Program 1:

```
def a_star_algo(start_node, stop_node):
    open_set = {start_node}
    closed_set = set()
    g = {start_node: 0}
    parents = {start_node: start_node}

    while open_set:
        n = min(open_set, key=lambda x: g[x] + heuristic(x) if g[x]
else float("inf"))

        if n == stop_node or Graph_nodes.get(n) is None:
            break

        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
            elif g[m] > g[n] + weight:
                g[m], parents[m] = g[n] + weight, n

                if m in closed_set:
                    closed_set.remove(m)
                    open_set.add(m)

        open_set.remove(n)
        closed_set.add(n)

    if n is None:
        print("Path does not exist!")
        return None

    path = [n]
    while parents[n] != n:
        path.append(parents[n])
        n = parents[n]

    path.reverse()
    print("Path found:", path)
    return path


def get_neighbors(v):
    return Graph_nodes.get(v, [])
```

```python
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,
    "E": [("D", 6)],
    "D": [("G", 1)],
}

a_star_algo("A", "G")
```

Output: Path found:[‘A’,’E’,’D’,’G’]

Program 3:

```python
import numpy as np
import pandas as pd

data = pd.DataFrame(data=pd.read_csv("3.csv"))
print(data)
concepts = np.array(data.iloc[:, 0:-1])
print(concepts)
target = np.array(data.iloc[:, -1])
print(target)


def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("initialization of specific_h and general_h")
    print(specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in 
range(len(specific_h))]
    print(general_h)

    for i, h in enumerate(concepts):
        if target[i] == "yes":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = "?"
                    general_h[x][x] = "?"

        if target[i] == "no":
```

```
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = "?"

        print("steps of candidate Elimination Algorithm", i + 1)
        print(specific_h)
        print(general_h)

    indices = [
        i for i, val in enumerate(general_h) if val == ["?", "?", "?",
"?", "?", "?"]
    ]
    for i in indices:
        general_h.remove(["?", "?", "?", "?", "?", "?"])
    return specific_h, general_h


s_final, g_final = learn(concepts, target)
print("\nFinal Specific_h:", s_final, sep="\n")
print("\nFinal General_h:", g_final, sep="\n")
```

Program 5:

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X / np.amax(X, axis=0)
y = y / 100


class Neural_Network(object):
    def __init__(self):
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)

    def forward(self, X):
        self.z = np.dot(X, self.W1)
```

```
        self.z2 = self.sigmoid(self.z)
        self.z3 = np.dot(self.z2, self.W2)
        o = self.sigmoid(self.z3)
        return o

    def sigmoid(self, s):
        return 1 / (1 + np.exp(-s))

    def sigmoidPrime(self, s):
        return s * (1 - s)

    def backward(self, X, y, o):
        self.o_error = y - o  # error in output
        self.o_delta = self.o_error * self.sigmoidPrime(o)
        self.z2_error = self.o_delta.dot(self.W2.T)
        self.z2_delta = self.z2_error * self.sigmoidPrime(self.z2)
        self.W1 += X.T.dot(self.z2_delta)
        self.W2 += self.z2.T.dot(self.o_delta)

    def train(self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)


NN = Neural_Network()
print("\nInput: \n" + str(X))
print("\nActual Output: \n" + str(y))
print("\nPredicted Output: \n" + str(NN.forward(X)))
print("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(X)))))
NN.train(X, y)
```

Program 6:

```
import pandas as pd
from sklearn import tree
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import LabelEncoder

data = pd.read_csv("6.csv")
print("The first 5 values of data is :\n", data.head())

X = data.iloc[:, :-1]
print("\nThe First 5 values of train data is\n", X.head())

y = data.iloc[:, -1]
```

```
print("\nThe first 5 values of Train output is\n", y.head())

le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)

print("\nNow the Train data is :\n", X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n", y)

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.20)

classifier = GaussianNB()
classifier.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

print("Accuracy is:", accuracy_score(classifier.predict(X_test),
y_test))
```

Program 7:

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

```
dataset = load_iris()

