**Computer Science (CS):**

Computer Science is the study of algorithms, computation, information, and their implementations through various systems. It encompasses a wide range of topics including software development, programming languages, data structures, algorithms, computer architecture, artificial intelligence, and more. CS involves both theoretical understanding and practical application to design, develop, and analyze computing systems and software solutions.

**Artificial Intelligence (AI):**

Artificial Intelligence refers to the simulation of human intelligence in machines that are programmed to think and perform tasks that typically require human intelligence. It encompasses a broad spectrum of techniques, including machine learning, natural language processing, robotics, computer vision, and more. AI systems aim to replicate cognitive functions such as learning, problem-solving, perception, and decision-making.

**Machine Learning (ML):**

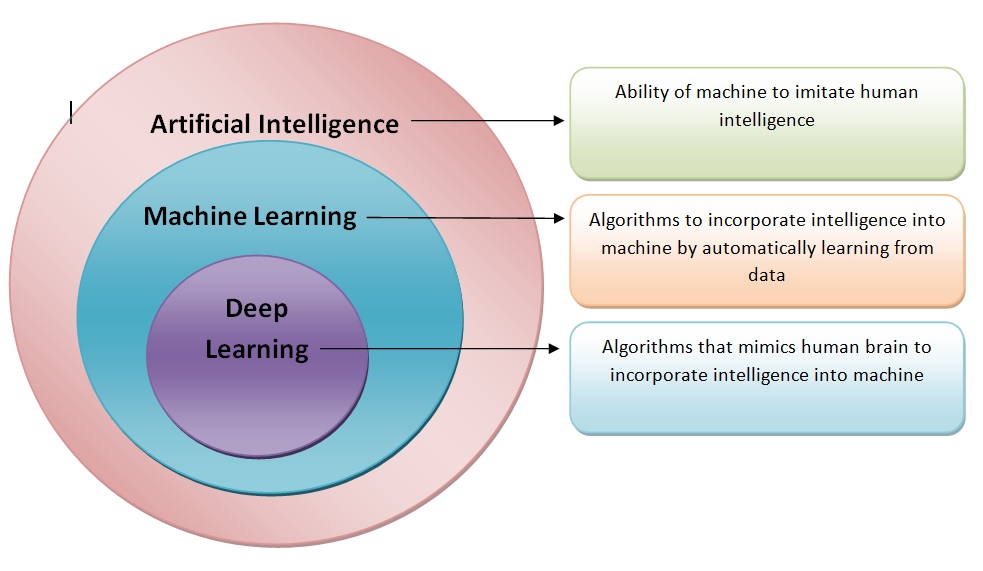
Machine Learning is a subset of artificial intelligence that focuses on enabling machines to learn and improve from experience without being explicitly programmed. ML algorithms use data to identify patterns, make predictions, or optimize processes. It involves various types of learning, including supervised learning (learning from labeled data), unsupervised learning (finding patterns in unlabeled data), and reinforcement learning (learning from feedback).

**Deep Learning (DL):**

Deep Learning is a specialized field of machine learning that uses neural networks with multiple layers (deep neural networks) to extract high-level features from data. DL models are capable of learning representations of data in a hierarchical manner, enabling them to perform tasks such as image and speech recognition, natural language processing, and other complex pattern recognition tasks. Deep Learning has gained attention due to its ability to handle large amounts of data and solve intricate problems.

In summary, Computer Science provides the foundational knowledge and tools for various fields, including AI, which focuses on creating intelligent

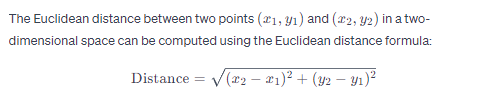
systems. Within AI, Machine Learning employs algorithms to enable machines to learn from data, while Deep Learning, a subset of ML, utilizes complex neural networks for pattern recognition and sophisticated tasks.



1. **Write a program to solve the traveling salesman problem-TSP.**

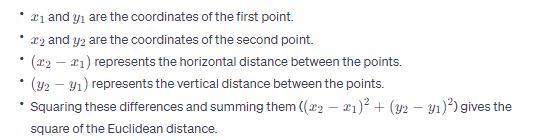
Solution:

The distance function calculates the Euclidean distance between two points in a two-dimensional space. In simple terms, it measures the straight-line distance between two points in a plane.



This formula is derived from the Pythagorean theorem. It calculates the length of the straight line (hypotenuse) between the two points when considering them as the vertices of a right-angled triangle, with the horizontal and vertical distances as the other two sides.

Here's a breakdown of the formula:



Taking the square root of this sum gives the actual Euclidean distance between the points.

This distance calculation is often used in various applications, including geometry, computer science, machine learning (especially in clustering algorithms), and more, to measure the proximity or separation between points in a two-dimensional space.

Example:

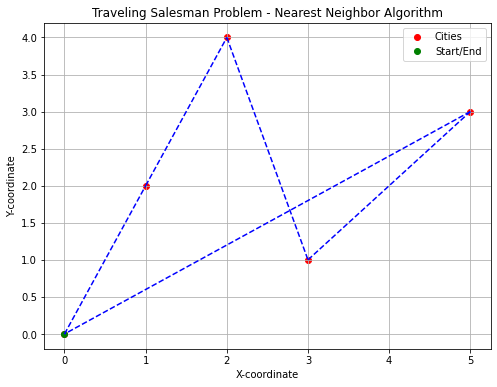
(0, 0),-0

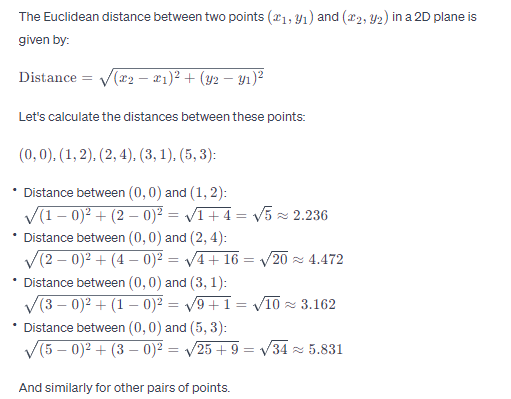
(1, 2),

(2, 4),

(3, 1),

(5, 3)





**Explanation of the Nearest Neighbour Algorithm:**

Initialization:

Choose a starting city arbitrarily. In most cases, the starting city is selected randomly or as the first city in the list of cities.

