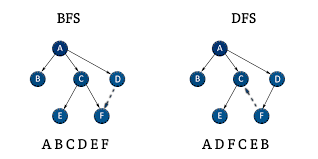
**PRACTICAL - 01**

**AIM**: Implement Breadth first search algorithm for Romanian map problem.

**THEORY**:

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining node A and all of its neighbors. In the next step, the neighbors of the nearest node of A are explored and the process continues in the further steps. The algorithm explores all neighbors of all the nodes and ensures that each node is visited exactly once, and no node is visited twice.



**Code:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue: # Creating loop to visit each node

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

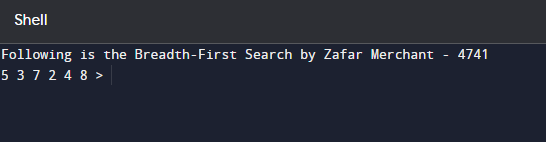
queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search by Zafar Merchant - 4741")

bfs(visited, graph, '5') # function calling

**Output:**

****

**Conclusion:**

Implementing the Breadth-First Search algorithm provides a robust and efficient method for traversing or searching through graphs. BFS guarantees finding the shortest path in an unweighted graph from a source node to all other reachable nodes

**PRACTICAL - 02**

**Aim:** Implement Iterative deep depth first search for Romanian map problem.

**Theory:**

Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. It essentially explores as far as possible along each branch of the structure before backtracking. Imagine you're navigating a maze, DFS would be like taking a single turn at every junction until you hit a dead end, then backtracking and trying a different turn.

Here's a breakdown of how DFS works:

1. Start at a Root Node: You begin at a designated node, typically the root node in a tree or any node in a graph.
2. Explore Unvisited Neighbors: Visit all the unvisited nodes connected to the current node.
3. Mark Visited and Go Deeper: Mark the current node as visited to avoid revisiting it and move to one of its unvisited neighbors. This becomes the new current node.
4. Repeat Until Stack Empty: Keep repeating steps 2 and 3 until there are no more unvisited neighbors of the current node. If you reach a dead end, backtrack to the previous node and continue exploring unvisited neighbors there.

**Code:**

dict\_hn = {'Arad': 336, 'Bucharest': 0, 'Craiova': 160, 'Drobeta': 242,

'Eforie': 161, 'Fagaras': 176, 'Giurgiu': 77, 'Hirsova': 151,

'Iasi': 226, 'Lugoj': 244, 'Mehadia': 241, 'Neamt': 234,

'Oradea': 380, 'Pitesti': 100, 'Rimnicu': 193, 'Sibiu': 253,

'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374}

dict\_gn = {'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

'Bucharest': {'Urziceni': 85, 'Giurgiu': 90, 'Pitesti': 101, 'Fagaras': 211},

'Craiova': {'Drobeta': 120, 'Pitesti': 138, 'Rimnicu': 146},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Eforie': {'Hirsova': 86},

'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Giurgiu': {'Bucharest': 90},

'Hirsova': {'Eforie': 86, 'Urziceni': 98},

'Iasi': {'Neamt': 87, 'Vaslui': 92},

'Lugoj': {'Mehadia': 70, 'Timisoara': 111},

'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

'Neamt': {'Iasi': 87},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Pitesti': {'Rimnicu': 97, 'Bucharest': 101, 'Craiova': 138},

'Rimnicu': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

'Sibiu': {'Rimnicu': 80, 'Fagaras': 99, 'Arad': 140, 'Oradea': 151},

'Timisoara': {'Lugoj': 111, 'Arad': 118},

'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},

'Vaslui': {'Iasi': 92, 'Urziceni': 142},

'Zerind': {'Oradea': 71, 'Arad': 75}}

import queue as Q

def DLS(city, visitedstack, startlimit, endlimit):

global result

found=0

result=result+city+' '

visitedstack.append(city)

if city==goal:

return 1

if startlimit==endlimit:

return 0

for eachcity in dict\_gn[city].keys():

if eachcity not in visitedstack:

found=DLS(eachcity, visitedstack, startlimit+1, endlimit)

if found:

return found

def IDDFS(city, visitedstack, endlimit):

global result

for i in range(0, endlimit):

print("Searching at Limit: ",i)

found=DLS(city, visitedstack, 0, i)

if found:

print("Found")

break

else:

print("Not Found! ")

print(result)

print(" ---- ")

result=' '

visitedstack=[]

start = 'Arad'

goal = 'Bucharest'

result = ''

visitedstack=[]

IDDFS(start, visitedstack, 9)

print("IDDFS Traversal from ",start," to ", goal," is: ")

print(result)

**Output:**

****

**CONCLUSION:**

Hence, we have successfully implemented Iterative deep depth first search

algorithm.

**PRACTICAL -03**

**AIM:** Implement A\* search algorithm for Romanian map problem.

**THEORY**

The A\* search algorithm is an informed search algorithm used for finding the shortest path from a start node to a goal node in a weighted graph. It is widely known for its efficiency and optimality, making it a popular choice for pathfinding and route planning in various applications, including artificial intelligence, robotics, and games.

1. **Nodes and Edges:** A\* operates on a graph composed of nodes (vertices) connected by edges (links). Each edge has a non-negative weight representing the cost or distance between nodes.
2. **Heuristic Function (h):** A\* utilizes a heuristic function, denoted as 'h(n)', which estimates the cost from a given node 'n' to the goal node. The heuristic function provides guidance to the algorithm by helping it prioritize nodes likely to lead to the goal more efficiently.
3. **Cost Functions:**

g(n): Represents the actual cost from the start node to node 'n'. It tracks the cumulative cost incurred along the path.

f(n): The total estimated cost of the cheapest path from the start node to the goal node passing through node 'n'. It is calculated as the sum of 'g(n)' and 'h(n)'.

1. **Open and Closed Lists:**

Open List: Contains nodes that have been discovered but not yet evaluated. Nodes in the open list are candidates for expansion.

Closed List: Contains nodes that have already been evaluated. Once a node is evaluated, it is moved from the open list to the closed list.

