AI-LAB

1.Write a program to implement Hill climbing algorithm using python

```
import random
# Distance matrix representing distances between cities
# Replace this with the actual distance matrix for your problem
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
def total_distance(path):

    total = 0
    for i in range(len(path) - 1):
        total += distance_matrix[path[i]][path[i+1]]
    total += distance_matrix[path[-1]][path[0]]
    return total
def hill_climbing_tsp(num_cities, max_iterations=10000):
    current_path = list(range(num_cities))
    current_distance = total_distance(current_path)
    for _ in range(max_iterations):

        neighbor_path = current_path.copy()
        i, j = random.sample(range(num_cities), 2)
        neighbor_path[i], neighbor_path[j] = neighbor_path[j], neighbor_path[i]
        neighbor_distance = total_distance(neighbor_path)


        if neighbor_distance < current_distance:
            current_path = neighbor_path
            current_distance = neighbor_distance
    return current_path

def main():
    num_cities = 4
    solution = hill_climbing_tsp(num_cities)
    print("Optimal path:", solution)
    print("Total distance:", total_distance(solution))
if __name__ == "__main__":
    main()
```

Output :

```
Optimal path: [1, 0, 2, 3]
Total distance: 80
```

2.Write a program to implement Iterative deepening depth-first search python

```python
from collections import defaultdict


class Graph:

    def __init__(self,vertices):
        self.V = vertices
        self.graph = defaultdict(list)


    def addEdge(self,u,v):
        self.graph[u].append(v)

    def DLS(self,src,target,maxDepth):

        if src == target : return True
        if maxDepth <= 0 : return False

        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1)):
                    return True
        return False

    def IDDFS(self,src, target, maxDepth):

        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False
```

```
# Create a graph given in the above diagram
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

if g.IDDFS(src, target, maxDepth) == True:
     print ("Target is reachable from source " +
          "within max depth")
else :
     print ("Target is NOT reachable from source " +
          "within max depth")
```


Output:

```
Target is reachable from source within max depth
```

3.Write a program to implement Depth Limit Search

```
def depth_limit_search(array, depth_limit):

    def dls_helper(arr, current_depth):
        if current_depth > depth_limit:
            return

        for element in arr:
            if isinstance(element, list):
                print(f"At depth {current_depth}: Encountered
nested list, diving deeper...")
```

```
                dls_helper(element, current_depth + 1)
            else:
                print(f"At depth {current_depth}: Processing
element: {element}")

    # Start the depth-limited search with the initial depth of 0
    dls_helper(array, 0)

# Example usage
nested_array = [1, [2, 3], [4, [5, 6]], 7, [8, [9, [10, 11]]]]
depth_limit = 2
depth_limit_search(nested_array, depth_limit)
```

Output:

```
At depth 0: Processing element: 1
At depth 0: Encountered nested list, diving deeper...
At depth 1: Processing element: 2
At depth 1: Processing element: 3
At depth 0: Encountered nested list, diving deeper...
At depth 1: Processing element: 4
At depth 1: Encountered nested list, diving deeper...
At depth 2: Processing element: 5
At depth 2: Processing element: 6
At depth 0: Processing element: 7
At depth 0: Encountered nested list, diving deeper...
At depth 1: Processing element: 8
At depth 1: Encountered nested list, diving deeper...
At depth 2: Processing element: 9
At depth 2: Encountered nested list, diving deeper..
```

4.Write a program to implement Find-S algorithm using python

```
def find_s_algorithm(examples):

    hypothesis = None

    for features, label in examples:
        if label == 1:
            if hypothesis is None:

                hypothesis = features.copy()
```

```
            else:

                for i in range(len(hypothesis)):
                    if hypothesis[i] != features[i]:
                        hypothesis[i] = '?'

    return hypothesis

# Example usage
examples = [
    (['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'], 1),
    (['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same'], 1),
    (['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change'], 0),
    (['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change'], 1)
]

hypothesis = find_s_algorithm(examples)
print("Most specific hypothesis:", hypothesis)
```

Output


```
Most specific hypothesis: ['Sunny', 'Warm', '?',
'Strong', '?', '?']
```