Iterative Process:

From the current city, find the nearest unvisited city.

Mark that city as visited and add it to the tour.

Update the current city to be the newly visited city.

Repeat this process until all cities are visited.

Completion:

Once all cities are visited, return to the starting city to complete the tour.

Step-by-Step Execution:

**Starting City Selection:**

The algorithm begins at a randomly chosen starting city (in the provided code, it starts from city 0).

**City Selection Process:**

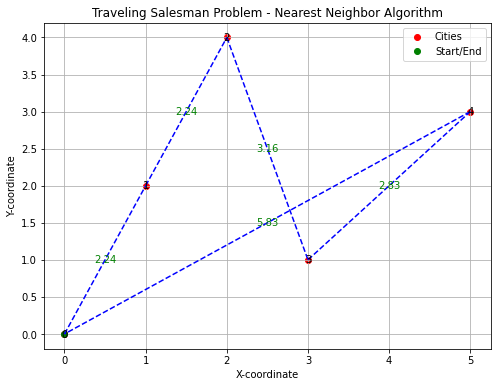
From the current city, the algorithm identifies the nearest unvisited city by calculating the distances between the current city and all unvisited cities.

It selects the city with the shortest distance as the next city to visit and adds it to the tour.

This process continues iteratively until all cities are visited.

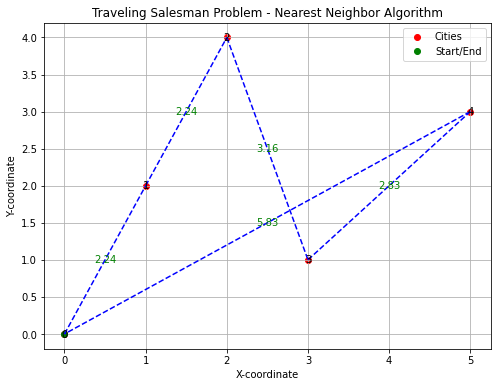
**Returning to the Starting City:**

After visiting all cities, the algorithm returns to the starting city, completing the tour.



|  |
| --- |
| import math  import matplotlib.pyplot as plt  # Function to calculate Euclidean distance between two points  def distance(point1, point2):  return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)  # Nearest Neighbor algorithm for solving TSP  def nearest\_neighbor(cities):  num\_cities = len(cities)  visited = [False] \* num\_cities  path = [0] # Start from city 0  visited[0] = True  total\_distance = 0  distances = []  for i in range(num\_cities - 1):  current\_city = path[-1]  min\_dist = float('inf')  nearest\_city = None  for i in range(num\_cities):  if not visited[i] and i != current\_city:  dist = distance(cities[current\_city], cities[i])  if dist < min\_dist:  min\_dist = dist  nearest\_city = i  distances.append(min\_dist)  total\_distance += min\_dist  path.append(nearest\_city)  visited[nearest\_city] = True  # Returning to the starting city  path.append(0)  total\_distance += distance(cities[path[-2]], cities[0])  distances.append(distance(cities[path[-2]], cities[0]))  return path, total\_distance, distances  # Example cities represented by their coordinates (x, y)  cities = [  (0, 0),  (1, 2),  (2, 4),  (3, 1),  (5, 3)  ]  # Solve TSP using nearest neighbor algorithm  optimal\_path, total\_distance, distances = nearest\_neighbor(cities)  # Extract x and y coordinates for plotting  x\_coords = [city[0] for city in cities]  y\_coords = [city[1] for city in cities]  # Plotting the cities  plt.figure(figsize=(8, 6))  plt.scatter(x\_coords, y\_coords, c='red', label='Cities')  plt.plot(x\_coords[0], y\_coords[0], 'go', label='Start/End') # Mark the starting city  # Plotting the optimal path  for i in range(len(optimal\_path) - 1):  city1 = cities[optimal\_path[i]]  city2 = cities[optimal\_path[i + 1]]  plt.plot([city1[0], city2[0]], [city1[1], city2[1]], 'b--')  # Annotate the cities with their indices and distances  for i, (x, y) in enumerate(cities):  plt.text(x, y, f'{i}', ha='center', va='center')  # Show distances between cities  for i, dist in enumerate(distances):  city1 = cities[optimal\_path[i]]  city2 = cities[optimal\_path[i + 1]]  mid\_x = (city1[0] + city2[0]) / 2  mid\_y = (city1[1] + city2[1]) / 2  plt.text(mid\_x, mid\_y, f'{dist:.2f}', color='green', ha='center', va='center')  plt.title('Traveling Salesman Problem - Nearest Neighbor Algorithm')  plt.xlabel('X-coordinate')  plt.ylabel('Y-coordinate')  plt.legend()  plt.grid(visible=True)  plt.show()  print("Optimal Path:", optimal\_path)  print("Total Distance:", total\_distance)  print("Distances between cities:", distances) |

Output:



Optimal Path: [0, 1, 2, 3, 4, 0]

Total Distance: 16.29379263475945

Distances between cities: [2.23606797749979, 2.23606797749979, 3.1622776601683795, 2.8284271247461903, 5.830951894845301]