**Code:**

import queue as Q

dict\_hn = {'Arad': 336, 'Bucharest': 0, 'Craiova': 160, 'Drobeta': 242,

'Eforie': 161, 'Fagaras': 176, 'Giurgiu': 77, 'Hirsova': 151,

'Iasi': 226, 'Lugoj': 244, 'Mehadia': 241, 'Neamt': 234,

'Oradea': 380, 'Pitesti': 100, 'Rimnicu': 193, 'Sibiu': 253,

'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374}

dict\_gn = {'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

'Bucharest': {'Urziceni': 85, 'Giurgiu': 90, 'Pitesti': 101, 'Fagaras': 211},

'Craiova': {'Drobeta': 120, 'Pitesti': 138, 'Rimnicu': 146},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Eforie': {'Hirsova': 86},

'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Giurgiu': {'Bucharest': 90},

'Hirsova': {'Eforie': 86, 'Urziceni': 98},

'Iasi': {'Neamt': 87, 'Vaslui': 92},

'Lugoj': {'Mehadia': 70, 'Timisoara': 111},

'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

'Neamt': {'Iasi': 87},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Pitesti': {'Rimnicu': 97, 'Bucharest': 101, 'Craiova': 138},

'Rimnicu': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

'Sibiu': {'Rimnicu': 80, 'Fagaras': 99, 'Arad': 140, 'Oradea': 151},

'Timisoara': {'Lugoj': 111, 'Arad': 118},

'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},

'Vaslui': {'Iasi': 92, 'Urziceni': 142},

'Zerind': {'Oradea': 71, 'Arad': 75}}

start = 'Arad'

goal = 'Bucharest'

result = ''

def get\_fn(citystr):

cities = citystr.split(" , ")

hn = gn = 0

for ctr in range(0, len(cities)-1):

gn = gn+dict\_gn[cities[ctr]][cities[ctr+1]]

hn = dict\_hn[cities[len(cities)-1]]

return (hn+gn)

def expand(cityq):

global result

tot, citystr, thiscity = cityq.get()

if thiscity == goal:

result = citystr+" : : "+str(tot)

return

for cty in dict\_gn[thiscity]:

cityq.put((get\_fn(citystr+" , "+cty), citystr+" , "+cty, cty))

expand(cityq)

def main():

cityq = Q.PriorityQueue()

thiscity = start

cityq.put((get\_fn(start), start, thiscity))

expand(cityq)

print("Zafar Merchant - 4741")

print("The A\* path with the total is: ")

print(result)

main()

**Output:**

**A screenshot of a computer

Description automatically generated**

**Conclusion:**

Hence, we successfully implemented A\* star search algorithm on Romanian map.

# **Practical No: 4**

**Aim:** Implement recursive best-first search algorithm for Romanian map problem

**Theory:**

Best-First Search for Pathfinding in the Romanian Map Problem

The Romanian map problem is a well-established benchmark in heuristic search algorithms. It involves finding the shortest path between cities in Romania, often with Arad as the starting point and Bucharest as the goal. Best-first search is a viable approach to solve this problem, but it's crucial to acknowledge its limitations compared to the more powerful A\* search algorithm

**Code:**

dict\_hn={'Arad':336,'Bucharest':0,'Craiova':160,'Drobeta':242,'Eforie':161,

'Fagaras':176,'Giurgiu':77,'Hirsova':151,'Iasi':226,'Lugoj':244,

'Mehadia':241,'Neamt':234,'Oradea':380,'Pitesti':100,'Rimnicu':193,

'Sibiu':253,'Timisoara':329,'Urziceni':80,'Vaslui':199,'Zerind':374}

dict\_gn=dict(

Arad=dict(Zerind=75,Timisoara=118,Sibiu=140), Bucharest=dict(Urziceni=85,Giurgiu=90,Pitesti=101,Fagaras=211), Craiova=dict(Drobeta=120,Pitesti=138,Rimnicu=146), Drobeta=dict(Mehadia=75,Craiova=120), Eforie=dict(Hirsova=86), Fagaras=dict(Sibiu=99,Bucharest=211), Giurgiu=dict(Bucharest=90), Hirsova=dict(Eforie=86,Urziceni=98), Iasi=dict(Neamt=87,Vaslui=92), Lugoj=dict(Mehadia=70,Timisoara=111), Mehadia=dict(Lugoj=70,Drobeta=75), Neamt=dict(Iasi=87), Oradea=dict(Zerind=71,Sibiu=151), Pitesti=dict(Rimnicu=97,Bucharest=101,Craiova=138), Rimnicu=dict(Sibiu=80,Pitesti=97,Craiova=146), Sibiu=dict(Rimnicu=80,Fagaras=99,Arad=140,Oradea=151), Timisoara=dict(Lugoj=111,Arad=118), Urziceni=dict(Bucharest=85,Hirsova=98,Vaslui=142), Vaslui=dict(Iasi=92,Urziceni=142), Zerind=dict(Oradea=71,Arad=75)

)

import queue as Q

start='Arad' goal='Bucharest' result='' def get\_fn(citystr):

cities=citystr.split(',')

hn=gn=0

for ctr in range(0,len(cities)-1):

gn=gn+dict\_gn[cities[ctr]][cities[ctr+1]]

hn=dict\_hn[cities[len(cities)-1]]

return(hn+gn)

def printout(cityq):

for i in range(0,cityq.qsize()):

print(cityq.queue[i])

def expand(cityq):

global result

tot,citystr,thiscity=cityq.get()

nexttot=999

if not cityq.empty():

22

nexttot,nextcitystr,nextthiscity=cityq.queue[0]

if thiscity==goal and tot<nexttot:

result=citystr+'::'+str(tot)

return

print("Expanded city------------------------------",thiscity)

print("Second best f(n)------------------------------",nexttot)

tempq=Q.PriorityQueue()

for cty in dict\_gn[thiscity]:

tempq.put((get\_fn(citystr+','+cty),citystr+','+cty,cty))

for ctr in range(1,3):

ctrtot,ctrcitystr,ctrthiscity=tempq.get()

if ctrtot<nexttot:

cityq.put((ctrtot,ctrcitystr,ctrthiscity))

else:

cityq.put((ctrtot,citystr,thiscity))

break

printout(cityq)

expand(cityq)

def main():

cityq=Q.PriorityQueue()

thiscity=start

cityq.put((999,"NA","NA"))

