearRegression
- regressor = LinearRegression()
- regressor.fit(X_train,y_train) #actually produces the linear eqn for the data
- regressor
- # predicting the test set results
- y_pred = regressor.predict(X_test)
- #comparing both y_test and y_pred
- df1=pd.DataFrame({'Actual':y_test,'Prediction':y_pred})
- df1
- # visualizing the results
- #plot for the TRAIN
- plt.scatter(X_train, y_train, color='red') # plotting the observation line
- plt.plot(X_train, regressor.predict(X_train), color='blue') # plotting the regression line
- plt.title("size vs cost (Training set)") # stating the title of the graph
- plt.xlabel("size") # adding the name of x-axis
- plt.ylabel("cost") # adding the name of y-axis
- plt.show() # specifies end of graph
- #plot for the TEST
- plt.scatter(X_test, y_test, color='red')
- plt.plot(X_train, regressor.predict(X_train), color='blue') # plotting the regression line
- plt.title("size vs cost (Training set)") # stating the title of the graph
- plt.xlabel("size") # adding the name of x-axis
- plt.ylabel("cost") # adding the name of y-axis

- plt.show()
- dataset.head()

Out Put:-





[4]:

	size	cost
0	8	10
1	10	13
2	12	16

8. Write a program to implement decision tree based ID3 algorithm.

Source Code:-

```
import pandas as pd
from collections import Counter
import math
from pprint import pprint

# Entropy calculation function
def entropy(probs):
    return sum(-prob * math.log(prob, 2) for prob in probs if prob > 0)

# Calculate entropy of a list
def entropy_of_list(a_list):
    cnt = Counter(a_list)
    num_instances = len(a_list)
    probs = [x / num_instances for x in cnt.values()]
    return entropy(probs)

# Information gain function
def information_gain(df, split_attribute_name, target_attribute_name):
    df_split = df.groupby(split_attribute_name)
    print(df_split)
    nobs = len(df.index) * 1.0

    df_agg_ent = df_split[target_attribute_name].agg(
        [entropy_of_list, lambda x: len(x) / nobs]
    )
```

```
    avg_info = sum(df_agg_ent['entropy_of_list'] * df_agg_ent['<lambda_0>'])
    old_entropy = entropy_of_list(df[target_attribute_name])
    return old_entropy - avg_info


# ID3 Decision Tree algorithm
def id3DT(df, target_attribute_name, attribute_names, default_class=None):
    cnt = Counter(df[target_attribute_name])
    if len(cnt) == 1:
        return next(iter(cnt))
    elif df.empty or not attribute_names:
        return default_class
    else:
        default_class = max(cnt, key=cnt.get)
        gainz = [information_gain(df, attr, target_attribute_name) for attr in attribute_names]

        index_of_max = gainz.index(max(gainz))
        best_attr = attribute_names[index_of_max]
        tree = {best_attr: {}}
        remaining_attributes = [i for i in attribute_names if i != best_attr]
        for attr_val, data_subset in df.groupby(best_attr):
            subtree = id3DT(data_subset, target_attribute_name, remaining_attributes,
default_class)
            tree[best_attr][attr_val] = subtree

        return tree


# Simulate the dataset from the image provided earlier
data = {
```

```
    'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain', 'Overcast', 'Sunny', 'Sunny', 'Rain', 'Sunny', 'Overcast', 'Overcast', 'Rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],
    'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Strong'],
    'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
}


df = pd.DataFrame(data)
df
```

Output:-

	Outlook	Temperature	Humidity	Wind	PlayTennis
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Cool	Normal	Strong	No
6	Overcast	Cool	Normal	Strong	Yes
7	Sunny	Mild	High	Weak	No
8	Sunny	Cool	Normal	Weak	Yes
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Mild	Normal	Strong	Yes
11	Overcast	Mild	High	Strong	Yes
12	Overcast	Hot	Normal	Weak	Yes
13	Rain	Mild	High	Strong	No

Code Continue:-

```
# Define attribute names and target column name
attribute_names = list(df.columns)
attribute_names.remove('PlayTennis')


# Build the decision tree
tree = id3DT(df, 'PlayTennis', attribute_names)
print("The Resultant Decision Tree is:")
pprint(tree)
```

