```
from collections import defaultdict
graph=defaultdict(list)

def addEdge(u,v):
    graph[u].append(v)
def dfid(start,goal,max_depth):
    print("Start node us: ",start," Goal node is :",goal)
    for i in range(max_depth):
        print("At level :",i+1)
        print("Path taken is: ",end=" ")
        isfound=dfs(start,goal,i)
        if isfound:
            print("\nNode found !\n")
            return
        else:
            print("\nNode not found!\n")
def dfs(start,goal,depth):
    print(start,end=" ")
    if start==goal:
        return True
    if depth<=0:
        return
    for i in graph[start]:
        if dfs(i,goal,depth-1):
            return True
    return False
goal=defaultdict(list)
addEdge('A','B')
addEdge('A','C')
addEdge('A','D')
addEdge('C','E')
addEdge('C','F')
addEdge('D','G')
addEdge('D','H')
addEdge('G','I')
addEdge('H','K')
addEdge('H','L')
addEdge('I','J')
addEdge('K','O')
addEdge('L','M')
addEdge('M','N')
dfid('A','O',4)
```

```python
SuccList = {
  'S': [['A', 3], ['B', 6], ['C', 5]],
  'A': [['E', 8], ['D', 9]],
  'B': [['G', 14], ['F', 12]],
  'C': [['H', 7]],
  'H': [['J', 6], ['I', 5]],
  'I': [['M', 2], ['L', 10], ['K', 1]]
}

def best_first_search(start, goal):
  open_list = [[start, 5]]
  closed_list = []
  i = 1

  while open_list:
    print(f"\n<<<<<<<<<---({i})-->>>>>>>>>\n")
    n = open_list.pop(0)
    closed_list.append(n)
    print(f"N= {n}")
    print(f"CLOSED= {closed_list}")

    if n[0] == goal:
      return closed_list, True

    children = [child for child in SuccList.get(n[0], []) if child not in open_list and child not
in closed_list]
    print(f"CHILD= {children}")

    open_list.extend(children)
    print(f"Unsorted OPEN= {open_list}")
    open_list.sort(key=lambda x: x[1])
    print(f"Sorted OPEN= {open_list}")

    i += 1

  return closed_list, False

start = input("Enter Source node >> ").upper()
goal = input("Enter Goal node >> ").upper()
path, found = best_first_search(start, goal)

print("Best First Search Path >>>>>", path, "<<<<", found)
```

```
import heapq
def astar(graph, start, goal, heuristic):
    queue = [(0 + heuristic[start], start, [])]  # (priority, node, path)
    cost_so_far = {start: 0}
    iteration_step = 0
    while queue:
        iteration_step += 1
        # Get the node to explore next
        _, current, path = heapq.heappop(queue)
        print("Iteration", iteration_step, "- Current node:", current, "- Path:", path)
        if current == goal:
            return path + [current], iteration_step
        for neighbor, cost in graph[current].items():
            new_cost = cost_so_far[current] + cost
            # Update cost if the new path is cheaper or this is the first time visiting the node
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + heuristic[neighbor]
                heapq.heappush(queue, (priority, neighbor, path + [current]))
                print("   -> Neighbor:", neighbor, "- Path:", path + [current], "- Cost:", new_cost, "- Priority:",
priority)
    return None, iteration_step
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'D': 3, 'E': 4},
    'C': {'F': 2},
    'D': {},
    'E': {'G': 5},
    'F': {},
    'G': {}
}
heuristic = {
    'A': 10,
    'B': 5,
    'C': 8,
    'D': 4,
    'E': 3,
    'F': 2,
    'G': 0
}
start_node = 'A'
goal_node = 'G'
path, iterations = astar(graph, start_node, goal_node, heuristic)
if path:
    print("Path found:", path)
    total_cost = sum(graph[path[i]][path[i+1]] for i in range(len(path)-1))
    print("Total cost:", total_cost)
else:
    print("No path found.")

print("Iterations:", iterations)
```

```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    # Base case: targetDepth reached
    if curDepth == targetDepth:
        return scores[nodeIndex]

    if maxTurn:
        left = minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth)
        right = minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth)
        print("Maximizing node at depth", curDepth, "with value", max(left, right))
        return max(left, right)
    else:
        left = minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth)
        right = minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth)
        print("Minimizing node at depth", curDepth, "with value", min(left, right))
        return min(left, right)

# Driver code
scores = [5, 2, 1, 3, 6, 2, 0, 7]
treeDepth = math.floor(math.log(len(scores), 2))  # Use floor to ensure integer depth

print("The optimal value is:", minimax(0, 0, True, scores, treeDepth))
```

```
import csv
a = []
with open('enjoysport.csv', 'r') as csvfile:
    next(csvfile)  # Skip the header
    for row in csv.reader(csvfile):
        a.append(row)

print(a)
print("\nThe total number of training instances are: ", len(a))

num_attribute = len(a[0]) - 1

hypothesis = ['0'] * num_attribute
print("\nThe initial hypothesis is: ", hypothesis)

for i in range(len(a)):
    if a[i][num_attribute] == 'yes':
        print("\nInstance", i + 1, "is", a[i], "and is a Positive Instance")
        for j in range(num_attribute):
            if hypothesis[j] == '0' or hypothesis[j] == a[i][j]:
                hypothesis[j] = a[i][j]
            else:
                hypothesis[j] = '?'