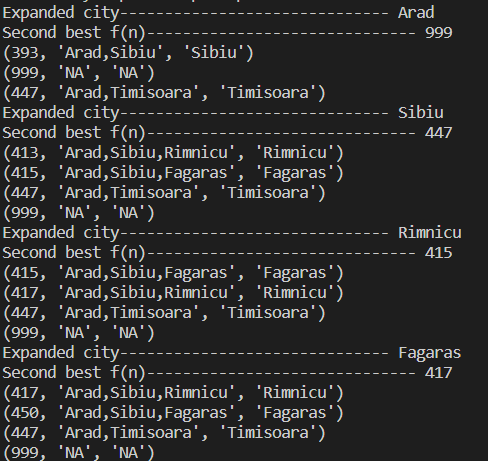
cityq.put((get\_fn(start),start,thiscity))

expand(cityq)

print(result)

main()

**Output:**

****

A computer screen shot of white text

Description automatically generated

**Conclusion:**

Successfully implemented recursive best-first search algorithm for Romanian map problem

# **Practical No: 5**

**Aim:** Write a program to implement the general structure and working of the Genetic Algorithm.

**Theory:**

Genetic Algorithm (GA) is a heuristic search algorithm inspired by the process of natural selection and genetics. It is commonly used to find optimal or near-optimal solutions to optimization and search problems. Here's how it generally works:

Initialization: Start with a population of candidate solutions (also known as chromosomes or individuals). These solutions are typically represented as binary strings, but they can also be represented in other formats depending on the problem.

Evaluation: Each individual in the population is evaluated using a fitness function, which quantifies how good or bad the solution is with respect to the optimization problem. The fitness function guides the search towards better solutions.

Selection: Select individuals from the current population to create a mating pool for the next generation. Selection is typically done using a probabilistic method where individuals with higher fitness have a greater chance of being selected. Common selection methods include roulette wheel selection, tournament selection, and rank-based selection.

**Code:**

import numpy

# Parameter initialization

genes = 2

chromosomes = 10

mattingPoolSize = 6

offspringSize = chromosomes - mattingPoolSize

lb = -5

ub = 5

populationSize = (chromosomes, genes)

generations = 3

# Population initialization

population = numpy.random.uniform(lb, ub, populationSize)

for generation in range(generations):

print(("Generation:", generation+1))

fitness = numpy.sum(population\*population, axis=1)

print("\npopulation")

print(population)

print("\nfitness calcuation")

print(fitness)

# Following statement will create an empty two dimensional array to store parents

parents = numpy.empty((mattingPoolSize, population.shape[1]))

# A loop to extract one parent in each iteration

for p in range(mattingPoolSize):

# Finding index of fittest chromosome in the population

fittestIndex = numpy.where(fitness == numpy.max(fitness))

# Extracting index of fittest chromosome

fittestIndex = fittestIndex[0][0]

# Copying fittest chromosome into parents array

parents[p, :] = population[fittestIndex, :]

# Changing fitness of fittest chromosome to avoid reselection of that chromosome

fitness[fittestIndex] = -1

print("\nParents:")

print(parents)

# Following statement will create an empty two dimensional array to store offspring

offspring = numpy.empty((offspringSize, population.shape[1]))

for k in range(offspringSize):

# Determining the crossover point

crossoverPoint = numpy.random.randint(0, genes)

# Index of the first parent.

parent1Index = k % parents.shape[0]

# Index of the second.

parent2Index = (k+1) % parents.shape[0]

# Extracting first half of the offspring

offspring[k, 0: crossoverPoint] = parents[parent1Index, 0: crossoverPoint]

# Extracting second half of the offspring

offspring[k, crossoverPoint:] = parents[parent2Index, crossoverPoint:]

print("\nOffspring after crossover:")

print(offspring)

# Implementation of random initialization mutation.

for index in range(offspring.shape[0]):

randomIndex = numpy.random.randint(1, genes)

randomValue = numpy.random.uniform(lb, ub, 1)

offspring[index, randomIndex] = offspring[index, randomIndex] + randomValue

print("\n Offspring after Mutation")

print(offspring)

population[0:parents.shape[0], :] = parents

population[parents.shape[0]:, :] = offspring

print("\nNew Population for next generation:")

print(population)

fitness = numpy.sum(population\*population, axis=1)

fittestIndex = numpy.where(fitness == numpy.max(fitness))

# Extracting index of fittest chromosome

fittestIndex = fittestIndex[0][0]

# Getting Best chromosome

fittestInd = population[fittestIndex, :]

bestFitness = fitness[fittestIndex]

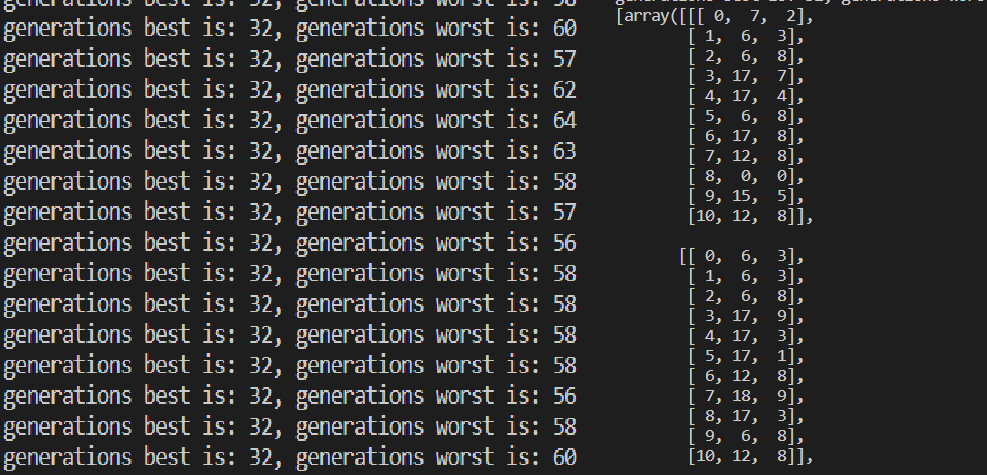
print("\nBest Individual:")

