**DAY-2 LAB PROGRAMS**

S.Kangana Sri

192210006

CSA1760-Artificial Intelligence

**13. MIN AND MAX ALGORITHM:**

# Define constants

MAX = 1 # Maximizing player

MIN = -1 # Minimizing player

# Example board representation with 0 = empty, 1 = X, -1 = O

board = [

[0, 1, -1],

[1, 1, 0],

[-1, 0, -1]

]

# Function to evaluate the board state

def evaluate(board):

# Check rows for a win

for row in board:

if row[0] == row[1] == row[2] and row[0] != 0:

return row[0]

# Check columns for a win

for col in range(3):

if board[0][col] == board[1][col] == board[2][col] and board[0][col] != 0:

return board[0][col]

# Check diagonals for a win

if board[0][0] == board[1][1] == board[2][2] and board[0][0] != 0:

return board[0][0]

if board[0][2] == board[1][1] == board[2][0] and board[0][2] != 0:

return board[0][2]

return 0 # No winner (draw or ongoing game)

# Check if there are moves left on the board

def is\_moves\_left(board):

for row in board:

if 0 in row:

return True

return False

# Minimax function

def minimax(board, depth, is\_max):

score = evaluate(board)

# If maximizer (X) has won

if score == MAX:

return score - depth

# If minimizer (O) has won

if score == MIN:

return score + depth

# If no moves left, it's a draw

if not is\_moves\_left(board):

return 0

# Maximizing player's move

if is\_max:

best = -float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == 0:

board[i][j] = MAX

best = max(best, minimax(board, depth + 1, not is\_max))

board[i][j] = 0

return best

# Minimizing player's move

else:

best = float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == 0:

board[i][j] = MIN

best = min(best, minimax(board, depth + 1, not is\_max))

board[i][j] = 0

return best

# Function to find the best move for the maximizing player

def find\_best\_move(board):

best\_val = -float('inf')

best\_move = (-1, -1)

for i in range(3):

for j in range(3):

if board[i][j] == 0:

board[i][j] = MAX

move\_val = minimax(board, 0, False)

board[i][j] = 0

if move\_val > best\_val:

best\_move = (i, j)

best\_val = move\_val

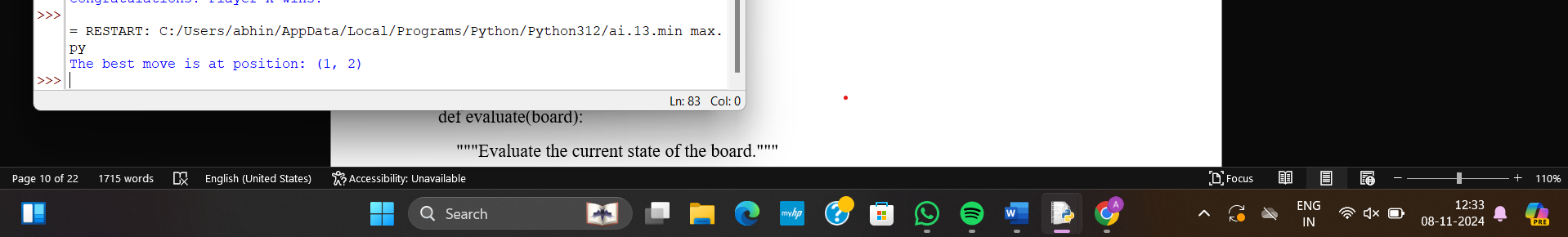
return best\_move

# Find and print the best move

best\_move = find\_best\_move(board)

print(f"The best move is at position: {best\_move}")

**OUTPUT:**



**14. ALPHA BETA PRUNNING:**

import math

def alpha\_beta\_pruning(depth, node\_index, maximizing\_player, values, alpha, beta):

if depth == math.log2(len(values)):

return values[node\_index]

if maximizing\_player:

max\_eval = -math.inf

for i in range(2):

eval = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, False, values, alpha, beta)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = math.inf

for i in range(2):

eval = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, True, values, alpha, beta)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

values = [3, 5, 6, 9, 1, 2, 0, -1]

