Practical 1:

Aim: Implementation of Simple Search 1

Code:

```
import random

OPEN=['S']
map_list={'S':['A','B','C'],
        'A':['S','D'],
        'B':['S','E']
        'C':['S','F'],
        'D':['A','G'],
        'E':['B','G','F'],
        'F':['C','E'],
        'G':['D','E']}


def movegen(node):
    return map_list[node]
def goaltest(node)

return node=='G'


def ss1():
    while len(OPEN)>0:
        random.shuffle(OPEN)
        N=OPEN.pop()
        if goaltest(N):
            return "Found"
        else:
```

```
            n=movegen(N)
            for i in n:
                if n not in OPEN:
                    OPEN.append(i)
                print("OPEN_LIST",OPEN)
        return "NOT Found"
    print(ss1())
```

Output:

```
OPEN_LIST ['A']
OPEN_LIST ['A', 'B']
OPEN_LIST ['A', 'B', 'C']
OPEN_LIST ['C', 'A', 'S']
OPEN_LIST ['C', 'A', 'S', 'E']
OPEN_LIST ['A', 'C', 'S', 'B']
OPEN_LIST ['A', 'C', 'S', 'B', 'G']
OPEN_LIST ['A', 'C', 'S', 'B', 'G', 'F']
OPEN_LIST ['C', 'F', 'S', 'G', 'B', 'S']
OPEN_LIST ['C', 'F', 'S', 'G', 'B', 'S', 'D']
OPEN_LIST ['C', 'D', 'S', 'B', 'S', 'G', 'C']
OPEN_LIST ['C', 'D', 'S', 'B', 'S', 'G', 'C', 'E']
OPEN_LIST ['S', 'C', 'C', 'G', 'D', 'S', 'B', 'B']
OPEN_LIST ['S', 'C', 'C', 'G', 'D', 'S', 'B', 'B', 'G']
OPEN_LIST ['S', 'C', 'C', 'G', 'D', 'S', 'B', 'B', 'G', 'F']
OPEN_LIST ['G', 'B', 'C', 'S', 'F', 'D', 'G', 'S', 'C', 'S']
OPEN_LIST ['G', 'B', 'C', 'S', 'F', 'D', 'G', 'S', 'C', 'S', 'E']
Found
```

Practical 2:

Aim: Implementation of Simple Search 2

Code:

```
import random

OPEN=['S']

CLOSED=[]

map_list={'S':['A','B','C'],
         'A':['S','D'],
         'B':['S','E'],
         'C':['S','F'],
         'D':['A','G'],
         'E':['B','G','F'],
         'F':['C','E'],
         'G':['D','E']}


def movegen(node):
    return map_list[node]
def goaltest(node):
    return node=='G'


def ss2():
    while len(OPEN)>0:
        random.shuffle(OPEN)
        N=OPEN.pop()
        CLOSED.append(N)
        if goaltest(N):
            return "Found"
```

```
        else:
            n=movegen(N)
            for i in n:
                if i not in OPEN and i not in OPEN:
                    OPEN.append(i)
        print("OPEN_LIST",OPEN)
        print("CLOSED_LIST",CLOSED)
    return "NOT Found"
     print(ss2())
```

Output:

```
OPEN_LIST ['A', 'B', 'C']
CLOSED_LIST ['S']
OPEN_LIST ['C', 'A', 'S', 'E']
CLOSED_LIST ['S', 'B']
OPEN_LIST ['E', 'C', 'S', 'D']
CLOSED_LIST ['S', 'B', 'A']
OPEN_LIST ['D', 'S', 'C', 'B', 'G', 'F']
CLOSED_LIST ['S', 'B', 'A', 'E']
OPEN_LIST ['F', 'C', 'B', 'G', 'S', 'A']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D']
OPEN_LIST ['C', 'G', 'F', 'A', 'B']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S']
OPEN_LIST ['B', 'G', 'A', 'F', 'S']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C']
OPEN_LIST ['S', 'G', 'F', 'A', 'E']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B']
OPEN_LIST ['S', 'F', 'G', 'A', 'B']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E']
OPEN_LIST ['F', 'G', 'S', 'B', 'D']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A']
OPEN_LIST ['F', 'G', 'B', 'S', 'A']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A', 'D']
OPEN_LIST ['A', 'F', 'G', 'B', 'C']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A', 'D', 'S']
OPEN_LIST ['C', 'G', 'A', 'F', 'S', 'E']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A', 'D', 'S', 'B']
Found
```

