```
graph = {}
edge_node = set()

def add_node(node):
    if node in graph:
        print(f"This node '{node}' already exists. Enter a new node:")
        return False
    graph[node] = []
    return True

def add_edge(u, v):
    edge = (u, v)
    if edge in edge_node:
        print(f"Edge {u}-{v} already exists. Enter a different edge.")
        return False
    if u not in graph or v not in graph:
        print("Enter nodes before connecting edges.")
        return False
    graph[u].append(v)
    edge_node.add(edge)
    return True

visited_bfs = []
queue = []

def bfs(start):
    visited_bfs.clear()
    queue.clear()
    queue.append(start)
    visited_bfs.append(start)
    while queue:
        node = queue.pop(0)
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in visited_bfs:
                visited_bfs.append(neighbor)
                queue.append(neighbor)

visited_dfs = []

def dfs(start):
    visited_dfs.clear()
    _dfs_helper(start)

def _dfs_helper(node):
    visited_dfs.append(node)
    print(node, end=" ")
    for neighbor in graph[node]:
        if neighbor not in visited_dfs:
            _dfs_helper(neighbor)
```

```
# Input nodes
n = int(input("Enter number of nodes: "))
i = 0
while i < n:
    node = input(f"Enter node {i+1}: ").strip()
    if add_node(node):
        i += 1

# Input edges
e = int(input("Enter number of edges: "))
for i in range(e):
    while True:
        u, v = input(f"Enter edge {i+1} (two nodes separated by
space): ").split()
        if add_edge(u, v):
            break

start = input("Enter starting node: ").strip()
if start in graph:
    print("BFS: ")
    bfs(start)
    print("\nDFS: ")
    dfs(start)
else:
    print("Start node not found.")
```

```
Enter number of nodes:  7
Enter node 1:  5
Enter node 2:  2
Enter node 3:  3
Enter node 4:  4
Enter node 5:  8
Enter node 6:  6
Enter node 7:  7
Enter number of edges:  6
Enter edge 1 (two nodes separated by space):  5 2
Enter edge 2 (two nodes separated by space):  5 3
Enter edge 3 (two nodes separated by space):  2 4
Enter edge 4 (two nodes separated by space):  2 8
Enter edge 5 (two nodes separated by space):  3 6
Enter edge 6 (two nodes separated by space):  8 7
Enter starting node:  5

BFS:
5 2 3 4 8 6 7
DFS:
5 2 4 8 7 3 6
```

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n is None:
            print('Path does not exist!')
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        for (m, weight) in get_neighbors(n) or []:
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None
def get_neighbors(v):
    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.get(n, float('inf'))
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
    'G': None
}

aStarAlgo('A', 'G')
```

```
Path found: ['A', 'E', 'D', 'G']
```

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

```
from sys import maxsize
from itertools import permutations
v=4
def travellingSalesmanProblem(graph,s):
    vertex=[]
    for i in range(v):
        if i!=s:
            vertex.append(i)
    min_path=maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        current_pathweight=0
        k=s
        for j in i:
            current_pathweight+=graph[k][j]
            k=j
        current_pathweight+=graph[k][s]
        min_path=min(min_path,current_pathweight)
    return min_path
graph=[[0,3,6,8],[3,0,9,4],[6,9,0,2],[8,4,2,0]]
s=0
print(travellingSalesmanProblem(graph,s))
```

```
15
```

```
colors=['Red','Blue','Green']
states=['a','b','c','d']
neighbours={}
neighbours['a']=['b','c','d']
neighbours['b']=['a','d']
neighbours['c']=['a','d']
neighbours['d']=['c','b','a']
colors_of_states={}
def promising(state,color):
    for neighbour in neighbours.get(state):
        color_of_neighbour=colors_of_states.get(neighbour)
        if color_of_neighbour==color:
            return False
    return True
def get_color_for_state(state):
    for color in colors:
        if promising(state,color):
            return color
def main():
    for state in states:
        colors_of_states[state]=get_color_for_state(state)
    print(colors_of_states)
main()
```

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

```
from sympy import symbols, Or ,Not ,Implies ,Xor,satisfiable
Rain= symbols('Rain');
Harry_Visted_Hagrid = symbols('Harry_Visted_Hagrid')
Harry_Visted_Dumbledore=symbols('Harry_Visted_Dumbledore')
sentence_1= Implies(Not(Rain),Harry_Visted_Hagrid)
sentence_2=Xor(Harry_Visted_Hagrid,Harry_Visted_Dumbledore)
sentence_3=Harry_Visted_Dumbledore
knowledge_base =sentence_1 & sentence_2 & sentence_3
solution =satisfiable(knowledge_base,all_models=True)
for model in solution:
    if model[Rain]:
        print("it rained today:")
    else:
        print("there is no rain today")
```