X = pd.DataFrame(dataset.data)
X.columns = ["Sepal_Length", "Sepal_Width", "Petal_Length",
"Petal_Width"]
y = pd.DataFrame(dataset.target)
y.columns = ["Targets"]

plt.figure(figsize=(14, 7))
colormap = np.array(["red", "lime", "black"])

plt.subplot(1, 3, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title("Real")

plt.subplot(1, 3, 2)
model = KMeans(n_clusters=3)
model.fit(X)
predY = np.choose(model.labels_, [0, 1, 2]).astype(np.int64)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[predY], s=40)
plt.title("KMeans")

scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns=X.columns)
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
y_cluster_gmm = gmm.predict(xs)
plt.subplot(1, 3, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm],
s=40)
plt.title("GMM Classification")
```

Program 8:

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np

dataset = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    dataset["data"], dataset["target"], random_state=0
```

```
)
kn = KNeighborsClassifier(n_neighbors=1)
kn.fit(X_train, y_train)
for i in range(len(X_test)):
    x = X_test[i]
    x_new = np.array([x])
    prediction = kn.predict(x_new)
    print(
        "TARGET=",
        y_test[i],
        dataset["target_names"][y_test[i]],
        "PREDICTED=",
        prediction,
        dataset["target_names"][prediction],
    )

print(kn.score(X_test, y_test))
```

Program 9:

```
from math import ceil

import numpy as np
from scipy import linalg


def lowess(x, y, f, iterations):
    n = len(x)
    r = int(ceil(f * n))
    h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
    w = np.clip(np.abs((x[:, None] - x[None, :]) / h), 0.0, 1.0)
    w = (1 - w**3) ** 3
    yest = np.zeros(n)
    delta = np.ones(n)
    for iteration in range(iterations):
        for i in range(n):
            weights = delta * w[:, i]
            b = np.array([np.sum(weights * y), np.sum(weights * y *
x)])
            A = np.array(
                [
                    [np.sum(weights), np.sum(weights * x)],
                    [np.sum(weights * x), np.sum(weights * x * x)],
                ]
            )
```

```
            beta = linalg.solve(A, b)
            yest[i] = beta[0] + beta[1] * x[i]

        residuals = y - yest
        s = np.median(np.abs(residuals))
        delta = np.clip(residuals / (6.0 * s), -1, 1)
        delta = (1 - delta**2) ** 2

    return yest


import math

n = 100
x = np.linspace(0, 2 * math.pi, n)
y = np.sin(x) + 0.3 * np.random.randn(n)
f = 0.25
iterations = 3
yest = lowess(x, y, f, iterations)

import matplotlib.pyplot as plt

plt.plot(x, y, "r.")
plt.plot(x, yest, "b-")
```