Output:

```
Sumanth Ganeshan
Roll no - TCS2223077
OPEN_LIST ['A', 'B', 'C']
CLOSED_LIST ['S']
OPEN_LIST ['C', 'A', 'S', 'E']
CLOSED_LIST ['S', 'B']
OPEN_LIST ['E', 'C', 'S', 'D']
CLOSED_LIST ['S', 'B', 'A']
OPEN_LIST ['D', 'S', 'C', 'B', 'G', 'F']
CLOSED_LIST ['S', 'B', 'A', 'E']
OPEN_LIST ['F', 'C', 'B', 'G', 'S', 'A']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D']
OPEN_LIST ['C', 'G', 'F', 'A', 'B']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S']
OPEN_LIST ['B', 'G', 'A', 'F', 'S']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C']
OPEN_LIST ['S', 'G', 'F', 'A', 'E']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B']
OPEN_LIST ['S', 'F', 'G', 'A', 'B']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E']
OPEN_LIST ['F', 'G', 'S', 'B', 'D']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A']
OPEN_LIST ['F', 'G', 'B', 'S', 'A']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A', 'D']
OPEN_LIST ['A', 'F', 'G', 'B', 'C']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A', 'D', 'S']
OPEN_LIST ['C', 'G', 'A', 'F', 'S', 'E']
CLOSED_LIST ['S', 'B', 'A', 'E', 'D', 'S', 'C', 'B', 'E', 'A', 'D', 'S', 'B']
Found
```

Practical 3:

Aim: Implementation of Simple Search 3

Code:

```
import random
```

```
OPEN=[['S',None]]
CLOSED=[]
map_list={'S':['A','B','C'],
         'A':['S','D'],
         'B':['S','E'],
         'C':['S','F'],
         'D':['A','G'],
         'E':['B','G','F'],
         'F':['C','E'],
         'G':['D','E']}

def movegen(node):
    return map_list[node]
def goaltest(node):
    return node=='G'
def returnpath(path):
    if path is not None:
        return str(path[0] + returnpath(path[1]))
    else:
        return ""

def ss3():
    while len(OPEN)>0:
        random.shuffle(OPEN)
        print("Open list",OPEN)
```

```
        M=OPEN.pop()
        N=M[0]
        CLOSED.append(N)
        print("Picked: ",CLOSED)
        if goaltest(N):
            print("Goal Found")
            print("Path: ", returnpath(M)[::-1])
            return
        else:
            neigh=movegen(N)
            for node in neigh:
                if node not in CLOSED and node not in OPEN:
                    new_list=[node,M]
                    OPEN.append(new_list)
    return "NOT Found"
print(ss3())
```

Output:

```
Open list [['S', None]]
Picked:  ['S']
Open list [['C', ['S', None]], ['B', ['S', None]], ['A', ['S', None]]]
Picked:  ['S', 'A']
Open list [['C', ['S', None]], ['B', ['S', None]], ['D', ['A', ['S', None]]]]
Picked:  ['S', 'A', 'D']
Open list [['G', ['D', ['A', ['S', None]]]], ['B', ['S', None]], ['C', ['S', None]]]
Picked:  ['S', 'A', 'D', 'C']
Open list [['G', ['D', ['A', ['S', None]]]], ['F', ['C', ['S', None]]], ['B', ['S', None]]]
Picked:  ['S', 'A', 'D', 'C', 'B']
Open list [['G', ['D', ['A', ['S', None]]]], ['E', ['B', ['S', None]]], ['F', ['C', ['S', None]]]]
Picked:  ['S', 'A', 'D', 'C', 'B', 'F']
Open list [['E', ['B', ['S', None]]], ['G', ['D', ['A', ['S', None]]]], ['E', ['F', ['C', ['S', None]]]]]
Picked:  ['S', 'A', 'D', 'C', 'B', 'F', 'E']
Open list [['G', ['E', ['F', ['C', ['S', None]]]]], ['G', ['D', ['A', ['S', None]]]], ['E', ['B', ['S', None]]]]
Picked:  ['S', 'A', 'D', 'C', 'B', 'F', 'E', 'E']
Open list [['G', ['D', ['A', ['S', None]]]], ['G', ['E', ['F', ['C', ['S', None]]]]], ['G', ['E', ['B', ['S', None]]]]]
Picked:  ['S', 'A', 'D', 'C', 'B', 'F', 'E', 'E', 'G']
Goal Found
Path:  SBEG
None
```

]:

Practical No.4:

Aim:

To write a python program to implement Breadth First Search algorithm.