Example-2:

|  |
| --- |
| import matplotlib.pyplot as plt  from itertools import permutations  V = 4  # Function to find all possible paths and the optimal path  def find\_paths(graph, s):  min\_path = float('inf')  optimal\_path = []  all\_paths = []  # Generate all permutations of cities  perm = permutations(range(V))    for p in perm:  current\_pathweight = 0  for i in range(V - 1):  current\_pathweight += graph[p[i]][p[i + 1]]  current\_pathweight += graph[p[-1]][p[0]] # Add return to start city    all\_paths.append((list(p), current\_pathweight))    if current\_pathweight < min\_path:  min\_path = current\_pathweight  optimal\_path = list(p) + [p[0]]    return min\_path, optimal\_path, all\_paths  # Function to plot the graph and paths  def plot\_paths(graph, paths, optimal\_path):  x\_coords = [0, 10, 15, 20]  y\_coords = [10, 0, 35, 25]    plt.figure(figsize=(8, 6))  plt.scatter(x\_coords, y\_coords, c='red', label='Cities')  plt.plot(x\_coords[0], y\_coords[0], 'go', label='Start/End') # Mark the starting city  for path, weight in paths:  if path != optimal\_path:  for i in range(V - 1):  plt.plot([x\_coords[path[i]], x\_coords[path[i + 1]]], [y\_coords[path[i]], y\_coords[path[i + 1]]], 'k--')    for i in range(V - 1):  plt.plot([x\_coords[optimal\_path[i]], x\_coords[optimal\_path[i + 1]]], [y\_coords[optimal\_path[i]], y\_coords[optimal\_path[i + 1]]], 'g--')  plt.title('Traveling Salesman Problem - All Paths')  plt.xlabel('X-coordinate')  plt.ylabel('Y-coordinate')  plt.legend()  plt.grid(visible=True)  plt.show()  # Driver Code  if \_\_name\_\_ == "\_\_main\_\_":  graph = [  [0, 10, 15, 20],  [10, 0, 35, 25],  [15, 35, 0, 30],  [20, 25, 30, 0]  ]  start\_city = 0  min\_dist, optimal\_path, all\_paths = find\_paths(graph, start\_city)    print("\nOptimal Path:", optimal\_path)  print("Total Distance of Optimal Path:", min\_dist)  print("\nAll Paths and their Distances:")  for path, dist in all\_paths:  print(f"Path: {path}, Distance: {dist}")  plot\_paths(graph, all\_paths, optimal\_path) |

OUTPUT:

Optimal Path: [0, 1, 3, 2, 0]

Total Distance of Optimal Path: 80

All Paths and their Distances:

Path: [0, 1, 2, 3], Distance: 95

Path: [0, 1, 3, 2], Distance: 80

Path: [0, 2, 1, 3], Distance: 95

Path: [0, 2, 3, 1], Distance: 80

Path: [0, 3, 1, 2], Distance: 95

Path: [0, 3, 2, 1], Distance: 95

Path: [1, 0, 2, 3], Distance: 80

Path: [1, 0, 3, 2], Distance: 95

Path: [1, 2, 0, 3], Distance: 95

Path: [1, 2, 3, 0], Distance: 95

Path: [1, 3, 0, 2], Distance: 95

Path: [1, 3, 2, 0], Distance: 80

Path: [2, 0, 1, 3], Distance: 80

Path: [2, 0, 3, 1], Distance: 95

Path: [2, 1, 0, 3], Distance: 95

Path: [2, 1, 3, 0], Distance: 95

Path: [2, 3, 0, 1], Distance: 95

Path: [2, 3, 1, 0], Distance: 80

Path: [3, 0, 1, 2], Distance: 95

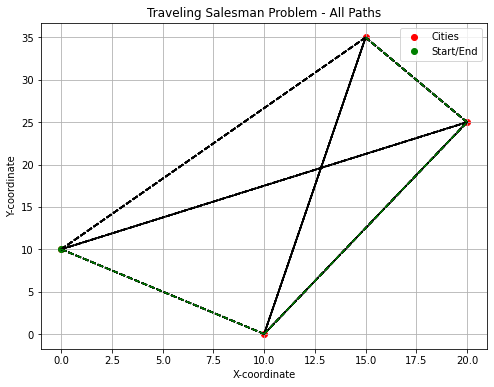
Path: [3, 0, 2, 1], Distance: 95

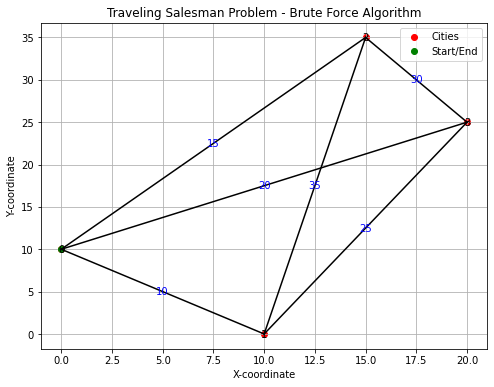
Path: [3, 1, 0, 2], Distance: 80

Path: [3, 1, 2, 0], Distance: 95

Path: [3, 2, 0, 1], Distance: 80

Path: [3, 2, 1, 0], Distance: 95



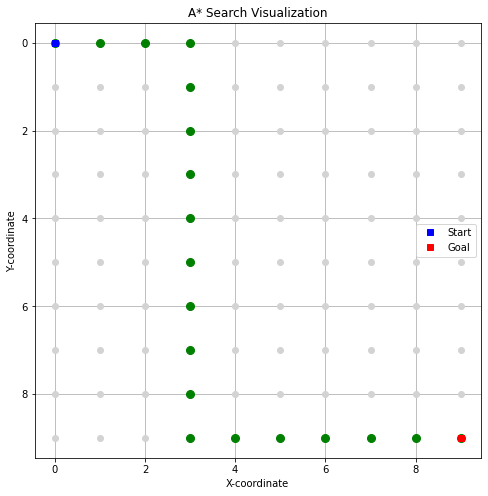


|  |
| --- |
| 1. Implement A\* Search algorithm.   Solution:  The A\* (A-star) search algorithm is a widely used graph traversal and pathfinding algorithm that efficiently finds the shortest path between nodes in a graph, considering both the actual cost from the start node and the heuristic estimate to the goal node.  The A\* (A-star) algorithm is a pathfinding algorithm commonly used for finding the shortest path between two points in a graph. It's widely applied in various fields, especially in robotics, gaming, and navigation systems. A\* efficiently combines the features of Dijkstra's algorithm (which guarantees the shortest path) and greedy best-first search (which is faster but doesn't guarantee optimality).  Implementation with Python  we are going to find out how the A\* search algorithm can be used to find the most cost-effective path in a graph. Consider the following graph above.  The numbers written on edges represent the distance between the nodes, while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using the A\* Algorithm.  Let’s start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero, and from the graph, we get the heuristic value of A is 11, therefore |