```
it rained today:
```

```
P_burglary=0.002
P_earthquake=0.001
P_alarm_given_burglary_and_earthquake=0.94
P_alarm_given_bugrlary_and_no_earthquake=0.95
P_alarm_given_no_burglary_and_earthquake=0.31
P_alarm_given_no_burglary_and_no_earthquake=0.001
P_david_calls_given_alarm=0.91
P_david_does_not_call_given_alarm=0.09
P_david_calls_given_no_alarm=0.05
P_david_does_not_call_given_no_alarm=0.95
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
def
joint_probability(alarm,burglary,earthquake,david_calls,sophia_calls):
    if alarm:
        if burglary and earthquake:
            P_alarm=P_alarm_given_burglary_and_earthquake
        elif burglary:
            P_alarm=P_alarm_given_bugrlary_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
    else:
        if burglary and earthquake:
            P_alarm=1-P_alarm_given_burglary_and_earthquake
        elif burglary:
            P_alarm=1-P_alarm_given_bugrlary_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)
    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)
    return (P_burglary if burglary else 1- P_burglary)*(P_earthquake
if earthquake else 1-P_earthquake)*P_alarm*P_david*P_sophia
result=joint_probability(
    alarm=True,
    burglary=False,
    earthquake=False,
    david_calls=True,
```

```
    sophia_calls=True
)
print(f'{result:.8f}')
```

```
0.00068045
```

```
import numpy as np
class HMM:
    def __init__(self , states , observations):
        self.states = states
        self.n_states = len(states)
        self.n_obs = len(observations)
        self.state_index = {state: i for i, state in
enumerate(states)}
        self.obs_index = {obs: i for i, obs in
enumerate(observations)}
        self.A = np.array([[0.6,0.3,0.1],[0.2,0.5,0.3],[0.1,0.4,0.5]])
        self.B = np.array([[0.8,0.15,0.05],[0.3,0.4,0.3],
[0.1,0.2,0.7]])
        self.pi = np.array([0.5,0.3,0.2])
    def forward(self, obs_seq):
        n = len(obs_seq)
        alpha = np.zeros((n, self.n_states))
        alpha[0] = self.pi * self.B[:, obs_seq[0]]
        for t in range(1, n):
            for j in range(self.n_states):
                alpha[t, j] = (alpha[t-1] @ self.A[:, j]) * self.B[j,
obs_seq[t]]
        return alpha.sum(axis=1)[-1]
states = ['Sunny','Cloudy','Rainy']
observations = ['Umbrella','Normal', 'Raincoat']
hmm = HMM(states, observations)
obs_seq = ['Umbrella', 'Normal', 'Umbrella','Raincoat']
obs_seq_indices = [hmm.obs_index[obs] for obs in obs_seq]
prob = hmm.forward(obs_seq_indices)
print(f"Probability of the observation sequence '{obs_seq}':
{prob:.4f}")
```

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

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

dataset=pd.read_csv('Salary_Data.csv')
dataset.head()
```

```
   YearsExperience   Salary
0              1.1  39343.0
1              1.3  46205.0
2              1.5  37731.0
3              2.0  43525.0
4              2.2  39891.0
```

```
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,rando
m_state=0)
x_test
```

```
array([[ 1.5],
       [10.3],
       [ 4.1],
       [ 3.9],
       [ 9.5],
       [ 8.7],
       [ 9.6],
       [ 4. ],
       [ 5.3],
       [ 7.9]])
```

```
x=dataset.iloc[:,:-1].values
y=dataset.iloc[:,1].values
x
```

```
array([[ 1.1],
       [ 1.3],
       [ 1.5],
       [ 2. ],
       [ 2.2],
       [ 2.9],
       [ 3. ],
       [ 3.2],
       [ 3.2],
       [ 3.7],
       [ 3.9],
       [ 4. ],
       [ 4. ],
       [ 4.1],
       [ 4.5],
       [ 4.9],
       [ 5.1],
```

```
       [ 5.3],
       [ 5.9],
       [ 6. ],
       [ 6.8],
       [ 7.1],
       [ 7.9],
       [ 8.2],
       [ 8.7],
       [ 9. ],
       [ 9.5],
       [ 9.6],
       [10.3],
       [10.5]])
```

```
from sklearn.linear_model import LinearRegression
regressor=LinearRegression()
regressor.fit(x_train,y_train)
```

```
LinearRegression()
```

```
y_pred=regressor.predict(x_test)
y_pred
```

```
array([ 40835.10590871, 123079.39940819,  65134.55626083,
63265.36777221,
       115602.64545369, 108125.8914992 , 116537.23969801,
64199.96201652,
        76349.68719258, 100649.1375447 ])
```

```
y_test
```

```
array([ 37731., 122391.,  57081.,  63218., 116969., 109431., 112635.,
        55794.,  83088., 101302.])
```

```
plt.scatter(x_train,y_train,color='red')
plt.plot(x_train,regressor.predict(x_train),color='blue')
plt.title("Salary vs Experience(Training set)")
plt.xlabel("Years of Experience")
plt.ylabel("Salaries")
plt.show()
```


```
plt.scatter(x_test,y_test,color='red')
plt.plot(x_train,regressor.predict(x_train),color='blue')
plt.title("Salary vs Experience(Testing set)")
plt.xlabel("Years of Experience")
plt.ylabel("Salaries")
plt.show()
```