5.Write a program to implement Forward Chaining example

```
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises
        self.conclusion = conclusion

# Knowledge base
rules = [
    Rule(["A"], "B"),
    Rule(["B", "C"], "D"),
    Rule(["D"], "E"),
    Rule(["F"], "C")
```

```
]

def forward_chaining(knowledge_base, query):
    inferred = set()
    agenda = [query]

    while agenda:
        fact = agenda.pop(0)
        if fact not in inferred:
            inferred.add(fact)  # Add fact to inferred set

            # Find rules whose premises are satisfied by
inferred facts
            matching_rules = [rule for rule in knowledge_base
if all(premise in inferred for premise in rule.premises)]


            for rule in matching_rules:
                agenda.append(rule.conclusion)

    return inferred

# Test forward chaining
print("Forward Chaining:")
print("Inferred facts:", forward_chaining(rules, "A"))
```

Output :

```
Forward Chaining:
Inferred facts: {'B', 'A'}
```

6.Write a program to implement Backward Chaining example

```
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises
```

```python
        self.conclusion = conclusion

# Knowledge base
rules = [
    Rule(["A"], "B"),
    Rule(["B", "C"], "D"),
    Rule(["D"], "E"),
    Rule(["F"], "C")
]

def backward_chaining(knowledge_base, query):
    def ask(fact):
        # If fact is already inferred, return True
        if fact in inferred:
            return True

        # Find rules whose conclusion is fact
        matching_rules = [rule for rule in knowledge_base if
rule.conclusion == fact]

        # Try to prove premises of matching rules
        for rule in matching_rules:
            if all(ask(premise) for premise in
rule.premises):
                inferred.add(fact)
                return True

        return False

    inferred = set()
    return ask(query)


# Test backward chaining
print("\nBackward Chaining:")
print("Can prove E?", backward_chaining(rules, "E"))
```

Output :

```
Backward Chaining:
Can prove B? False
```

7.Write a program to implement Simple Chatbot Program using python

```
def chatbot_response(user_input):
    # Convert the input to lowercase to make the bot case
insensitive
    user_input = user_input.lower()

    # Simple keyword-based responses
    if "hello" in user_input or "hi" in user_input:
        return "Hello! How can I help you today?"
    elif "how are you" in user_input:
        return "I'm just a bot, but I'm here to help you! How can
I assist you?"
    elif "name" in user_input:
        return "I am a chatbot created by OpenAI. What's your
name?"
    elif "bye" in user_input or "goodbye" in user_input:
        return "Goodbye! Have a great day!"
    else:
        return "I'm sorry, I don't understand that. Can you
rephrase?"

# Main loop to interact with the chatbot
print("Welcome to the simple chatbot. Type 'bye' to exit.")
while True:
    user_input = input("You: ")
    if user_input.lower() == "bye":
        print("Chatbot: Goodbye! Have a great day!")
        break
    response = chatbot_response(user_input)
    print("Chatbot:", response)
```

OUTPUT:

```
Welcome to the simple chatbot. Type 'bye' to exit.
You: d
Chatbot: I'm sorry, I don't understand that. Can you
rephrase?

You: hi
Chatbot: Hello! How can I help you today?
You: d

You: bye
Chatbot: Goodbye! Have a great day!
```

8.Write a program to implement Hough circle transformation

```
import sys
import cv2 as cv
import numpy as np


def main(argv):

 default_file = 'v.jpg'
 filename = argv[0] if len(argv) > 0 else default_file
 # Loads an image
 src = cv.imread(cv.samples.findFile(filename), cv.IMREAD_COLOR)
 # Check if image is loaded fine
 if src is None:
     print ('Error opening image!')
     print ('Usage: hough_circle.py [image_name -- default ' +
default_file + '] \n')
     return -1


 gray = cv.cvtColor(src, cv.COLOR_BGR2GRAY)


 gray = cv.medianBlur(gray, 5)
```

```
rows = gray.shape[0]
circles = cv.HoughCircles(gray, cv.HOUGH_GRADIENT, 1, rows / 8,
param1=100, param2=30,
minRadius=1, maxRadius=30)


if circles is not None:
   circles = np.uint16(np.around(circles))

   for i in circles[0, :]:
        center = (i[0], i[1])
        # circle center
        cv.circle(src, center, 1, (0, 100, 100), 3)
        # circle outline
        radius = i[2]
        cv.circle(src, center, radius, (255, 0, 255), 3)


cv.imshow("detected circles", src)
cv.waitKey(0)

return 0
if __name__ == "__main__":
main(sys.argv[1:])
```