|  |
| --- |
| import heapq  import matplotlib.pyplot as plt  class Node:  def \_\_init\_\_(self, state, parent=None, cost=0, heuristic=0):  self.state = state  self.parent = parent  self.cost = cost  self.heuristic = heuristic  def \_\_lt\_\_(self, other):  return (self.cost + self.heuristic) < (other.cost + other.heuristic)  def a\_star\_search(start, goal, neighbors, heuristic):  open\_set = []  closed\_set = set()  start\_node = Node(start, None, 0, heuristic(start, goal))  heapq.heappush(open\_set, start\_node)  while open\_set:  current\_node = heapq.heappop(open\_set)  if current\_node.state == goal:  path = []  while current\_node:  path.append(current\_node.state)  current\_node = current\_node.parent  return path[::-1]  closed\_set.add(current\_node.state)  for neighbor in neighbors(current\_node.state):  if neighbor in closed\_set:  continue    neighbor\_node = Node(neighbor, current\_node, current\_node.cost + 1, heuristic(neighbor, goal))    for node in open\_set:  if node.state == neighbor and node.cost <= neighbor\_node.cost:  break  else:  heapq.heappush(open\_set, neighbor\_node)  return None  # Example heuristic function (Manhattan distance for grid-like environments)  def manhattan\_distance(state, goal):  return abs(state[0] - goal[0]) + abs(state[1] - goal[1])  # Example function to get neighbors of a node in a grid  def get\_neighbors(state):  x, y = state  neighbors = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)] # For a 4-connected grid  return [(nx, ny) for nx, ny in neighbors if 0 <= nx < 10 and 0 <= ny < 10] # Example grid size  # Function to visualize grid and path  def visualize\_grid(start, goal, path):  plt.figure(figsize=(8, 8))  plt.title('A\* Search Visualization')  plt.xlabel('X-coordinate')  plt.ylabel('Y-coordinate')  for i in range(10): # Example grid size  for j in range(10):  plt.plot(i, j, 'o', color='lightgray') # Plot grid cells  for node in path:  plt.plot(node[0], node[1], 'go', markersize=8) # Plot path nodes    plt.plot(start[0], start[1], 'bs', label='Start') # Plot start node  plt.plot(goal[0], goal[1], 'rs', label='Goal') # Plot goal node  plt.legend()  plt.grid(visible=True)  plt.gca().invert\_yaxis() # Invert y-axis to match grid representation  plt.show()  # Example usage:  start\_state = (0, 0)  goal\_state = (9, 9)  path = a\_star\_search(start\_state, goal\_state, get\_neighbors, manhattan\_distance)  if path:  print("Path found:", path)  visualize\_grid(start\_state, goal\_state, path)  else:  print("No path found.") |

Output:

Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9)]



# AO\* algorithm – Artificial intelligence

# 

In the above figure, the**buying of a car** may be broken down into smaller problems or tasks that can be accomplished **to achieve the main goal** in the above figure, which is an example of a simple AND-OR graph. The other task is to either steal a car that will help us accomplish the main goal or use your own money to purchase a car that will accomplish the main goal. The AND symbol is used to indicate the AND part of the graphs, which refers to the need that all subproblems containing the AND to be resolved before the preceding node or issue may be finished.

                                      The start state and the target state are already known in the knowledge-based search strategy known as the **AO\* algorithm**, and the best path is identified by heuristics. The informed search technique considerably reduces the algorithm’s **time complexity**. The AO\* algorithm is far more effective in searching AND-OR trees **than** the A\* algorithm.

# 

## Difference between the A\* Algorithm and AO\* algorithm

* A\* algorithm and AO\* algorithm both works onthe**best first search**.
* They are both **informed search** and works on given heuristics values.
* **A\*** always **gives** the **optimal solution** but AO\* doesn’t guarantee to give the optimal solution.
* Once AO\* got a solution **doesn’t explore** all possible paths but A\* explores all paths.
* When compared to the A\* algorithm, the AO\* algorithm uses **less memory.**
* opposite to the A\* algorithm, the AO\* algorithm cannot go into an endless **loop.**

## Example:

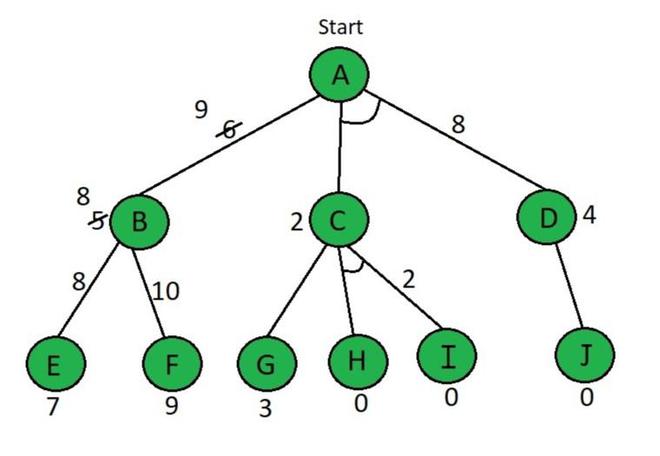
# AO* Algorithm (Questions) -GeeksforgeeksHere in the above example below the Node which is given is the heuristic value i.e **h(n)**. Edge length is considered as**1**.