Output:-

```
The Resultant Decision Tree is:
{'Outlook': {'Overcast': 'Yes',
            'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},
            'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

9. Write a program to implement K-Means Clustering algorithm.

Source Code:-

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np

iris =datasets.load_iris()
X=pd.DataFrame(iris.data)
X.columns=['Sepal_Length','Sepal_Width', 'Petal_length', 'Petal_Width']
y=pd.DataFrame(iris.target)
y.columns=['target']
plt.figure(figsize=(14,7))
colormap=np.array(['red','lime','black'])
plt.subplot(1,2,1)
plt.scatter(X.Sepal_Length,X.Sepal_Width,c=colormap[y.target],s=40)
plt.title('Sepal')
plt.subplot(1,2,2)
plt.scatter(X.Petal_length,X.Petal_Width,c=colormap[y.target],s=40)
plt.title('Petal')
model=KMeans(n_clusters=3)
model.fit(X)
print(model.labels_)
plt.subplot(1,2,1)
plt.scatter(X.Petal_length,X.Petal_Width,c=colormap[y.target],s=40)
```

plt.title('Real Classification')

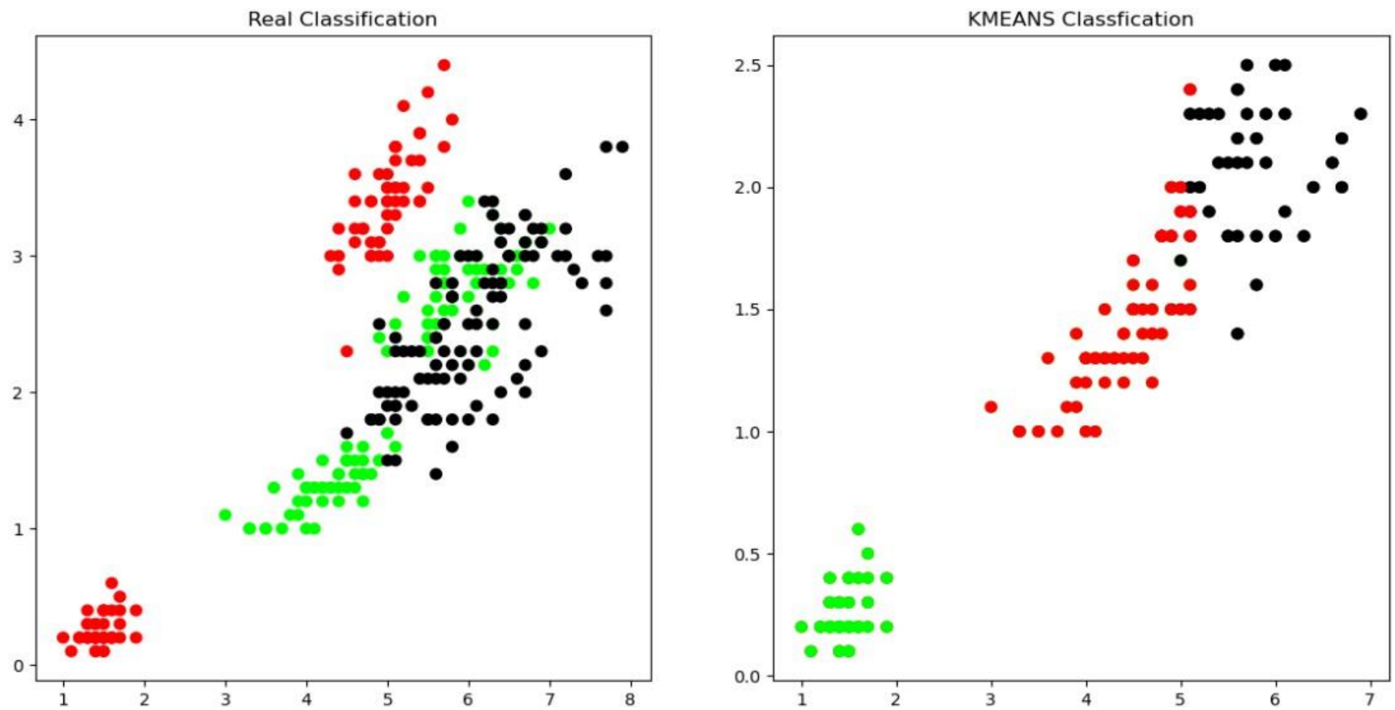
plt.subplot(1,2,2)

plt.scatter(X.Petal_length,X.Petal_Width,c=colormap[model.labels_],s=40)

plt.title('KMEANS Classfication')