Program 2:

```
class Graph:
    def __init__(
        self, graph, heuristicNodeList, startNode
    ):  # instantiate graph object with graph topology, heuristic
values, start node
        self.graph = graph
        self.H = heuristicNodeList
        self.start = startNode
        self.parent = {}
        self.status = {}
        self.solutionGraph = {}

    def applyAOStar(self):  # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v):  # gets the Neighbors of a given node
        return self.graph.get(v, "")
```

```
    def getStatus(self, v):  # return the status of a given node
        return self.status.get(v, 0)

    def setStatus(self, v, val):  # set the status of a given node
        self.status[v] = val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n, 0)  # always return the heuristic value of
a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n] = value  # set the revised heuristic value of a given
node

    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE
STARTNODE:", self.start)
        print("---------------------------------------------------------
----")
        print(self.solutionGraph)
        print("---------------------------------------------------------
----")

    def computeMinimumCostChildNodes(
        self, v
    ):  # Computes the Minimum Cost of child nodes of a given node v
        minimumCost = 0
        costToChildNodeListDict = {}
        costToChildNodeListDict[minimumCost] = []
        flag = True
        for nodeInfoTupleList in self.getNeighbors(
            v
        ):  # iterate over all the set of child node/s
            cost = 0
            nodeList = []
            for c, weight in nodeInfoTupleList:
                cost = cost + self.getHeuristicNodeValue(c) + weight
                nodeList.append(c)

            if (
                flag == True
            ):  # initialize Minimum Cost with the cost of first set of
child node/s
                minimumCost = cost
                costToChildNodeListDict[
                    minimumCost
```

```
                ] = nodeList  # set the Minimum Cost child node/s
                flag = False
            else:  # checking the Minimum Cost nodes with the current
Minimum Cost
                if minimumCost > cost:
                    minimumCost = cost
                    costToChildNodeListDict[
                        minimumCost
                    ] = nodeList  # set the Minimum Cost child node/s

        return (
            minimumCost,
            costToChildNodeListDict[minimumCost],
        )  # return Minimum Cost and Minimum Cost child node/s

    def aoStar(
        self, v, backTracking
    ):  # AO* algorithm for a start node and backTracking status flag
        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)

        print(
            "-----------------------------------------------------------
-------------------------------"
        )

        if (
            self.getStatus(v) >= 0
        ):  # if status node v >= 0, compute Minimum Cost nodes of v
            minimumCost, childNodeList =
self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v, len(childNodeList))

            solved = True  # check the Minimum Cost nodes of v are
solved

            for childNode in childNodeList:
                self.parent[childNode] = v
                if self.getStatus(childNode) != -1:
                    solved = solved & False

            if (
                solved == True
```

```
            ):  # if the Minimum Cost nodes of v are solved, set the
current node status as solved(-1)
                self.setStatus(v, -1)
                self.solutionGraph[
                    v
                ] = childNodeList  # update the solution graph with the
solved nodes which may be a part of solution

            if (
                v != self.start
            ):  # check the current node is the start node for
backtracking the current node value
                self.aoStar(
                    self.parent[v], True
                )  # backtracking the current node value with
backtracking status set to true

            if backTracking == False:  # check the current call is not
for backtracking
                for childNode in childNodeList:  # for each Minimum
Cost child node
                    self.setStatus(
                        childNode, 0
                    )  # set the status of child node to 0(needs
exploration)
                    self.aoStar(
                        childNode, False
                    )  # Minimum Cost child node is further explored
with backtracking status as false


h1 = {
    "A": 1,
    "B": 6,
    "C": 2,
    "D": 12,
    "E": 2,
    "F": 1,
    "G": 5,
    "H": 7,
    "I": 7,
    "J": 1,
    "T": 3,
}
graph1 = {
    "A": [[("B", 1), ("C", 1)], [("D", 1)]],
```

```
    "B": [[("G", 1)], [("H", 1)]],
    "C": [[("J", 1)]],
    "D": [[("E", 1), ("F", 1)]],
    "G": [[("I", 1)]],
}
G1 = Graph(graph1, h1, "A")
G1.applyAOStar()
G1.printSolution()

h2 = {
    "A": 1,
    "B": 6,
    "C": 12,
    "D": 10,
    "E": 4,
    "F": 4,
    "G": 5,
    "H": 7,
}  # Heuristic values of Nodes
graph2 = {  # Graph of Nodes and Edges
    "A": [
        [("B", 1), ("C", 1)],
        [("D", 1)],
    ],  # Neighbors of Node 'A', B, C & D with repective weights
    "B": [[("G", 1)], [("H", 1)]],  # Neighbors are included in a list
of lists
    "D": [[("E", 1), ("F", 1)]],  # Each sublist indicate a "OR" node
or "AND" nodes
}

G2 = Graph(
    graph2, h2, "A"
)  # Instantiate Graph object with graph, heuristic values and start
Node
G2.applyAOStar()  # Run the AO* algorithm
G2.printSolution()  # print the solution graph as AO* Algorithm search
```