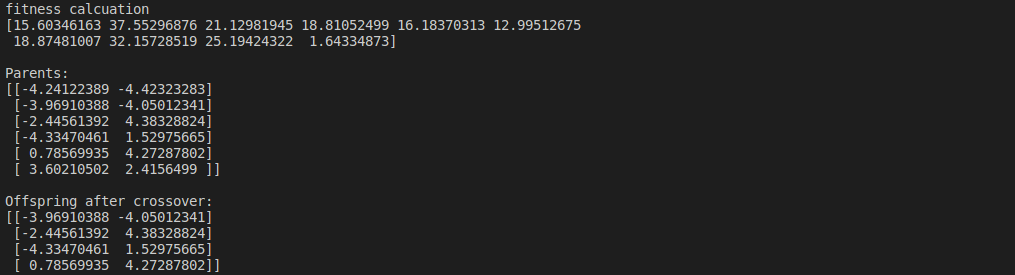
print(fittestInd)

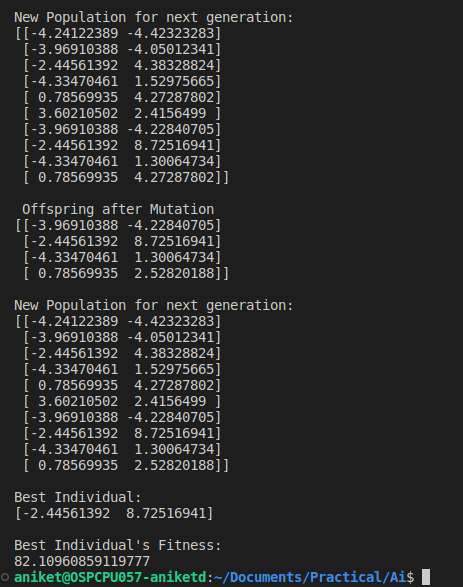
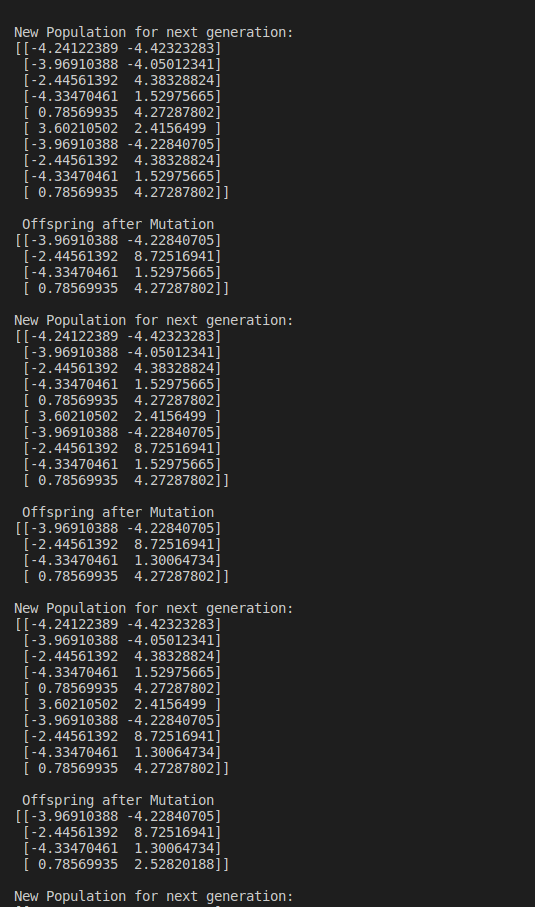
print("\nBest Individual's Fitness:")

print(bestFitness)

**Output:**



****



# 

**Conclusion**

Successfully implemented recursive best-first search algorithm for Romanian map problem

# **Practical No: 6**

**Aim:** Implement the Perceptron Algorithm

**Theory:**

The Perceptron Algorithm is a simple binary classification algorithm used for supervised learning tasks. It's based on the concept of a single-layer neural network, called a perceptron, which learns to classify input data into two categories based on a linear decision boundary.

Here's a brief theory on how the Perceptron Algorithm works:

⦁ Initialization: Initialize the weights and bias parameters of the perceptron randomly or with predefined values.

⦁ Input Processing: For each input sample, calculate the weighted sum of its features along with the bias term. This is essentially a linear combination of the input features.

⦁ Activation: Apply an activation function to the weighted sum. Traditionally, the step function is used, which returns 1 if the weighted sum is greater than or equal to a threshold, and 0 otherwise.

⦁ Prediction: Use the activation result to classify the input sample into one of the two categories.

⦁ Error Calculation: Compare the predicted output with the actual target label to calculate the prediction error.

⦁ Weight Update: Adjust the weights and bias parameters based on the prediction error, using a learning rate to control the magnitude of the update. The weights are updated in the direction that minimizes the error.

⦁ Iteration: Repeat steps 2 to 6 for multiple iterations (epochs) or until convergence criteria are met. During each iteration, the perceptron learns to better classify the input samples.

⦁ Convergence: The perceptron algorithm converges when the prediction error becomes sufficiently small, or when a maximum number of iterations is reached.

⦁ Classification: Once trained, the perceptron can be used to classify new, unseen input samples into the appropriate categories based on the learned decision boundary.