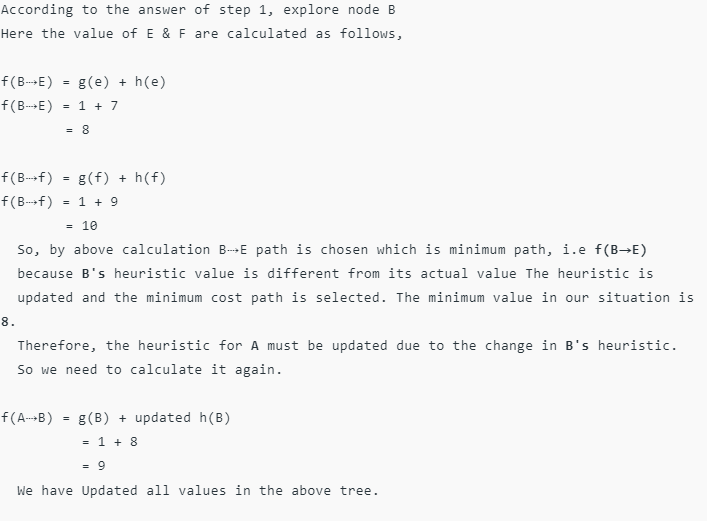
# Step-1

# 

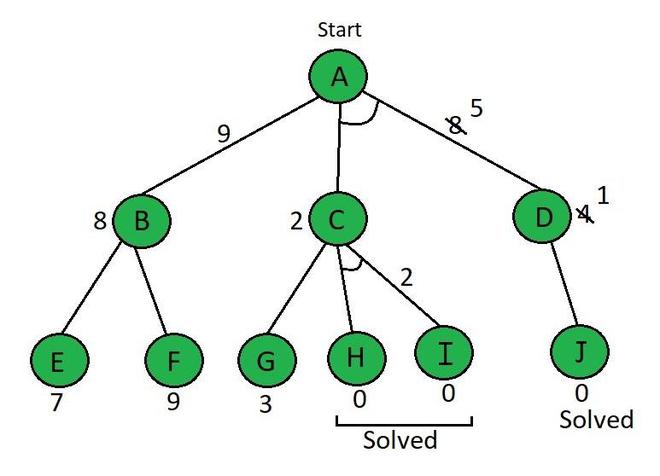
# 

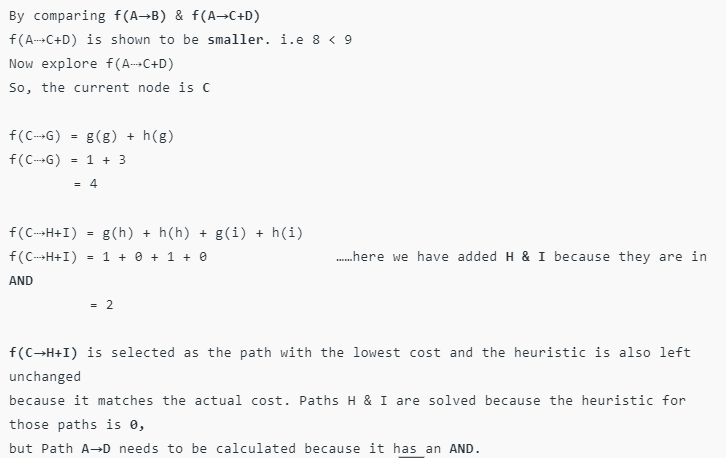
### **Step 2**

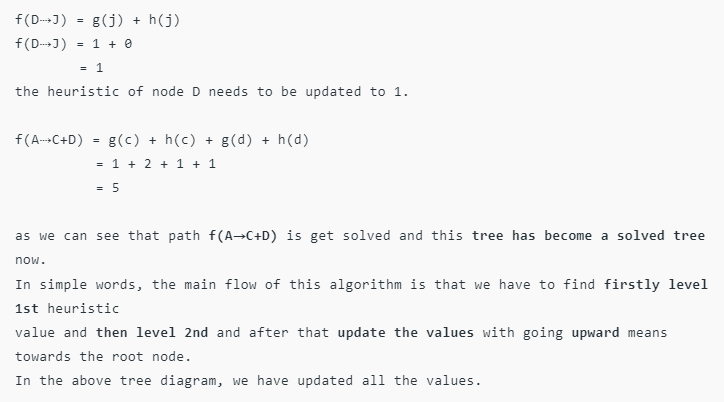




### Step 3



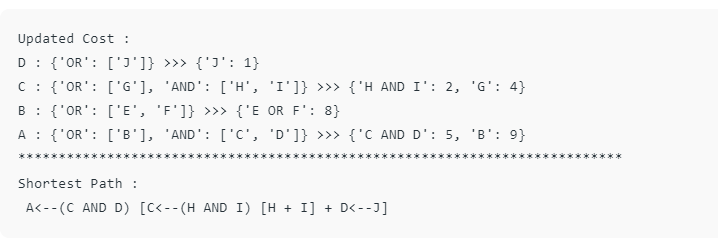




Code

|  |
| --- |
| # Cost to find the AND and OR path  def Cost(H, condition, weight = 1):  cost = {}  if 'AND' in condition:  AND\_nodes = condition['AND']  Path\_A = ' AND '.join(AND\_nodes)  PathA = sum(H[node]+weight for node in AND\_nodes)  cost[Path\_A] = PathA  if 'OR' in condition:  OR\_nodes = condition['OR']  Path\_B =' OR '.join(OR\_nodes)  PathB = min(H[node]+weight for node in OR\_nodes)  cost[Path\_B] = PathB  return cost  # Update the cost  def update\_cost(H, Conditions, weight=1):  Main\_nodes = list(Conditions.keys())  Main\_nodes.reverse()  least\_cost= {}  for key in Main\_nodes:  condition = Conditions[key]  print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))  c = Cost(H, condition, weight)  H[key] = min(c.values())  least\_cost[key] = Cost(H, condition, weight)  return least\_cost  # Print the shortest path  def shortest\_path(Start,Updated\_cost, H):  Path = Start  if Start in Updated\_cost.keys():  Min\_cost = min(Updated\_cost[Start].values())  key = list(Updated\_cost[Start].keys())  values = list(Updated\_cost[Start].values())  Index = values.index(Min\_cost)    # FIND MINIMIMUM PATH KEY  Next = key[Index].split()  # ADD TO PATH FOR OR PATH  if len(Next) == 1:  Start =Next[0]  Path += '<--' +shortest\_path(Start, Updated\_cost, H)  # ADD TO PATH FOR AND PATH  else:  Path +='<--('+key[Index]+') '  Start = Next[0]  Path += '[' +shortest\_path(Start, Updated\_cost, H) + ' + '  Start = Next[-1]  Path += shortest\_path(Start, Updated\_cost, H) + ']'  return Path      H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}  Conditions = {  'A': {'OR': ['B'], 'AND': ['C', 'D']},  'B': {'OR': ['E', 'F']},  'C': {'OR': ['G'], 'AND': ['H', 'I']},  'D': {'OR': ['J']}  }  # weight  weight = 1  # Updated cost  print('Updated Cost :')  Updated\_cost = update\_cost(H, Conditions, weight=1)  print('\*'\*75)  print('Shortest Path :\n',shortest\_path('A', Updated\_cost,H)) |