Program 4:

```
import pandas as pd
import math

def base_entropy(dataset):
    p = 0
    n = 0
```

```python
    target = dataset.iloc[:, -1]
    targets = list(set(target))
    for i in target:
        if i == targets[0]:
            p = p + 1
        else:
            n = n + 1
    if p == 0 or n == 0:
        return 0
    elif p == n:
        return 1
    else:
        entropy = 0 - (
            (
                (p / (p + n)) * (math.log2(p / (p + n)))
                + (n / (p + n)) * (math.log2(n / (p + n)))
            )
        )
        return entropy

def entropy(dataset, feature, attribute):
    p = 0
    n = 0
    target = dataset.iloc[:, -1]
    targets = list(set(target))
    for i, j in zip(feature, target):
        if i == attribute and j == targets[0]:
            p = p + 1
        elif i == attribute and j == targets[1]:
            n = n + 1
        if p == 0 or n == 0:
            return 0
        elif p == n:
            return 1
        else:
            entropy = 0 - (
                (
                    (p / (p + n)) * (math.log2(p / (p + n)))
                    + (n / (p + n)) * (math.log2(n / (p + n)))
                )
            )
            return entropy

def counter(target, attribute, i):
    p = 0
    n = 0
```

```
    targets = list(set(target))
    for j, k in zip(target, attribute):
        if j == targets[0] and k == i:
            p = p + 1
        elif j == targets[1] and k == i:
            n = n + 1
    return p, n

def Information_Gain(dataset, feature):
    Distinct = list(set(feature))
    Info_Gain = 0
    for i in Distinct:
        Info_Gain = Info_Gain + feature.count(i) / len(feature) *
entropy(
            dataset, feature, i
        )
        Info_Gain = base_entropy(dataset) - Info_Gain
    return Info_Gain

def generate_childs(dataset, attribute_index):
    distinct = list(dataset.iloc[:, attribute_index])
    childs = dict()
    for i in distinct:
        childs[i] = counter(dataset.iloc[:, -1], dataset.iloc[:,
attribute_index], i)
    return childs

def modify_data_set(dataset, index, feature, impurity):
    size = len(dataset)
    subdata = dataset[dataset[feature] == impurity]
    del subdata[subdata.columns[index]]
    return subdata

def greatest_information_gain(dataset):
    max = -1
    attribute_index = 0
    size = len(dataset.columns) - 1
    for i in range(0, size):
        feature = list(dataset.iloc[:, i])
        i_g = Information_Gain(dataset, feature)
        if max < i_g:
            max = i_g
            attribute_index = i
    return attribute_index

def construct_tree(dataset, tree):
```

```
    target = dataset.iloc[:, -1]
    impure_childs = []
    attribute_index = greatest_information_gain(dataset)
    childs = generate_childs(dataset, attribute_index)
    tree[dataset.columns[attribute_index]] = childs
    targets = list(set(dataset.iloc[:, -1]))
    for k, v in childs.items():
        if v[0] == 0:
            tree[k] = targets[1]
        elif v[1] == 0:
            tree[k] = targets[0]
        elif v[0] != 0 or v[1] != 0:
            impure_childs.append(k)
    for i in impure_childs:
        sub = modify_data_set(
            dataset, attribute_index, dataset.columns[attribute_index],
i
        )
        tree = construct_tree(sub, tree)
    return tree

def main():
    df = pd.read_csv("filename.csv")
    tree = dict()
    result = construct_tree(df, tree)
    for key, value in result.items():
        print(key, " => ", value)

if __name__ == "__main__":
    main()
```

Program 4.csv file saved as filename.csv:

```
Outlook,Temperature,Humidity,Windy,PlayTennis
Sunny,Hot,High,False,No
Sunny,Hot,High,True,No
Overcast,Hot,High,False,Yes
Rainy,Mild,High,False,Yes
Rainy,Cool,Normal,False,Yes
Rainy,Cool,Normal,True,No
Overcast,Cool,Normal,True,Yes
Sunny,Mild,High,False,No
Sunny,Cool,Normal,False,Yes
Rainy,Mild,Normal,False,Yes
Sunny,Mild,Normal,True,Yes
```

```
Overcast,Mild,High,True,Yes
Overcast,Hot,Normal,False,Yes
Rainy,Mild,High,True,No
```

3.csv:

```
sky,airtemp,humidity,wind,water,forcast,enjoysport
sunny,warm,normal,strong,warm,same,yes
sunny,warm,high,strong,warm,same,yes
rainy,cold,high,strong,warm,change,no
sunny,warm,high,strong,cool,change,yes
```

6.csv:

```
Outlook,Temperature,Humidity,Windy,PlayTennis
Sunny,Hot,High,False,No
Sunny,Hot,High,True,No
Overcast,Hot,High,False,Yes
Rainy,Mild,High,False,Yes
Rainy,Cool,Normal,False,Yes
Rainy,Cool,Normal,True,No
Overcast,Cool,Normal,True,Yes
Sunny,Mild,High,False,No
Sunny,Cool,Normal,False,Yes
Rainy,Mild,Normal,False,Yes
Sunny,Mild,Normal,True,Yes
Overcast,Mild,High,True,Yes
Overcast,Hot,Normal,False,Yes
Rainy,Mild,High,True,No
```