**Code:**

import numpy as np

class Perceptron:

def \_\_init\_\_(self, learning\_rate, epochs):

self.weights = None

self.bias = None

self.learning\_rate = learning\_rate

self.epochs = epochs

def activation(self, z):

return np.heaviside(z, 0)

def fit(self, X, y):

n\_features = X.shape[1]

self.weights = np.zeros((n\_features))

self.bias = 0

for epoch in range(self.epochs):

for i in range(len(X)):

z = np.dot(X, self.weights) + self.bias

y\_pred = self.activation(z)

self.weights = self.weights + self.learning\_rate \* (y[i] - y\_pred[i]) \* X[i]

self.bias = self.bias + self.learning\_rate \* (y[i] - y\_pred[i])

return self.weights, self.bias

def predict(self, X):

z = np.dot(X, self.weights) + self.bias

return self.activation(z)

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

import numpy as np

iris = load\_iris()

X = iris.data[:, (0, 1)]

y = (iris.target == 0).astype(int)

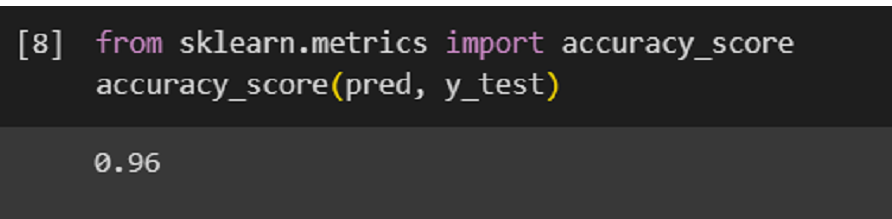
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.5, random\_state=42)

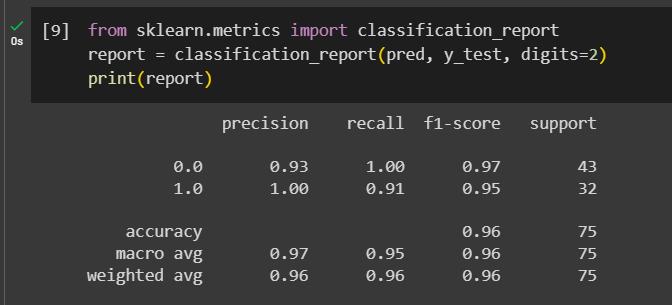
perceptron = Perceptron(0.001, 100)

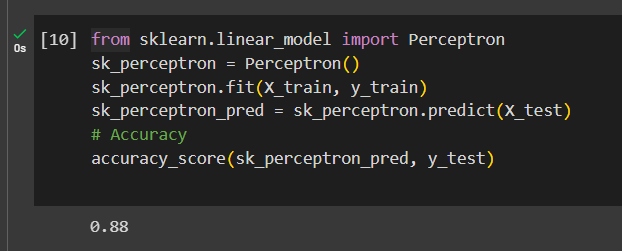
perceptron.fit(X\_train, y\_train)

pred = perceptron.predict(X\_test)

**Output:**

****

****

****

**Conclusion:**

Successfully implemented the Perceptron Algorithm

# **Practical No: 7**

**Aim:** : Implement Fuzzy Inference System

**Theory:**   
A Fuzzy Inference System (FIS) is a computational model that utilizes fuzzy logic to represent and process uncertain or imprecise information. Here's a brief overview of the theory behind implementing a Fuzzy Inference System:

1. **Fuzzy Logic**: Fuzzy logic is an extension of classical logic that deals with uncertainty by allowing values to be partially true or partially false. In fuzzy logic, variables can have degrees of membership to different fuzzy sets, represented by membership functions.
2. **Fuzzy Sets**: Fuzzy sets are defined by membership functions that assign degrees of membership to elements in a universe of discourse. These membership functions can take various shapes, such as triangular, trapezoidal, or Gaussian, and represent the degree of truth of a variable's value being a member of the set.
3. **Fuzzy Inference Process**: The fuzzy inference process involves several steps:
   * Fuzzification: Convert crisp input values into fuzzy sets using membership functions.
   * Rule Evaluation: Apply fuzzy IF-THEN rules to determine the degree of activation of each rule.
   * Aggregation: Combine the activated rules to generate a fuzzy output.
   * Defuzzification: Convert the fuzzy output into a crisp value using a defuzzification method, such as centroid or weighted average.
4. **Membership Functions**: The choice of membership functions plays a crucial role in the effectiveness of a Fuzzy Inference System. They should be carefully designed to capture the relationships between input and output variables accurately.
5. **Defuzzification Methods**: Defuzzification methods are used to convert the fuzzy output into a crisp value. Common defuzzification methods include centroid, mean of maximum, and weighted average.
6. **Applications**: Fuzzy Inference Systems are widely used in various fields, including control systems, pattern recognition, decision-making, and data analysis. They excel in domains where traditional methods struggle to handle uncertainty and imprecision.

**Code:**

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership

# functions

quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')

service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')

tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)

quality.automf(3)

service.automf(3)

# Custom membership functions can be built interactively with a familiar,

# Pythonic API

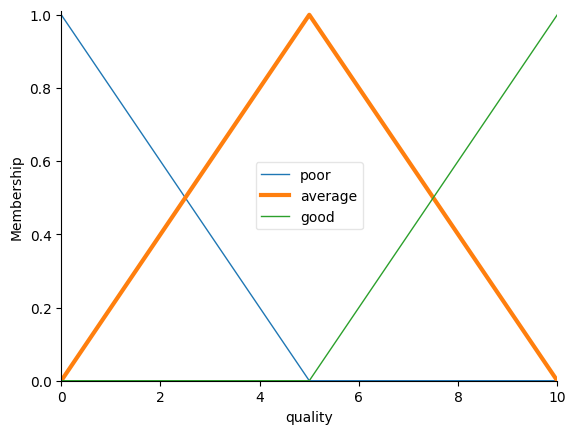
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 10])

tip['medium'] = fuzz.trimf(tip.universe, [0, 10, 20])

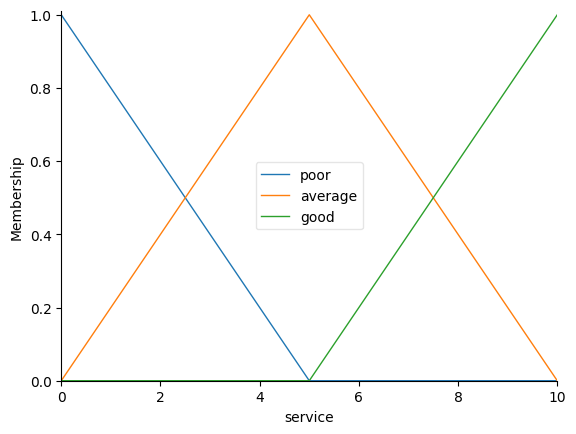
tip['high'] = fuzz.trimf(tip.universe, [10, 20, 25])