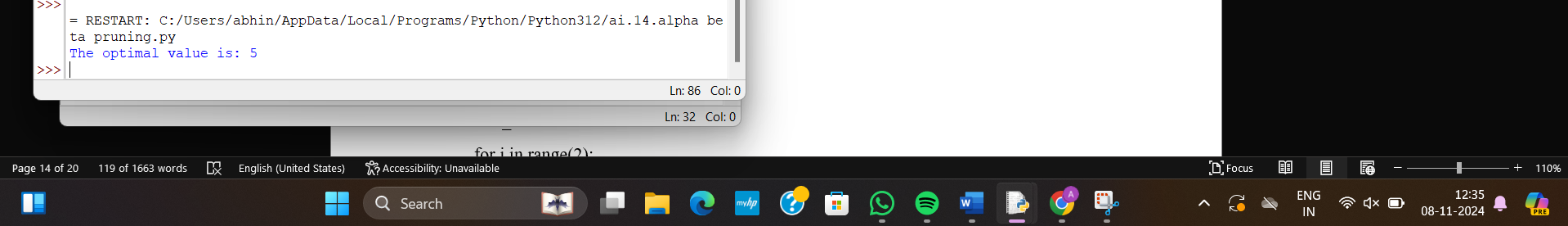
alpha = -math.inf

beta = math.inf

optimal\_value = alpha\_beta\_pruning(0, 0, True, values, alpha, beta)

print("The optimal value is:", optimal\_value)

**OUTPUT:**



**15. DECISION TREE:**

import math

from collections import Counter

def entropy(labels):

total\_count = len(labels)

label\_counts = Counter(labels)

entropy\_value = 0.0

for count in label\_counts.values():

probability = count / total\_count

entropy\_value -= probability \* math.log2(probability)

return entropy\_value

def information\_gain(data, labels, feature\_index):

original\_entropy = entropy(labels)

total\_count = len(labels)

feature\_values = [row[feature\_index] for row in data]

value\_counts = Counter(feature\_values)

weighted\_entropy = 0.0

for value, count in value\_counts.items():

subset\_labels = [labels[i] for i in range(total\_count) if data[i][feature\_index] == value]

weighted\_entropy += (count / total\_count) \* entropy(subset\_labels)

return original\_entropy - weighted\_entropy

def build\_tree(data, labels, features, depth=0):

if len(set(labels)) == 1:

return labels[0]

if len(features) == 0:

return Counter(labels).most\_common(1)[0][0]

gains = [information\_gain(data, labels, i) for i in range(len(features))]

best\_feature\_index = gains.index(max(gains))

tree = {features[best\_feature\_index]: {}}

unique\_values = set(row[best\_feature\_index] for row in data)

for value in unique\_values:

subset\_data = [row[:best\_feature\_index] + row[best\_feature\_index + 1:] for row in data if

row[best\_feature\_index] == value]

subset\_labels = [labels[i] for i in range(len(data)) if data[i][best\_feature\_index] == value]

new\_features = features[:best\_feature\_index] + features[best\_feature\_index + 1:]

subtree = build\_tree(subset\_data, subset\_labels, new\_features, depth + 1)

tree[features[best\_feature\_index]][value] = subtree

return tree

def predict(tree, sample):

if not isinstance(tree, dict):

return tree

feature = next(iter(tree))

feature\_value = sample.get(feature)

if feature\_value in tree[feature]:

return predict(tree[feature][feature\_value], sample)

else:

return None

data = [

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

]

labels = [0, 1, 0, 1, 0]

features = ['Feature 1', 'Feature 2']

decision\_tree = build\_tree(data, labels, features)

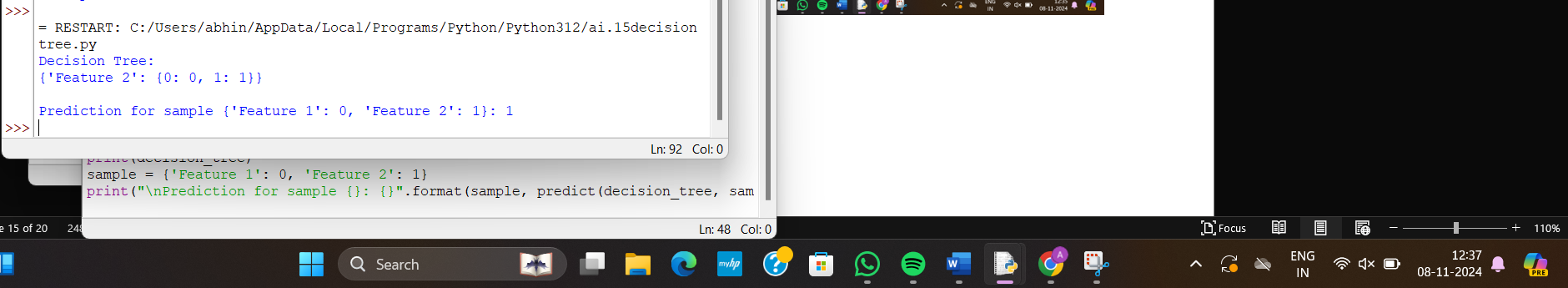
print("Decision Tree:")