Salary vs Experience(Testing set)
120000
100000
80000
60000
40000
Salaries
2
4
6
8
10
Years of Experience

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

data={
    'Outlook':['Sunny','Sunny','Overcast','Rain'],
    'Humidity':['High','Normal','High','Normal'],
    'PlayTennis':['No','Yes','Yes','Yes']
}

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

```
    Outlook Humidity PlayTennis
0     Sunny     High         No
1     Sunny   Normal        Yes
2  Overcast     High        Yes
3      Rain   Normal        Yes
```

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

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)

def information_gain(df, split_attribute_name, target_attribute_name):
    df_split = df.groupby(split_attribute_name)
    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

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_c
lass)
            tree[best_attr][attr_val]=subtree
        return tree

attribute_names=list(df.columns)
attribute_names.remove('PlayTennis')
attribute_names
```

```
['Outlook', 'Humidity']
```

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

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

```
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

iris=load_iris()
print("Dataset keys:",iris.keys())
print("Target array:\n",iris['target'])

Dataset keys: dict_keys(['data', 'target', 'frame', 'target_names',
'DESCR', 'feature_names', 'filename', 'data_module'])
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 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]

df=pd.DataFrame(iris['data'],columns=iris['feature_names'])
print("Feature Data:\n",df.head())

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

print("Target names:",iris['target_names'])
print("Feature names:",iris['feature_names'])
print("Target array:\n",iris['target'])

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 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 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]
```

```
X=df
y=iris['target']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.33,rand
om_state=42)

knn=KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train,y_train)
y_pred=knn.predict(X_test)
cm_test=confusion_matrix(y_test,y_pred)
print("Confusion Matrix (Test Data):\n",cm_test)
```

```
Confusion Matrix (Test Data):
 [[19  0  0]
 [ 0 15  0]
 [ 0  1 15]]
```

```
accuracy_test=accuracy_score(y_test,y_pred)
print("Confusion prediction on test data:",accuracy_test)
print("Wrong prediction on test data:",1-accuracy_test)
```

```
Confusion prediction on test data: 0.98
Wrong prediction on test data: 0.020000000000000018
```

```
y_train_pred=knn.predict(X_train)
cm_train=confusion_matrix(y_train,y_train_pred)
print("Confusion Matrix (Training Data):\n",cm_train)
```

```
Confusion Matrix (Training Data):
 [[31  0  0]
 [ 0 33  2]
 [ 0  2 32]]
```

```python
import numpy as np
def sigmoid(x):
    return 1/(1+np.exp(-x))
def sigmoid_derivative(x):
    return x*(1-x)
x=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([[0],[1],[1],[0]])
input_layer_neurons=2
hidden_layer_neurons=4
output_layer_neurons=1
epochs=10000
learning_rate=0.1
np.random.seed(42)
wh=np.random.uniform(size=(input_layer_neurons,hidden_layer_neurons))
bh=np.random.uniform(size=(1,hidden_layer_neurons))
wout=np.random.uniform(size=(hidden_layer_neurons,output_layer_neurons
))
bout=np.random.uniform(size=(1,output_layer_neurons))
for epoch in range(epochs):
    hidden_layer_input=np.dot(X,wh)+bh
    hidden_layer_output=sigmoid(hidden_layer_input)
    output_layer_input=np.dot(hidden_layer_output,wout)+bout
    output=sigmoid(output_layer_input)
    error=y-output
    output_layer_gradient=sigmoid_derivative(output)
    d_output=error*output_layer_gradient
    hidden_layer_gradient=sigmoid_derivative(hidden_layer_output)
    d_hidden_layer=d_output.dot(wout.T)*hidden_layer_gradient
    wout+=hidden_layer_output.T.dot(d_output)*learning_rate
    bout+=np.sum(d_output,axis=0,keepdims=True)*learning_rate
    wh+=X.T.dot(d_hidden_layer)*learning_rate
    bh+=np.sum(d_hidden_layer,axis=0,keepdims=True)*learning_rate
    if epoch%1000==0:
        print(f"Epoch {epoch},Error:{np.mean(np.abs(error))}")
print("Final predictions after training:")
print(output)
```