Output:



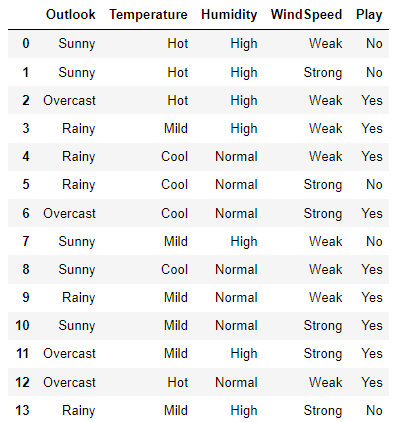
1. Write a program to demonstrate the working of the decision tree-based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

|  |
| --- |
| 1. import math 2. import pandas as pd 3. from operator import itemgetter |

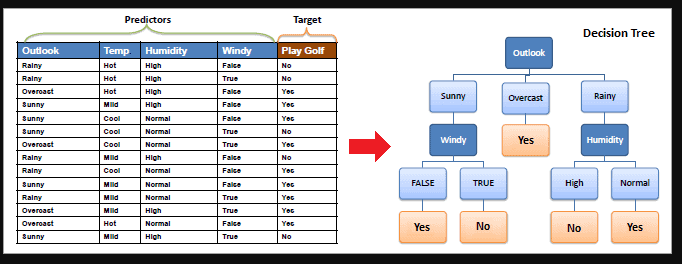
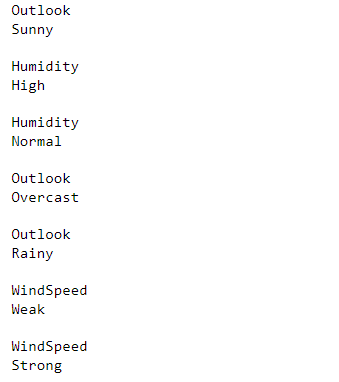
|  |
| --- |
| class DecisionTree:  def \_\_init\_\_(self, df, target, positive, parent\_val, parent):  self.data = df  self.target = target  self.positive = positive  self.parent\_val = parent\_val  self.parent = parent  self.childs = []  self.decision = ''  def \_get\_entropy(self, data):  p = sum(data[self.target]==self.positive)  n = data.shape[0] - p  p\_ratio = p/(p+n)  n\_ratio = 1 - p\_ratio  entropy\_p = -p\_ratio\*math.log2(p\_ratio) if p\_ratio != 0 else 0  entropy\_n = - n\_ratio\*math.log2(n\_ratio) if n\_ratio !=0 else 0  return entropy\_p + entropy\_n    def \_get\_gain(self, feat):  avg\_info=0  for val in self.data[feat].unique():  avg\_info+=self.\_get\_entropy(self.data[self.data[feat] == val])\*sum(self.data[feat]==val)/self.data.shape[0]  return self.\_get\_entropy(df) - avg\_info    def \_get\_splitter(self):  self.splitter = max(self.gains, key = itemgetter(1))[0]    def update\_nodes(self):  self.features = [col for col in self.data.columns if col != self.target]  self.entropy = self.\_get\_entropy(self.data)  if self.entropy != 0:  self.gains = [(feat, self.\_get\_gain(feat)) for feat in self.features]  self.\_get\_splitter()  residual\_columns = [k for k in self.data.columns if k != self.splitter]  for val in self.data[self.splitter].unique():  df\_tmp = self.data[self.data[self.splitter]==val][residual\_columns]  tmp\_node = DecisionTree(df\_tmp, self.target, self.positive, val, self.splitter)  tmp\_node.update\_nodes()  self.childs.append(tmp\_node) |

|  |
| --- |
| def print\_tree(n):  for child in n.childs:  if child:  print(child.\_\_dict\_\_.get('parent', ''))  print(child.\_\_dict\_\_.get('parent\_val', ''), '\n')  print\_tree(child) |

|  |
| --- |
| df = pd.read\_csv('id3.csv')  df |



|  |
| --- |
| dt = DecisionTree(df, 'Play', 'Yes', '', '')  dt.update\_nodes()  print\_tree(dt) |



**6. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets**

**Solution:**

Artificial Neural Network (ANN) using the backpropagation algorithm:

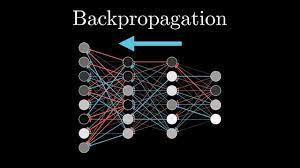
An Artificial Neural Network (ANN) is a computational model inspired by the way biological neural networks in the human brain operate. It consists of interconnected nodes, called neurons, organized in layers. In a typical ANN, there are three types of layers:

Input Layer: This layer receives input signals and passes them on to the next layer. The number of neurons in the input layer corresponds to the number of input features in your dataset.

Hidden Layers: These layers process the input data through weighted connections from the previous layer, applying an activation function to the weighted sum of inputs. Hidden layers allow neural networks to learn complex patterns in the data.

Output Layer: This layer produces the final output of the neural network. The number of neurons in the output layer depends on the nature of the problem you are trying to solve (e.g., regression, classification).

The backpropagation algorithm is a method used to train neural networks. It works by iteratively adjusting the weights of the connections in the network to minimize the difference between the predicted output and the actual output. This is done by computing the gradient of the loss function with respect to the weights using the chain rule of calculus and updating the weights in the direction that reduces the loss.