print(decision\_tree)

sample = {'Feature 1': 0, 'Feature 2': 1}

print("\nPrediction for sample {}: {}".format(sample, predict(decision\_tree, sample)))

**OUTPUT:**

****

**16. FORWARD NEURAL NETWORK:**

import math

import random

def sigmoid(x):

return 1 / (1 + math.exp(-x))

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.weights\_input\_hidden = []

for \_ in range(self.input\_size):

self.weights\_input\_hidden.append([random.uniform(-1, 1) for \_ in range(self.hidden\_size)])

self.bias\_hidden = [random.uniform(-1, 1) for \_ in range(self.hidden\_size)]

self.weights\_hidden\_output = []

for \_ in range(self.hidden\_size):

self.weights\_hidden\_output.append([random.uniform(-1, 1) for \_ in range(self.output\_size)])

self.bias\_output = [random.uniform(-1, 1) for \_ in range(self.output\_size)]

def feedforward(self, inputs):

hidden\_sum = []

for j in range(self.hidden\_size):

neuron\_sum = 0

for i in range(self.input\_size):

neuron\_sum += inputs[i] \* self.weights\_input\_hidden[i][j]

neuron\_sum += self.bias\_hidden[j]

hidden\_sum.append(neuron\_sum)

hidden\_output = [sigmoid(x) for x in hidden\_sum]

output\_sum = []

for k in range(self.output\_size):

neuron\_sum = 0

for j in range(self.hidden\_size):

neuron\_sum += hidden\_output[j] \* self.weights\_hidden\_output[j][k]

neuron\_sum += self.bias\_output[k]

output\_sum.append(neuron\_sum)

final\_output = [sigmoid(x) for x in output\_sum]

return final\_output

if \_\_name\_\_ == "\_\_main\_\_":

nn = NeuralNetwork(input\_size=2, hidden\_size=3, output\_size=1)

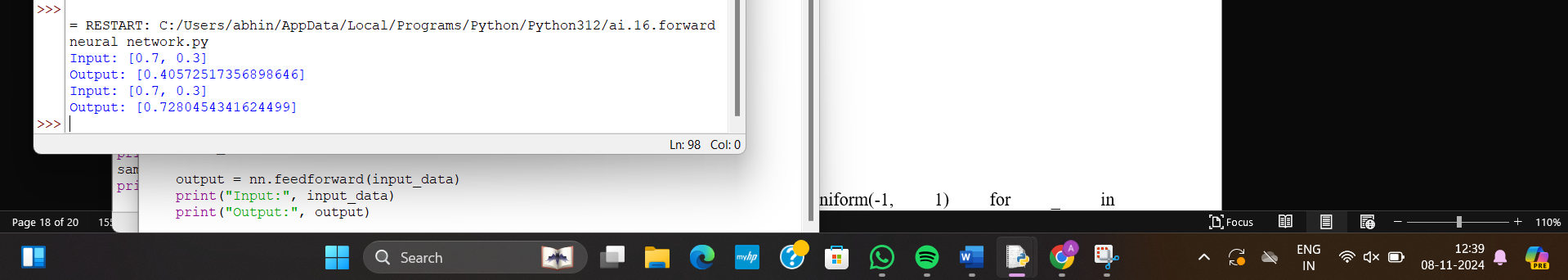
input\_data = [0.7, 0.3]

output = nn.feedforward(input\_data)

print("Input:", input\_data)

print("Output:", output)

**OUTPUT:**

****

**Sum of Integer 1 to n:**

% Base case: The sum of integers from 1 to 0 is 0.

sum\_to\_n(0, 0) :-

write('Base case reached: sum from 1 to 0 is 0.'), nl.