Output:purple color circle

9.Write a program to implement Template matching

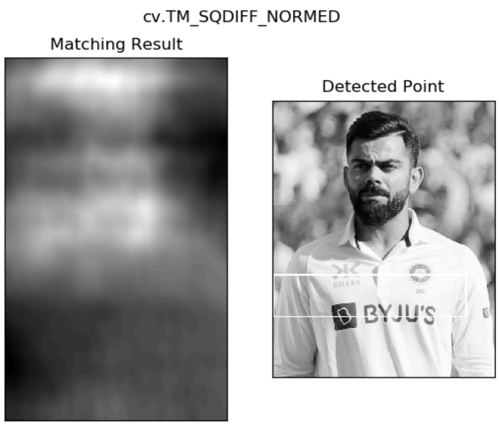
```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('v.jpg', cv.IMREAD_GRAYSCALE)
assert img is not None, "file could not be read, check with os.path.exists()"
img2 = img.copy()
template = cv.imread('template.jpg', cv.IMREAD_GRAYSCALE)
assert template is not None, "file could not be read, check with os.path.exists()"
w, h = template.shape[::-1]

# All the 6 methods for comparison in a list
methods = ['cv.TM_CCOEFF', 'cv.TM_CCOEFF_NORMED', 'cv.TM_CCORR', 'cv.TM_CCORR_NORMED', 'cv.TM_SQDIFF', 'cv.TM_SQDIFF_NORMED']

for meth in methods:
     img = img2.copy()
     method = eval(meth)
     res = cv.matchTemplate(img,template,method)
     min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)

 # If the method is TM_SQDIFF or TM_SQDIFF_NORMED, take minimum
     if method in [cv.TM_SQDIFF, cv.TM_SQDIFF_NORMED]:
         top_left = min_loc
     else:
         top_left = max_loc
         bottom_right = (top_left[0] + w, top_left[1] + h)

     cv.rectangle(img,top_left, bottom_right, 255, 2)

plt.subplot(121),plt.imshow(res,cmap = 'gray')
plt.title('Matching Result'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(img,cmap = 'gray')
```

```
plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
plt.suptitle(meth)

plt.show()
```

INPUT :

template.jpg v.jpg

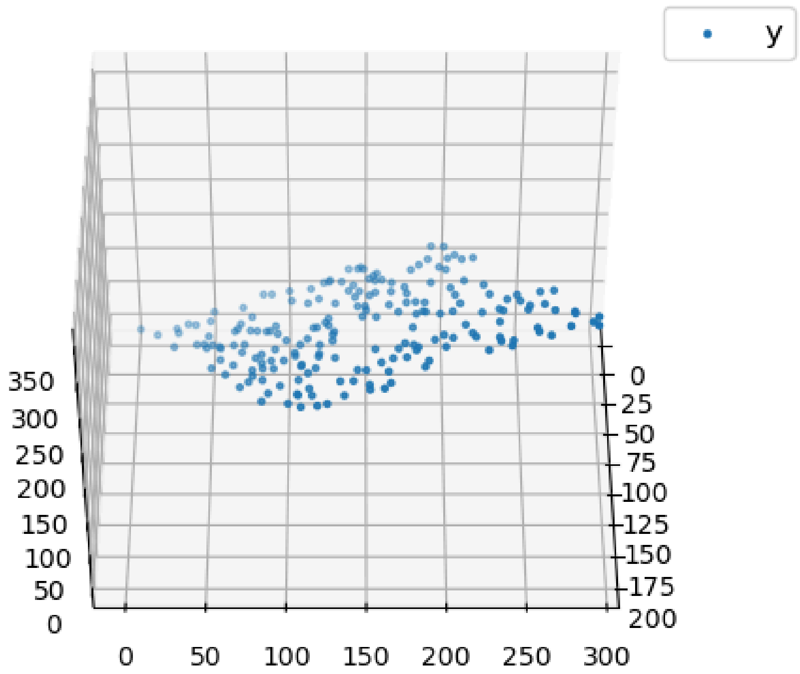
OUTPUT:



10. Write a program to implement Multiple Linear Regression Model using python

```
import numpy as np
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

def generate_dataset(n):
     x = []
     y = []
     random_x1 = np.random.rand()
     random_x2 = np.random.rand()
     for i in range(n):
          x1 = i
          x2 = i/2 + np.random.rand()*n
          x.append([1, x1, x2])
          y.append(random_x1 * x1 + random_x2 * x2 + 1)
     return np.array(x), np.array(y)

x, y = generate_dataset(200)

mpl.rcParams['legend.fontsize'] = 12

fig = plt.figure()
ax = fig.add_subplot(projection ='3d')

ax.scatter(x[:, 1], x[:, 2], y, label ='y', s = 5)
ax.legend()
ax.view_init(45, 0)

plt.show()
```

Output


y
0
50
100
150
200
250
300
350
0
50
100
150
200
250
300
0
25
50
75
100
125
150
175
200