# You can see how these look with .view()

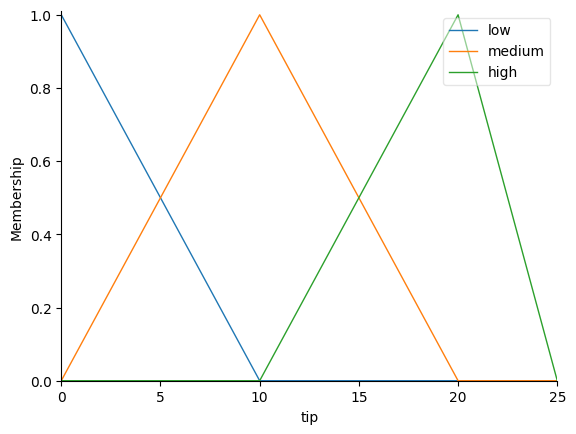
quality['average'].view()



service.view()



tip.view()

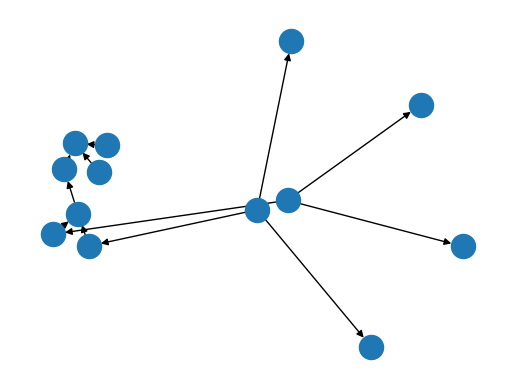


rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])

rule2 = ctrl.Rule(service['average'], tip['medium'])

rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()



tipping\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

tipping = ctrl.ControlSystemSimulation(tipping\_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API

# Note: if you like passing many inputs all at once, use .inputs(dict\_of\_data)

tipping.input['quality'] = 7.2

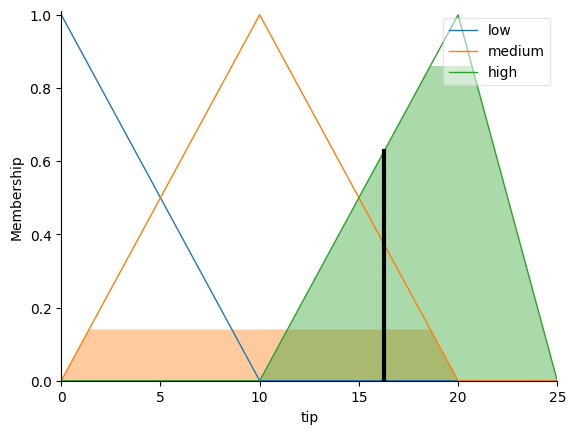
tipping.input['service'] = 9.3

# Crunch the numbers

tipping.compute()

print(tipping.output['tip'])

tip.view(sim=tipping)



**Conclusion:**

Successfully implemented Fuzzy Inference System.

# **Practical No: 8**

**Aim:** Solve Fuzzy Control Systems: The Tipping Problem

**Theory:**   
Fuzzy Control Systems are a type of control system where the mathematical principles of fuzzy logic are applied to regulate inputs and outputs of a system. The Tipping Problem is a classic example used to demonstrate the concept of fuzzy control systems.

In the Tipping Problem, the objective is to determine the appropriate tip amount in a restaurant based on two input variables: the quality of the food and the quality of the service. These input variables are represented as linguistic terms (e.g., poor, average, good) and are fuzzified to quantify their degree of membership in each category.

**Code:**

import numpy as np

import skfuzzy as fuzz

import matplotlib.pyplot as plt

x\_food\_qual = np.arange(0, 11, 1)

x\_service\_qual = np.arange(0, 11, 1)

x\_tip  = np.arange(0, 26, 1)

# Visualize these universes and membership functions

fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(8, 9))

ax0.plot(x\_food\_qual, qual\_lo, 'b', linewidth=1.5, label='Poor')

ax0.plot(x\_food\_qual, qual\_md, 'g', linewidth=1.5, label='Average')

ax0.plot(x\_food\_qual, qual\_hi, 'r', linewidth=1.5, label='Excellent')

ax0.set\_title('Food Quality')

ax0.legend()

ax1.plot(x\_service\_qual, serv\_lo, 'b', linewidth=1.5, label='Poor')

ax1.plot(x\_service\_qual, serv\_md, 'g', linewidth=1.5, label='Average')

ax1.plot(x\_service\_qual, serv\_hi, 'r', linewidth=1.5, label='Excellent')

ax1.set\_title('Service Quality')

ax1.legend()

ax2.plot(x\_tip, tip\_lo, 'b', linewidth=1.5, label='Low')

ax2.plot(x\_tip, tip\_md, 'g', linewidth=1.5, label='Medium')

ax2.plot(x\_tip, tip\_hi, 'r', linewidth=1.5, label='High')

ax2.set\_title('Tip Amount')

ax2.legend()

# Turn off top/right axes

for ax in (ax0, ax1, ax2):