Code:

BFS:

```
graph={'S':['A','B','C'],
        'A':['S','D'],
        'B':['S','E'],
        'C':['S','F'],
        'D':['A','G'],
        'E':['B','G','F'],
        'F':['C','E'],
        'G':['D','E']}

visited=[]
queue=[]
def bfs(visited,graph,node):
    visited.append(node)
    queue.append(node)
    while queue:
        m=queue.pop(0)
        print(m,end=" ")
        for neigh in  graph[m]:
            if neigh not in visited:
                visited.append(neigh)
                queue.append(neigh)
bfs(visited,graph,'S')
```

Output:

```
S A B C D E F G
```

Practical No.5:

Aim:

To write a python program to implement Depth First Search algorithm.

Code:

```
graph={'S':['A','B','C'],
        'A':['S','D'],
        'B':['S','E'],
        'C':['S','F'],
        'D':['A','G'],
        'E':['B','G','F'],
        'F':['C','E'],
        'G':['D','E']}
visited = []

def dfs(visited,graph,node):
   if node not in visited:
       print(node,end=" ")
       visited.append(node)
       for n in graph[node]:
          dfs(visited,graph,n)
dfs(visited,graph,'S')
```

Output:

```
S A D G E B F C
```

Practical No.7:

Practical No.7:

Aim:

To write a python program to implement Best First Search Algorithm.

Code:

```
map_list={'Mumbai': [('Pune',750),('Delhi',1500),('Goa',1300)],
        'Goa': [('Mumbai',1200)],
        'Delhi':[('Mumbai',1200),('Guwahati',100),('Pune',750)],
        'Chennai':[('Pune',750)],
        'Kolkata':[('Guwahati',100),('Pune',750)],
        'Pune':[('Mumbai',1200),('Kolkata',0),('Chennai',1600),('Delhi',1500)],
        'Guwahati':[('Delhi',1500),('Kolkata',0)]
        }

OPEN=[[('Mumbai',1200),None]]
CLOSED=[]

def movegen(node):
    return map_list[node]

def goaltest(node):
    return node=='Kolkata'

final=[]

def reconstructpath(path):
    if path is None:
        return ""
    else:
        final.append(path[0][0])
        reconstructpath(path[1])
        return final

def sort(a):
    for i in range(len(a)):
        for j in range(0,len(a)-i-1):
            if((a[j][0][1])>a[j+1][0][1]):
                (a[j],a[j+1])=(a[j+1],a[j])
    return a

def best():
    while len(OPEN)>0:
        print("Open List: ",OPEN)
        x=sort(OPEN)
```

```
        seen=x.pop(0)
        N=seen[0][0]
        CLOSED.append(N)
        print("Closed list contains ",CLOSED)
        print("Node Picked: ",N)
        if goaltest(N):
            print(reconstructpath(seen)[::-1])
            return "Found"
        else:
            neigh=movegen(N)
            for i in neigh:
                if i[0] not in CLOSED and i not in OPEN:
                    new=[i,seen]
                    OPEN.append(new)
    return "Not Found"
best()
```

Output:

```
Open List:  [[('Mumbai', 1200), None]]
Closed list contains  ['Mumbai']
Node Picked:  Mumbai
Open List:  [[('Pune', 750), [('Mumbai', 1200), None]], [('Delhi', 1500), [('Mumbai', 1200), None]], [('Goa', 1300), [('Mumba
i', 1200), None]]]
Closed list contains  ['Mumbai', 'Pune']
Node Picked:  Pune
Open List:  [[('Goa', 1300), [('Mumbai', 1200), None]], [('Delhi', 1500), [('Mumbai', 1200), None]], [('Kolkata', 0), [('Pune',
750), [('Mumbai', 1200), None]]], [('Chennai', 1600), [('Pune', 750), [('Mumbai', 1200), None]]], [('Delhi', 1500), [('Pune', 7
50), [('Mumbai', 1200), None]]]]
Closed list contains  ['Mumbai', 'Pune', 'Kolkata']
Node Picked:  Kolkata
['Mumbai', 'Pune', 'Kolkata']

'Found'
```

Practical No.6:

Aim:

To write a python program to implement A* Algorithm.