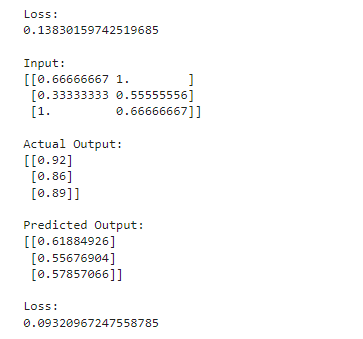
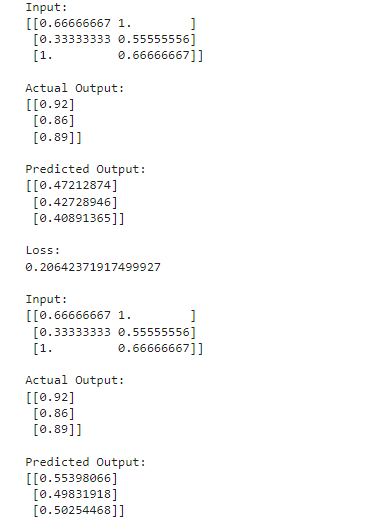


|  |
| --- |
| import numpy as np  X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float) # X = (hours sleeping, hours studying)  y = np.array(([92], [86], [89]), dtype=float) # y = score on test  # scale units  X = X/np.amax(X, axis=0) # maximum of X array  y = y/100 # max test score is 100 |
|  |

|  |
| --- |
| class Neural\_Network(object):  def \_\_init\_\_(self):  # Parameters  self.inputSize = 2  self.outputSize = 1  self.hiddenSize = 3  # Weights  self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden layer  self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer  def forward(self, X):  #forward propagation through our network  self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights  self.z2 = self.sigmoid(self.z) # activation function  self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights  o = self.sigmoid(self.z3) # final activation function  return o  def sigmoid(self, s):  return 1/(1+np.exp(-s)) # activation function  def sigmoidPrime(self, s):  return s \* (1 - s) # derivative of sigmoid    def backward(self, X, y, o):  # backward propgate through the network  self.o\_error = y - o # error in output  self.o\_delta = self.o\_error\*self.sigmoidPrime(o) # applying derivative of sigmoid to  self.z2\_error = self.o\_delta.dot(self.W2.T) # z2 error: how much our hidden layer weights contributed to output error  self.z2\_delta = self.z2\_error\*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2 error  self.W1 += X.T.dot(self.z2\_delta) # adjusting first set (input --> hidden) weights  self.W2 += self.z2.T.dot(self.o\_delta) # adjusting second set (hidden --> output) weights  def train (self, X, y):  o = self.forward(X)  self.backward(X, y, o) |

|  |
| --- |
| NN = Neural\_Network()  for i in range(1000): # trains the NN 1,000 times  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))))) # mean sum squared loss)  NN.train(X, y) |

Output:



7.Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

# import necessary libarities

import pandas as pd

from sklearn import tree

from sklearn.preprocessing import LabelEncoder

from sklearn.naive\_bayes import GaussianNB

# load data from CSV

data = pd.read\_csv('tennisdata.csv')

print("THe first 5 values of data is :\n",data.head())

The First 5 values of train data is

Outlook Temperature Humidity Windy

0 Sunny Hot High False

1 Sunny Hot High True

2 Overcast Hot High False

3 Rainy Mild High False

4 Rainy Cool Normal False

y = data.iloc[:,-1]

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

The first 5 values of Train output is

0 No

1 No

2 Yes

3 Yes

4 Yes

Name: PlayTennis, dtype: object

# Convert then in numbers

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())

Now the Train data is :

Outlook Temperature Humidity Windy

0 2 1 0 0

1 2 1 0 1

2 0 1 0 0

3 1 2 0 0

4 1 0 1 0

le\_PlayTennis = LabelEncoder()

y = le\_PlayTennis.fit\_transform(y)

print("\nNow the Train output is\n",y)

Now the Train output is

[0 0 1 1 1 0 1 0 1 1 1 1 1 0]

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

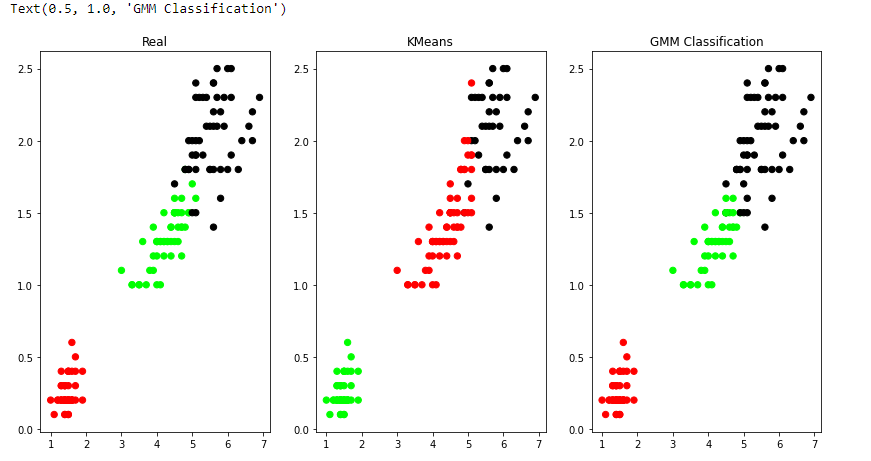
print("Accuracy is:",accuracy\_score(classifier.predict(X\_test),y\_test))