        print("The hypothesis for the training instance", i + 1, "is: ", hypothesis, "\n")
    else:
        print("\nInstance", i + 1, "is", a[i], "and is a Negative Instance Hence Ignored")
        print("The hypothesis for the training instance", i + 1, "is: ", hypothesis, "\n")
```

```
print("\nThe Maximally specific hypothesis for the training instance is: ", hypothesis)
# OR Gate
def OR():
    w1 = 0
    w2 = 0
    a = 0.2
    t = 0
    X = [[0, 0], [0, 1], [1, 0], [1, 1]]
    Y = [0, 1, 1, 1]
    while True:
        Out = []
        count = 0
        for i in X:
            step = (w1 * i[0] + w2 * i[1])
            if step <= t:
                O = 0
            else:
                O = 1
            if O == Y[count]:
                Out.append(O)
                count += 1
            else:
                if step <= t:
                    w1 += a * i[0]
                    w2 += a * i[1]
                print(w1, w2)
        print("------->")
        if Out == Y:
            print("Final Output of OR ::\n")
            print("Weights: w1={} and w2={} >>>> {}".format(w1, w2, Out))
            break
OR()

def AND():
    w1 = 0
    w2 = 0
    a = 0.2
    t = 1
    X = [[0, 0], [0, 1], [1, 0], [1, 1]]
    Y = [0, 0, 0, 1]
    while True:
        Out = []
        count = 0
        for i in X:
            step = (w1 * i[0] + w2 * i[1])
            if step < t:
                O = 0
            else:
                O = 1
            if O == Y[count]:
                Out.append(O)
                count += 1
            else:
                print('Weights changed to..')
                w1 += a * i[0]
                w2 += a * i[1]
                print("w1={} w2={}".format(round(w1, 2), round(w2, 2)))
            print(w1, w2, Out)
            print("------->")
        if Out == Y:
            print("\nFinal Output of AND::\n")
            print("Weights: w1={} and w2={} >>>> {}".format(round(w1, 2), round(w2, 2), Out))
            break
AND()

def NOT():
    X = [0, 1]
    Y = [1, 0]
    weight = -1
    bias = 1
    Out = []
    for i in X:
        j = weight * i + bias
        if j >= 0:
            Out.append(1)
        else:
            Out.append(0)
    print("\nFinal Output of NOT ::\n")
    for i in range(len(X)):
        print("NOT Gate {}--> {}".format(X[i], Out[i]))
NOT()
import numpy as np
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
expected_output = np.array([[0], [1], [1], [0]])
epochs = 10000
lr = 0.5
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2, 2, 1

# Random weights and bias initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons, hiddenLayerNeurons))
hidden_bias = np.random.uniform(size=(1, hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons, outputLayerNeurons))
output_bias = np.random.uniform(size=(1, outputLayerNeurons))

print("Initial hidden weights: ", end='')
print(*hidden_weights)
print("Initial hidden biases: ", end='')
print(*hidden_bias)
print("Initial output weights: ", end='')
print(*output_weights)
print("Initial output biases: ", end='')
print(*output_bias)

for _ in range(epochs):
    hidden_layer_activation = np.dot(inputs, hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    error = expected_output - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * lr

print("Final hidden weights: ", end='')
print(*hidden_weights)
print("Final hidden bias: ", end='')
print(*hidden_bias)
print("Final output weights: ", end='')
print(*output_weights)
print("Final output bias: ", end='')
print(*output_bias)
print("\nOutput from neural network after epochs :" + str(epochs))
print(*predicted_output)
```

```
x1=[1,1]
x2=[1,-1]
x3=[-1,1]
x4=[-1,-1]
xilist=[x1,x2,x3,x4]
y=[1,-1,-1,-1]

w1=w2=bw=0
b=1
def heb_learn():
    global w1,w2,bw
    print("dw1\tdw2\tdb\tw1\tw2\t")
    i=0
    for xi in xilist:
        dw1=xi[0]*y[i]
        dw2=xi[1]*y[i]
        db=y[i]
        w1=w1+dw1
        w2=w2+dw2
        bw+=db
        print(dw1,dw2,db,w1,w2,bw,sep='\t')
        i+=1
print("Learning...")