% Recursive case: The sum of integers from 1 to N is N plus the sum from 1 to (N-1).

sum\_to\_n(N, Sum) :-

N > 0, % Ensure N is greater than 0.

N1 is N - 1, % Calculate N-1.

sum\_to\_n(N1, Sum1), % Recursively calculate the sum of integers from 1 to N-1.

Sum is Sum1 + N, % Add N to the sum of integers from 1 to N-1.

write('Summing: 1 + 2 + ... + '), write(N), write(' = '), write(Sum), nl.

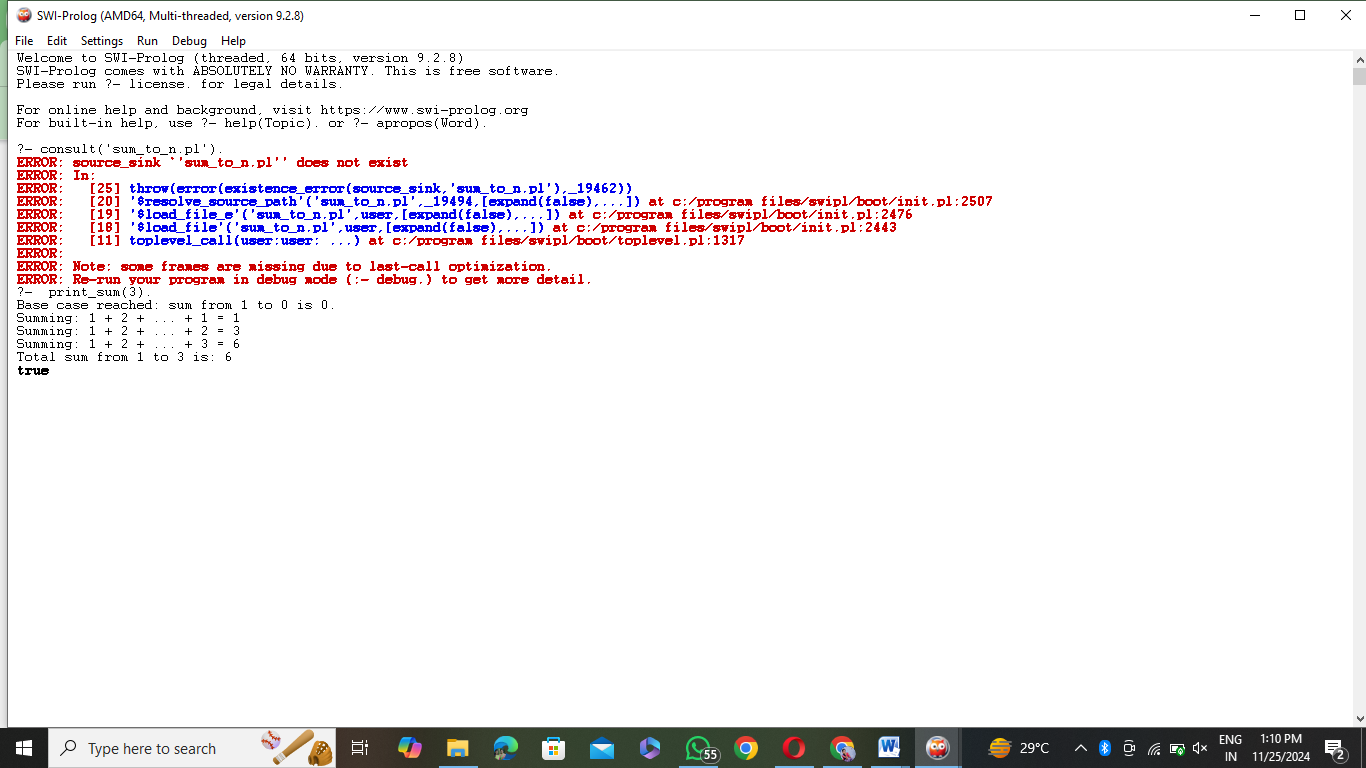
% Main entry point to call the sum and print the result

print\_sum(N) :-

sum\_to\_n(N, Sum),

write('Total sum from 1 to '), write(N), write(' is: '), write(Sum), nl.

**Output:**



**DB With name:**

% Facts: Name, Age, Gender

person('Alice', 25, female).

person('Bob', 30, male).

person('Charlie', 22, male).

person('Diana', 27, female).