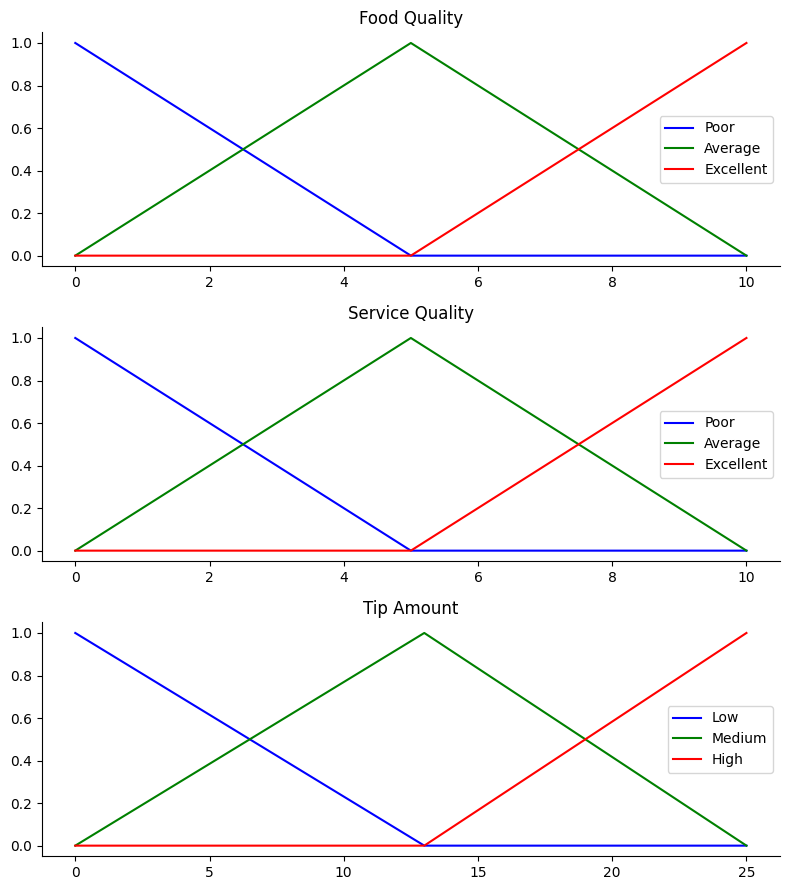
    ax.spines['top'].set\_visible(False)

    ax.spines['right'].set\_visible(False)

    ax.get\_xaxis().tick\_bottom()

    ax.get\_yaxis().tick\_left()

plt.tight\_layout()



# We need the activation of our fuzzy membership functions at these values.

# The exact values 6.5 and 9.8 do not exist on our universes...

# This is what fuzz.interp\_membership exists for!

food\_qual\_level\_lo = fuzz.interp\_membership(x\_food\_qual, qual\_lo, 6.5)

food\_qual\_level\_md = fuzz.interp\_membership(x\_food\_qual, qual\_md, 6.5)

food\_qual\_level\_hi = fuzz.interp\_membership(x\_food\_qual, qual\_hi, 6.5)

service\_qual\_level\_lo = fuzz.interp\_membership(x\_service\_qual, serv\_lo, 9.8)

service\_qual\_level\_md = fuzz.interp\_membership(x\_service\_qual, serv\_md, 9.8)

service\_qual\_level\_hi = fuzz.interp\_membership(x\_service\_qual, serv\_hi, 9.8)

# Now we take our rules and apply them. Rule 1 concerns poor food OR service.

# The OR operator means we take the maximum of these two.

active\_rule1 = np.fmax(food\_qual\_level\_lo, service\_qual\_level\_lo)

# Now we apply this by clipping the top off the corresponding output

# membership function with `np.fmin`

tip\_activation\_lo = np.fmin(active\_rule1, tip\_lo)  # removed entirely to 0

# For rule 2 we connect acceptable service to medium tipping

tip\_activation\_md = np.fmin(service\_qual\_level\_md, tip\_md)

# For rule 3 we connect high service OR high food with high tipping

active\_rule3 = np.fmax(food\_qual\_level\_hi, service\_qual\_level\_hi)

tip\_activation\_hi = np.fmin(active\_rule3, tip\_hi)

tip0 = np.zeros\_like(x\_tip)

# Visualize this

fig, ax0 = plt.subplots(figsize=(8, 3))

ax0.fill\_between(x\_tip, tip0, tip\_activation\_lo, facecolor='b', alpha=0.7)

ax0.plot(x\_tip, tip\_lo, 'b', linewidth=0.5, linestyle='--', )

ax0.fill\_between(x\_tip, tip0, tip\_activation\_md, facecolor='g', alpha=0.7)

ax0.plot(x\_tip, tip\_md, 'g', linewidth=0.5, linestyle='--')

ax0.fill\_between(x\_tip, tip0, tip\_activation\_hi, facecolor='r', alpha=0.7)

ax0.plot(x\_tip, tip\_hi, 'r', linewidth=0.5, linestyle='--')

ax0.set\_title('Output membership activity')

# Turn off top/right axes

for ax in (ax0,):

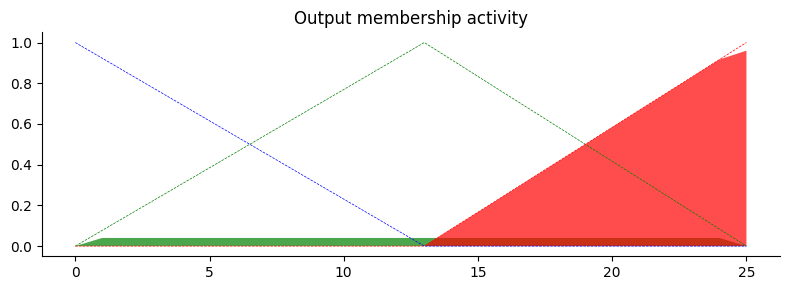
    ax.spines['top'].set\_visible(False)

    ax.spines['right'].set\_visible(False)

    ax.get\_xaxis().tick\_bottom()

    ax.get\_yaxis().tick\_left()

plt.tight\_layout()



# Aggregate all three output membership functions together

aggregated = np.fmax(tip\_activation\_lo,

                     np.fmax(tip\_activation\_md, tip\_activation\_hi))

# Calculate defuzzified result

tip = fuzz.defuzz(x\_tip, aggregated, 'centroid')

tip\_activation = fuzz.interp\_membership(x\_tip, aggregated, tip)  # for plot

# Visualize this

fig, ax0 = plt.subplots(figsize=(8, 3))

ax0.plot(x\_tip, tip\_lo, 'b', linewidth=0.5, linestyle='--', )

ax0.plot(x\_tip, tip\_md, 'g', linewidth=0.5, linestyle='--')

ax0.plot(x\_tip, tip\_hi, 'r', linewidth=0.5, linestyle='--')

ax0.fill\_between(x\_tip, tip0, aggregated, facecolor='Orange', alpha=0.7)

ax0.plot([tip, tip], [0, tip\_activation], 'k', linewidth=1.5, alpha=0.9)

ax0.set\_title('Aggregated membership and result (line)')