heb_learn()
print("Learning completed")
print("Output of AND gate using obtained w1,w2,bw")
print("x1\tx2\ty")
for xi in xilist:
    print(xi[0],xi[1],1 if w1*xi[0]+w2*xi[1]+b*bw>0 else -1,sep='\t')
print("Final weights are: w1= "+str(w1) + " w2= " +str(w2))
```

```
import numpy as np
import matplotlib.pyplot as plt
def euclidean_distance(a, b):
  return np.sqrt(np.sum((a - b) ** 2))
def initialize_centroids(X, k):
  indices = np.random.choice(X.shape[0], k, replace=False)
  return X[indices]
def assign_clusters(X, centroids):
  clusters = []
  for x in X:
    distances = [euclidean_distance(x, centroid) for centroid in centroids]
    cluster = np.argmin(distances)
    clusters.append(cluster)
  return np.array(clusters)

def update_centroids(X, clusters, k):
  new_centroids = []
  for i in range(k):
```

```
        cluster_points = X[clusters == i]
        if len(cluster_points) == 0: # If a cluster is empty, reinitialize its centroid randomly
            new_centroid = X[np.random.choice(X.shape[0])]
        else:
            new_centroid = np.mean(cluster_points, axis=0)
        new_centroids.append(new_centroid)
    return np.array(new_centroids)

def kmeans(X, k, max_iters=100, tol=1e-4):
    centroids = initialize_centroids(X, k)
    for _ in range(max_iters):
        clusters = assign_clusters(X, centroids)
        new_centroids = update_centroids(X, clusters, k)
        if np.all(np.abs(new_centroids - centroids) <= tol):
            break
        centroids = new_centroids
    return centroids, clusters

# Generate sample data
np.random.seed(42)
X = np.vstack([np.random.randn(100, 2) + np.array([3, 3]),
               np.random.randn(100, 2) + np.array([-3, -3]),
               np.random.randn(100, 2) + np.array([-3, 3]),
               np.random.randn(100, 2) + np.array([3, -3])])

# Apply K-means algorithm
k = 4
centroids, clusters = kmeans(X, k)

# Plotting the sample data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1])
plt.title("Sample Data for Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

# Plotting K-means clustering result
plt.figure(figsize=(8, 6))
colors = ['r', 'g', 'b', 'y']
for i in range(k):
    cluster_points = X[clusters == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f'Cluster {i+1}')
plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='red', label='Centroids', marker='X')
plt.title("K-means Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()

print("Final Centroids:")
print(centroids)
```

```
import numpy as np
import pandas as pd

data = {
   'Weather': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Rainy', 'Sunny',
'Overcast', 'Overcast', 'Rainy'],
   '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'],
   'Windy': [False, True, False, False, False, True, True, False, False, False, True, True, False, True],
   'Play': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
}
df = pd.DataFrame(data)

frequency_table = df.groupby(['Play', 'Weather']).size().unstack().fillna(0)
print("Frequency Table:\n", frequency_table)

P_Play = df['Play'].value_counts(normalize=True)
print("\nPrior Probabilities:\n", P_Play)

P_Weather_given_Play = frequency_table.div(frequency_table.sum(axis=1), axis=0)
print("\nLikelihoods:\n", P_Weather_given_Play)

P_Sunny = df['Weather'].value_counts(normalize=True)['Sunny']
P_Sunny_given_Yes = P_Weather_given_Play.loc['Yes', 'Sunny']

P_Yes_given_Sunny = (P_Sunny_given_Yes * P_Play['Yes']) / P_Sunny
print("\nP(Yes|Sunny):", P_Yes_given_Sunny)

class NaiveBayesClassifier:
   def __init__(self):
      self.priors = {}
      self.likelihoods = {}
   def fit(self, X, y):
      data = pd.concat([X, y], axis=1)
      self.priors = y.value_counts(normalize=True)
      self.likelihoods = {col: data.groupby([y.name, col]).size().unstack().fillna(0).div(y.value_counts(), axis=0) for col
in X.columns}
   def predict(self, X):
      results = []
      for i in range(X.shape[0]):
         probs = self.priors.copy()
         for cls in self.priors.index:
            for col in X.columns:
               probs[cls] *= self.likelihoods[col].loc[cls].get(X.iloc[i][col], 0)
         results.append(probs.idxmax())
      return results
X = df[['Weather', 'Temperature', 'Humidity', 'Windy']]
y = df['Play']
model = NaiveBayesClassifier()
model.fit(X, y)
predictions = model.predict(X)
print("\nPredictions:\n", predictions)
example = pd.DataFrame([{'Weather': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Windy': False}])
prediction = model.predict(example)
print("\nExample Prediction for Weather:Sunny, Temperature:Cool, Humidity:High, Windy:False -> Play:",
prediction)
```