% Query a person by name

find\_person(Name, Age, Gender) :-

person(Name, Age, Gender),

write('Name: '), write(Name), nl,

write('Age: '), write(Age), nl,

write('Gender: '), write(Gender), nl.

% Add a new person to the database

add\_person(Name, Age, Gender) :-

assertz(person(Name, Age, Gender)),

write('Person added: '), write(Name), nl.

% Remove a person from the database

delete\_person(Name) :-

retract(person(Name, \_, \_)),

write('Person removed: '), write(Name), nl.

delete\_person(Name) :-

write('No person found with name: '), write(Name), nl.

% List all people in the database

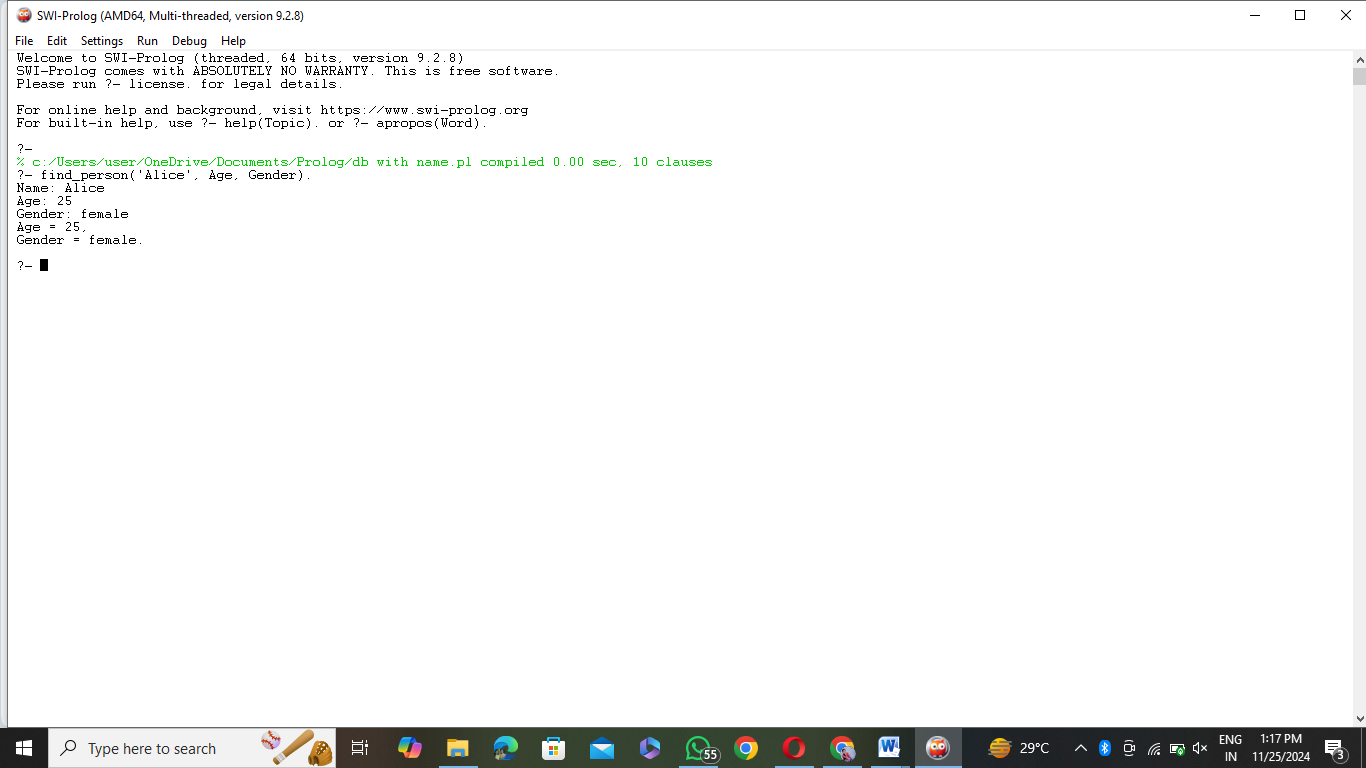
list\_people :-

person(Name, Age, Gender),

write('Name: '), write(Name), write(', Age: '), write(Age), write(', Gender: '), write(Gender), nl,

fail.

list\_people :- write('End of list.'), nl.

**Output:**