Code:

```
nodelist = {'mumbai':[('delhi',1200),('nasik',350),('goa',800),('pune',130)],
'delhi':[('nasik',375),('mumbai',1200)],
'nasik':[('indore',600),('delhi',375),('mumbai',350),('nagpur',600)],
'indore':[('nasik',600)],
        'nagpur':[('nasik',600),('pune',450)],
        'pune':[('mumbai',130),('nagpur',450),('blore',550)],
        'blore':[('hyd',110),('goa',750)],
        'goa':[('blore',750),('hyd',850),('mumbai',800)],
        'hyd':[('blore',110),('goa',850)]}

hd={'mumbai':790,'delhi':1515,'nasik':1140,'indore':1540,'nagpur':1110,'pune':660,'blore':110,'goa':85
0,'hyd':0}

#start node mumbai
#end note hyd
#hd for mumbai hf1->delhi->nasik->indore->delhi->infinite loop
#       hf2->nasik->nagpur->blore->hyd->350+600+450+550+110=2060
#       hf3->goa->blore->hyd->800+750+110=1600
#       hf4->pune->blore->hyd->130+550+110=790

openList=[('mumbai',700)]
closedList=[]

def goalTest(node):
  return node=='hyd'
def moveGen(node):
  return nodelist[node[0]]
def sort(mylist):
  for i in range(len(mylist)):
    for j in range(0,len(mylist)-i-1):
      if(mylist[j][1]>mylist[j+1][1]):
        temp=mylist[j]
        mylist[j]=mylist[j+1]
        mylist[j+1]=temp

def AStar():
  while(len(openList)>0):
    sort(openList)
    print("Open List Contains",openList)
```

```
        node=openList.pop(0)
        closedList.append((node[0],hd[node[0]]))
        print("picked node",node)
        if(goalTest(node[0])):
          return "Goal Found"
        else:
          neighbours=moveGen(node)
          print("Neighbours of", node,"are:",neighbours)
          for node in neighbours:

                                    if node not in
              openList and node[0] not in closedList[0]:
              tup=(node[0],(node[1]+hd[node[0]]))
              #tup=(delhi,1200+1515)
              #print(tup)
              openList.append(tup)
    return "Goal Not Found"
AStar()
```

Output:

```
Open List Contains [('mumbai', 700)]
picked node ('mumbai', 700)
Neighbours of ('mumbai', 700) are: [('delhi', 1200), ('nasik', 350), ('goa', 800), ('pune', 130)]
Open List Contains [('pune', 790), ('nasik', 1490), ('goa', 1650), ('delhi', 2715)]
picked node ('pune', 790)
Neighbours of ('pune', 790) are: [('mumbai', 130), ('nagpur', 450), ('blore', 550)]
Open List Contains [('blore', 660), ('nasik', 1490), ('nagpur', 1560), ('goa', 1650), ('delhi', 2715)]
picked node ('blore', 660)
Neighbours of ('blore', 660) are: [('hyd', 110), ('goa', 750)]
Open List Contains [('hyd', 110), ('nasik', 1490), ('nagpur', 1560), ('goa', 1600), ('goa', 1650), ('delhi', 2715)]
picked node ('hyd', 110)

'Goal Found'
```

```
            if node not in openList and node[0] not in closedList[0]:
                tup=(node[0],(node[1]+hd[node[0]]))
                #tup=(delhi,1200+1515)
                #print(tup)
                openList.append(tup)
    return "Goal Not Found"
AStar()
```

Output:

```
nodelist = {'mumbai':[('delhi',1200),('nasik',350),('goa',800),('pune',130)],
        'delhi':[('nasik',375),('mumbai',1200)],
        'nasik':[('indore',600),('delhi',375),('mumbai',350),('nagpur',600)],
        'indore':[('nasik',600)],
        'nagpur':[('nasik',600),('pune',450)],
        'pune':[('mumbai',130),('nagpur',450),('blore',550)],
        'blore':[('hyd',110),('goa',750)],
        'goa':[('blore',750),('hyd',850),('mumbai',800)],
        'hyd':[('blore',110),('goa',850)]}

hd={'mumbai':790,'delhi':1515,'nasik':1140,'indore':1540,'nagpur':1110,'pune':660,'blore':110,'goa':850,'hyd':0}

#start node mumbai
#end note hyd
#hd for mumbai hf1->delhi->nasik->indore->delhi->infinite loop
#         hf2->nasik->nagpur->blore->hyd->350+600+450+550+110=2060
#         hf3->goa->blore->hyd->800+750+110=1600
#         hf4->pune->blore->hyd->130+550+110=790

openList=[('mumbai',700)]
closedList=[]

def goalTest(node):
  return node=='hyd'
def moveGen(node):
  return nodelist[node[0]]
def sort(mylist):
  for i in range(len(mylist)):
    for j in range(0,len(mylist)-i-1):
      if(mylist[j][1]>mylist[j+1][1]):
        temp=mylist[j]
        mylist[j]=mylist[j+1]
        mylist[j+1]=temp

def AStar():
  while(len(openList)>0):
    sort(openList)
    print("Open List Contains",openList)
    node=openList.pop(0)
    closedList.append((node[0],hd[node[0]]))
    print("picked node",node)
    if(goalTest(node[0])):
      return "Goal Found"
    else:
      neighbours=moveGen(node)
      print("Neighbours of", node,"are:",neighbours)
      for node in neighbours:
```

```
            if node not in openList and node[0] not in closedList[0]:
                tup=(node[0],(node[1]+hd[node[0]]))
                #tup=(delhi,1200+1515)
                #print(tup)
                openList.append(tup)
    return "Goal Not Found"
AStar()
```