Out Put:-

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 2 2 2
 2 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 0 2 2 2 2 2 0 2 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
C:\ProgramData\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1446: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
[9]: Text(0.5, 1.0, 'KMEANS Classfication')
```



10. Write a program to implement K-Nearest Neighbor algorithm (K-NN).

Source Code:-

```
# Import necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

# Load the Iris dataset
iris = load_iris()

# Print dataset keys for reference
print("Dataset keys:", iris.keys())

# Convert data to DataFrame for better visualization
df = pd.DataFrame(iris['data'], columns=iris['feature_names'])
print("Feature Data:\n", df.head()) # Display first few rows of feature data

# Target names and feature names for reference
print("Target names:", iris['target_names'])
print("Feature names:", iris['feature_names'])

# Display target values (species labels)
print("Target array:\n", iris['target'])

# Define features (X) and target labels (y)
X = df
y = iris['target']

# Split the dataset into training and testing sets (67% train, 33% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,random_state=42)

# Initialize and train the k-NN classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Predict on the test set
y_pred = knn.predict(X_test)
```

```
    # Generate and print the confusion matrix on test data
    cm_test = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix (Test Data):\n", cm_test)

    # Calculate and print accuracy on test data
    accuracy_test = accuracy_score(y_test, y_pred)
    print("Correct prediction on test data:", accuracy_test)
    print("Wrong prediction on test data:", 1 - accuracy_test)

    # Predict on the training set to observe training performance
    y_train_pred = knn.predict(X_train)

    # Generate and print the confusion matrix on training data
    cm_train = confusion_matrix(y_train, y_train_pred)
    print("Confusion Matrix (Training Data):\n", cm_train)
```

Out Put:-

```
Dataset keys: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filena
me', 'data_module'])
Feature Data:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
Target names: ['setosa' 'versicolor' 'virginica']
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target array:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
Confusion Matrix (Test Data):
 [[19  0  0]
 [ 0 15  0]
 [ 0  1 15]]
Correct prediction on test data: 0.98
Wrong prediction on test data: 0.020000000000000018
Confusion Matrix (Training Data):
 [[31  0  0]
 [ 0 33  2]
```

```
 [ 0  2 32]]
```

11. Write a program to implement Back Propagation Algorithm.

Source Code:-

```
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Initialize dataset (input features and corresponding labels)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # XOR problem
y = np.array([[0], [1], [1], [0]])  # XOR output

# Set the hyperparameters
input_layer_neurons = 2  # 2 features
hidden_layer_neurons = 4  # Number of neurons in hidden layer
output_layer_neurons = 1  # Single output neuron
epochs = 10000  # Number of iterations
learning_rate = 0.1  # Learning rate

# Initialize weights and biases with random values
np.random.seed(42)  # For reproducibility
```

```
# Weights between input and hidden layer
wh = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))
bh = np.random.uniform(size=(1, hidden_layer_neurons))

# Weights between hidden and output layer
wout = np.random.uniform(size=(hidden_layer_neurons, output_layer_neurons))
bout = np.random.uniform(size=(1, output_layer_neurons))

# Training the neural network
for epoch in range(epochs):
    # Forward pass

    # Hidden layer input
    hidden_layer_input = np.dot(X, wh) + bh
    # Hidden layer activation (sigmoid)
    hidden_layer_output = sigmoid(hidden_layer_input)

    # Output layer input
    output_layer_input = np.dot(hidden_layer_output, wout) + bout
    # Output layer activation (sigmoid)
    output = sigmoid(output_layer_input)

    # Calculate the error (difference between actual and predicted)
    error = y - output

    # Backpropagation
    # Output layer gradients
    output_layer_gradient = sigmoid_derivative(output)
```

```
    d_output = error * output_layer_gradient  # Derivative of loss with respect to output

    # Hidden layer gradients
    hidden_layer_gradient = sigmoid_derivative(hidden_layer_output)
    d_hidden_layer = d_output.dot(wout.T) * hidden_layer_gradient
    # Derivative of loss w.r.t. hidden layer