```
Epoch 0,Error:0.49914791405546904
Epoch 1000,Error:0.4989908274224632
Epoch 2000,Error:0.49392112204426847
Epoch 3000,Error:0.46086324847622706
Epoch 4000,Error:0.37081148754970517
Epoch 5000,Error:0.22936859341508165
Epoch 6000,Error:0.14117007926640454
Epoch 7000,Error:0.10187019467760634
Epoch 8000,Error:0.08085064924133509
Epoch 9000,Error:0.06790718296112103
Final predictions after training:
[[0.04690963]
 [0.95663392]
```

```
 [0.92548675]
 [0.07177571]]
```

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

data=np.array([
    [1.5,2.0,1],
    [1.0,1.0,1],
    [2.0,2.5,1],
    [2.5,1.5,1],
    [3.0,1.0,0],
    [3.5,0.5,0],
    [4.0,1.0,0],
    [4.5,1.5,0]
])

X=data[:,:2]
y=data[:,2]
svm_model=SVC(kernel='linear')
svm_model.fit(X,y)
```

```
SVC(kernel='linear')
```

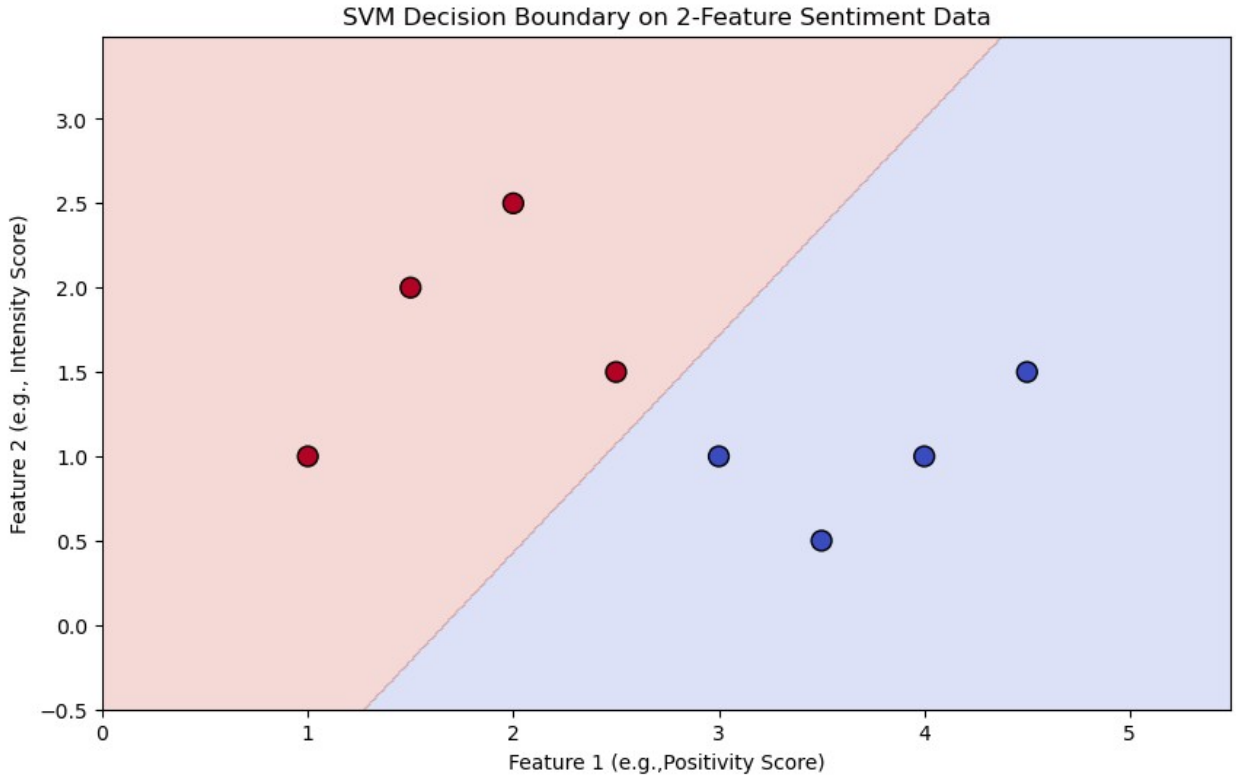
```
y_pred=svm_model.predict(X)
print("Accuracy:",accuracy_score(y,y_pred))
print("\nClassification Report:\n",classification_report(y,y_pred))
```

```
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
```

```
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)
Z=svm_model.predict(np.c_[xx.ravel(),yy.ravel()])
Z=Z.reshape(xx.shape)
print("Z=",Z)
```

```
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.]]
```

```
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()
```


SVM Decision Boundary on 2-Feature Sentiment Data
Feature 2 (e.g., Intensity Score)
Feature 1 (e.g.,Positivity Score)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5