# Turn off top/right axes

for ax in (ax0,):

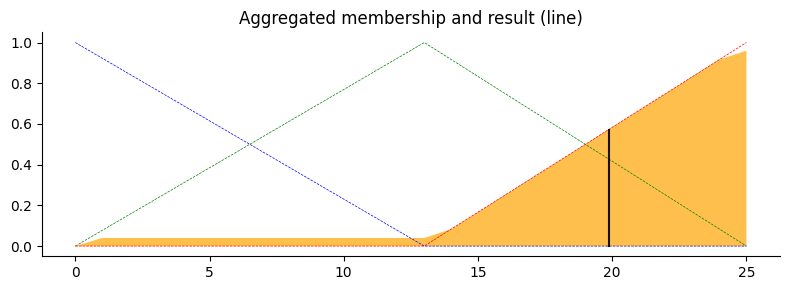
    ax.spines['top'].set\_visible(False)

    ax.spines['right'].set\_visible(False)

    ax.get\_xaxis().tick\_bottom()

    ax.get\_yaxis().tick\_left()

plt.tight\_layout()



**Conclusion:**

Successfully implemented The Tipping Problem in Fuzzy Control Systems

# **Practical No: 9**

**Aim:** Naive Bayes’ learning algorithm.

**Theory:**   
Naive Bayes is a simple, yet powerful probabilistic machine learning algorithm used mainly for classification tasks. It's based on Bayes' theorem with an assumption of independence among predictors, hence the "naive" in its name.

* **Bayes' Theorem**: It calculates the probability of a hypothesis given the data.
* **Naive Assumption**: Assumes independence among predictors, simplifying computations.
* **Algorithm Steps**:
  + Training: Calculate prior probabilities and likelihoods of each class based on training data.
  + Prediction: Use Bayes' theorem to calculate posterior probabilities and predict the class.
* **Types**: Gaussian, Multinomial, and Bernoulli Naive Bayes.

Assume we have a Hypothesis(H) and evidence(E),

According to Bayes theorem, the relationship between the probability of the Hypothesis before getting the evidence represented as P(H) and the probability of the hypothesis after getting

the evidence represented as

**P(H|E) is: P(H|E) = P(E|H)\*P(H)/P(E)**

**Prior probability** = P(H) is the probability before getting the evidence **Posterior probability** = P(H|E) is the probability after getting evidence In general,

**P(class|data) = (P(data|class) \* P(class)) / P(data)**

**Code:**

import math

import random

import pandas as pd

import numpy as np

from sklearn.datasets import load\_diabetes

# Function to encode class labels to numerical values

def encode\_class(mydata):

classes = []

for i in range(len(mydata)):

if mydata[i][-1] not in classes:

classes.append(mydata[i][-1])

for i in range(len(classes)):

for j in range(len(mydata)):

if mydata[j][-1] == classes[i]:

mydata[j][-1] = i

return mydata

# Function to split data into training and testing sets

def splitting(mydata, ratio):

train\_num = int(len(mydata) \* ratio)

train = []

test = list(mydata)

while len(train) < train\_num:

index = random.randrange(len(test))

train.append(test.pop(index))

return train, test

# Function to group data instances under each class

def groupUnderClass(mydata):

data\_dict = {}

for i in range(len(mydata)):

if mydata[i][-1] not in data\_dict:

data\_dict[mydata[i][-1]] = []

data\_dict[mydata[i][-1]].append(mydata[i])

return data\_dict

# Function to calculate mean and standard deviation for a set of numbers

def MeanAndStdDev(numbers):

avg = np.mean(numbers)

stddev = np.std(numbers)

return avg, stddev

# Function to calculate mean and standard deviation for each class

def MeanAndStdDevForClass(mydata):

info = {}

data\_dict = groupUnderClass(mydata)

for classValue, instances in data\_dict.items():

info[classValue] = [MeanAndStdDev(attribute) for attribute in zip(\*instances)]

return info

# Function to calculate Gaussian probability density function

def calculateGaussianProbability(x, mean, stdev):

epsilon = 1e-10

expo = math.exp(-(math.pow(x - mean, 2) / (2 \* math.pow(stdev + epsilon, 2))))

return (1 / (math.sqrt(2 \* math.pi) \* (stdev + epsilon))) \* expo

# Function to calculate probabilities for each class

def calculateClassProbabilities(info, test):

probabilities = {}

for classValue, classSummaries in info.items():

probabilities[classValue] = 1

for i in range(len(classSummaries)):

mean, std\_dev = classSummaries[i]

x = test[i]

probabilities[classValue] \*= calculateGaussianProbability(x, mean, std\_dev)

return probabilities

# Function to predict the class for a single instance

def predict(info, test):

probabilities = calculateClassProbabilities(info, test)

bestLabel = max(probabilities, key=probabilities.get)

return bestLabel

# Function to get predictions for multiple instances

def getPredictions(info, test):

predictions = [predict(info, instance) for instance in test]

return predictions

# Function to calculate accuracy rate

def accuracy\_rate(test, predictions):

correct = sum(1 for i in range(len(test)) if test[i][-1] == predictions[i])

return (correct / float(len(test))) \* 100.0

# Load data using sklearn

diabetes = load\_diabetes()

X, y = diabetes.data, diabetes.target

mydata = np.column\_stack((X, y))

# Encode classes and convert attributes to float

mydata = encode\_class(mydata)

# Split the data into training and testing sets

ratio = 0.7

train\_data, test\_data = splitting(mydata, ratio)

print('Total number of examples:', len(mydata))

print('Training examples:', len(train\_data))

print('Test examples:', len(test\_data))

# Train the model

info = MeanAndStdDevForClass(train\_data)

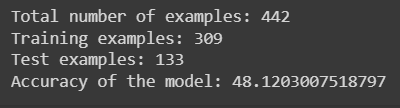
# Test the model

predictions = getPredictions(info, test\_data)

accuracy = accuracy\_rate(test\_data, predictions)

print('Accuracy of the model:', accuracy)

**Output:**

****

**Conclusion:**

Successfully implemented the Naive Bayes’ learning algorithm.