    # Update weights and biases using gradient descent
    wout += hidden_layer_output.T.dot(d_output) * learning_rate  # Update weights between hidden and output layer
    bout += np.sum(d_output, axis=0, keepdims=True) * learning_rate  # Update biases for output layer

    wh += X.T.dot(d_hidden_layer) * learning_rate  # Update weights between input and hidden layer
    bh += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
    # Update biases for hidden layer

    # Optionally print the error for every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Error: {np.mean(np.abs(error))}")

# Final predictions
print("Final predictions after training:")
print(output)
```

Out Put:-

```
Epoch 0, Error: 0.49914791405546904
Epoch 1000, Error: 0.4989908274224632
Epoch 2000, Error: 0.49392112204426847
Epoch 3000, Error: 0.46086324847622695
Epoch 4000, Error: 0.37081148754970494
```

```
Epoch 5000, Error: 0.2293685934150816
Epoch 6000, Error: 0.1411700792664044
Epoch 7000, Error: 0.10187019467760619
Epoch 8000, Error: 0.08085064924133495
Epoch 9000, Error: 0.06790718296112089
Final predictions after training:
[[0.04690963]
 [0.95663392]
 [0.92548675]
 [0.07177571]]
```

12. Write a program to implement Support Vector Machine.

Source Code:-

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# Step 1: Create a small dataset with 2 numeric features
# Example data points where each point has 2 features and a sentiment label
# Feature 1 could represent "positivity score" and Feature 2 "intensity score"
data = np.array([
    [1.5, 2.0, 1],  # Positive sentiment
    [1.0, 1.0, 1],  # Positive sentiment
    [2.0, 2.5, 1],  # Positive sentiment
    [2.5, 1.5, 1],  # Positive sentiment
    [3.0, 1.0, 0],  # Negative sentiment
    [3.5, 0.5, 0],  # Negative sentiment
    [4.0, 1.0, 0],  # Negative sentiment
    [4.5, 1.5, 0]   # Negative sentiment
])

# Separate features and labels
X = data[:, :2]  # First two columns are features
y = data[:, 2]   # Last column is the label (1 for positive, 0 for negative)

# Step 2: Train the SVM model
svm_model = SVC(kernel='linear')
svm_model.fit(X, y)
```

```
# Step 3: Evaluate the model
y_pred = svm_model.predict(X)
print("Accuracy:", accuracy_score(y, y_pred))
print("\nClassification Report:\n", classification_report(y, y_pred))

# Step 4: Visualize the Decision Boundary
# Create a mesh to plot the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))
print("xx=",xx)
print("yy=",yy)

# Predict on each point of the mesh to determine decision boundaries
Z = svm_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
print ("Z=",Z)

# Plot the decision boundary and data points
plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z, alpha=0.2, cmap='coolwarm')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolor='k', s=100)
plt.xlabel("Feature 1 (e.g., Positivity Score)")
plt.ylabel("Feature 2 (e.g., Intensity Score)")
plt.title("SVM Decision Boundary on 2-Feature Sentiment Data")
plt.show()
```

Out Put:-

```
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00         4
         1.0       1.00      1.00      1.00         4

    accuracy                           1.00         8
   macro avg       1.00      1.00      1.00         8
```

```
weighted avg       1.00      1.00      1.00         8

xx= [[0.   0.01 0.02 ... 5.47 5.48 5.49]
 [0.   0.01 0.02 .... 5.47 5.48 5.49]
 [0.   0.01 0.02 .... 5.47 5.48 5.49]
 ...
 [0.   0.01 0.02 .... 5.47 5.48 5.49]
 [0.   0.01 0.02 .... 5.47 5.48 5.49]
 [0.   0.01 0.02 .... 5.47 5.48 5.49]]
yy= [[-0.5  -0.5  -0.5  ... -0.5  -0.5  -0.5 ]
 [-0.49 -0.49 -0.49 ... -0.49 -0.49 -0.49]
 [-0.48 -0.48 -0.48 ... -0.48 -0.48 -0.48]
 ...
 [ 3.47  3.47  3.47 ...  3.47  3.47  3.47]
 [ 3.48  3.48  3.48 ...  3.48  3.48  3.48]
 [ 3.49  3.49  3.49 ...  3.49  3.49  3.49]]
Z= [[1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 ...
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]]
```