Output:

Practical No.8:

Aim:

To write a python program to implement Decision Tree Learning.

Code:

```
import numpy as np
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.metrics import accuracy_score
```

```
balance_data = pd.read_csv ('C:/Users/sies/Desktop/balance-scale.data',sep= ',', header= None)

print("Dataset Length:: ", len(balance_data))
print("Dataset Shape:: ", balance_data.shape)
print(balance_data.head())

X = balance_data.values[:,1:5]
Y = balance_data.values[:,0]

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.4, random_state = 100)

clf_entropy = DecisionTreeClassifier(criterion = "entropy", random_state = 100, max_depth=3,
min_samples_leaf=5)
clf_entropy.fit(X_train, y_train)

clf_gini = DecisionTreeClassifier(criterion = "gini", random_state=100, max_depth=3,
min_samples_leaf=5)
clf_gini.fit(X_train, y_train)

print(y_test)

y_pred_en = clf_entropy.predict(X_test)
y_pred_gini = clf_gini.predict(X_test)

print(y_pred_en)
print(y_pred_gini)

accuracy_score(y_pred_en, y_test)*100
```

Output:

```
Dataset Length::  625
Dataset Shape::  (625, 5)
   0  1  2  3  4
0  B  1  1  1  1
1  R  1  1  1  2
2  R  1  1  1  3
3  R  1  1  1  4
4  R  1  1  1  5
['L' 'L' 'R' 'L' 'R' 'B' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'L'
 'B' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L'
 'R' 'L' 'L' 'R' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L'
 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'L' 'R' 'B' 'B' 'R' 'R' 'R' 'L' 'L' 'B'
 'R' 'L' 'R' 'L' 'L' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'B' 'B' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L' 'B' 'L' 'B' 'R' 'L' 'R' 'R' 'R'
 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'B' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R'
 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'R' 'L' 'R' 'B' 'L' 'R' 'R'
 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'B' 'R' 'R' 'L' 'R' 'B' 'L' 'R' 'R'
 'R' 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'B' 'L' 'L'
 'B' 'R' 'L' 'R' 'B' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'L'
 'L' 'R' 'R' 'R' 'R' 'B' 'R' 'B' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R' 'L'
 'R' 'R' 'R' 'B' 'L' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'R']
['L' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L'
 'L' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L'
 'R' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L'
 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'L'
 'R' 'L' 'R' 'L' 'L' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'R' 'L'
 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'
 'L' 'L' 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'R'
 'R' 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'R' 'L' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R'
 'R' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'L' 'R' 'L'
```

```
 'R' 'R' 'L' 'L' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'R']
['R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'B' 'L' 'R' 'B' 'L' 'L' 'R' 'R' 'R' 'L'
 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'B' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L'
 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'R' 'L' 'B' 'R' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'R' 'L' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'B' 'R' 'L' 'L' 'R' 'L' 'R' 'B' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
 'B' 'L' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'L' 'L' 'L' 'R' 'R'
 'B' 'R' 'R' 'L' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'R' 'L' 'L'
 'R' 'R' 'B' 'R' 'R' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'L'
 'L' 'R' 'R' 'B' 'R' 'R' 'R' 'B' 'R' 'L' 'R' 'R' 'L' 'R' 'B' 'L' 'R' 'L'
 'R' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R']
```

```
[9]: 72.39999999999999
```

Practical:9
Aim: To implement Support Vector Machine Algorithm in Python

Code:

```
# Importing required libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Create a synthetic dataset
X, y = datasets.make_classification(n_samples=100, n_features=2, n_classes=2,
n_clusters_per_class=1, n_informative=2, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an SVM classifier with a linear kernel
clf = SVC(kernel='linear')

# Train the SVM classifier
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Calculate accuracy
```

```
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')

# Plot the decision boundary
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 50), np.linspace(ylim[0], ylim[1], 50))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100, linewidth=1,
facecolors='none', edgecolors='k')
plt.title('Support Vector Machine Decision Boundary')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```