Accuracy is: 0.6666666666666666

1. **Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.**

|  |
| --- |
| 1. from sklearn.cluster import KMeans 2. from sklearn import preprocessing 3. from sklearn.mixture import GaussianMixture 4. from sklearn.datasets import load\_iris 5. import sklearn.metrics as sm 6. import pandas as pd 7. import numpy as np 8. import matplotlib.pyplot as plt 9. dataset=load\_iris() 10. # print(dataset) 11. X=pd.DataFrame(dataset.data) 12. X.columns=['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width'] 13. y=pd.DataFrame(dataset.target) 14. y.columns=['Targets'] 15. # print(X) 16. plt.figure(figsize=(14,7)) 17. colormap=np.array(['red','lime','black']) 18. # REAL PLOT 19. plt.subplot(1,3,1) 20. plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y.Targets],s=40) 21. plt.title('Real') 22. # K-PLOT 23. plt.subplot(1,3,2) 24. model=KMeans(n\_clusters=3) 25. model.fit(X) 26. predY=np.choose(model.labels\_,[0,1,2]).astype(np.int64) 27. plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[predY],s=40) 28. plt.title('KMeans') 29. # GMM PLOT 30. scaler=preprocessing.StandardScaler() 31. scaler.fit(X) 32. xsa=scaler.transform(X) 33. xs=pd.DataFrame(xsa,columns=X.columns) 34. gmm=GaussianMixture(n\_components=3) 35. gmm.fit(xs) 36. y\_cluster\_gmm=gmm.predict(xs) 37. plt.subplot(1,3,3) 38. plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y\_cluster\_gmm],s=40) 39. plt.title('GMM Classification') |

Output:



9. **Write a program to implement the k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.**

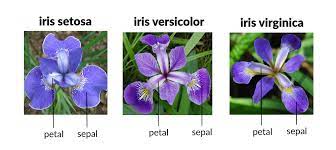
Answer:

The Iris dataset is a classic dataset in machine learning and statistics. It was introduced by the British biologist and statistician Ronald Fisher in his 1936 paper "The use of multiple measurements in taxonomic problems" as an example of linear discriminant analysis. The dataset consists of 150 samples of iris flowers from three different species: Setosa, Versicolor, and Virginica. For each sample, four features are measured: the length and width of the sepals and petals, in centimeters.

The k-Nearest Neighbors (k-NN) algorithm is a simple and intuitive classification algorithm. It works by storing all available cases and classifying new cases based on a similarity measure (e.g., distance functions). When a new case is to be classified, the algorithm finds the k nearest neighbors in the training dataset and assigns the most common class among those neighbors to the new case.

In the context of the Iris dataset, the k-NN algorithm can be used to classify iris flowers into one of the three species based on their sepal and petal measurements. The algorithm calculates the distances between the new flower and all other flowers in the dataset, then selects the k nearest neighbors and assigns the class label based on the majority class among those neighbors.

The performance of the k-NN algorithm can be evaluated using various metrics such as accuracy, precision, recall, and F1-score, which provide insights into how well the algorithm is classifying the flowers. These metrics help us understand the strengths and weaknesses of the algorithm and can be used to tune its parameters for better performance.



Code:

|  |
| --- |
| from sklearn.datasets import load\_iris  from sklearn.model\_selection import train\_test\_split  from sklearn.neighbors import KNeighborsClassifier  from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  import matplotlib.pyplot as plt  import numpy as np  from tabulate import tabulate  # Load the Iris dataset  iris = load\_iris()  X = iris.data  y = iris.target  target\_names = iris.target\_names  # Split the dataset into training and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  # Initialize the k-NN classifier  knn = KNeighborsClassifier(n\_neighbors=3)  # Train the classifier  knn.fit(X\_train, y\_train)  # Make predictions on the test set  y\_pred = knn.predict(X\_test)  # Calculate performance metrics  accuracy = accuracy\_score(y\_test, y\_pred)  precision = precision\_score(y\_test, y\_pred, average='weighted')  recall = recall\_score(y\_test, y\_pred, average='weighted')  f1 = f1\_score(y\_test, y\_pred, average='weighted')  # Print the performance metrics  print(f"Accuracy: {accuracy}")  print(f"Precision: {precision}")  print(f"Recall: {recall}")  print(f"F1-score: {f1}")  # Plot the performance metrics  labels = ['Accuracy', 'Precision', 'Recall', 'F1-score']  values = [accuracy, precision, recall, f1]  plt.figure(figsize=(10, 5))  plt.bar(labels, values, color=['blue', 'green', 'orange', 'red'])  plt.ylabel('Score')  plt.title('Performance Metrics of k-NN on Iris Dataset')  plt.ylim(0, 1)  plt.show()  # Create a table for actual and predicted flower names  table\_data = []  for i in range(len(y\_test)):  table\_data.append([target\_names[y\_test[i]], target\_names[y\_pred[i]]])  print("\nActual vs Predicted:")  print(tabulate(table\_data, headers=['Actual', 'Predicted'], tablefmt='grid')) |

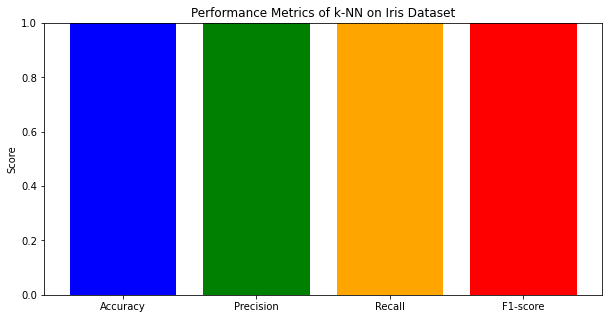
Output:

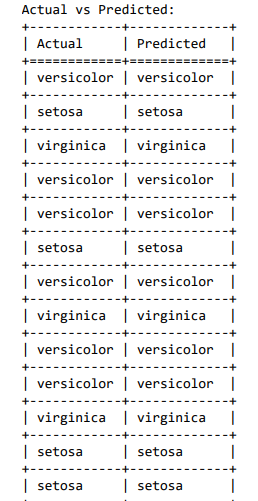
Accuracy: 1.0

Precision: 1.0

Recall: 1.0

F1-score: 1.0





**10. Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select the appropriate dataset for your experiment and draw gr**

**Answer:**

To implement the Locally Weighted Regression (LWR) algorithm, we'll use Python. We'll also select a dataset and fit data points using LWR. For this example, let's use the boston dataset from the sklearn—datasets module, which contains housing prices and other information about houses in Boston suburbs. We'll fit a regression line using LWR and plot the results.

|  |
| --- |
| **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-") |

Output:

