PROGRAM :

```
import pandas as pd
import numpy as np
import re
import nltk
from sklearn.preprocessing import LabelEncoder,
MinMaxScalar from transformers import pipeline

nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))

data = pd.read_csv("/content/IMDB Dataset.csv")
print("Original Dataset:\n", data.head())

data['review'].fillna("No review provided", inplace=True)
data['sentiment'].fillna("unknown", inplace=True)

def clean_text(text):
    text = re.sub(r"<.*?>", " ", text)
    text = re.sub(r"[^a-zA-Z\s]", " ", text)
    text = text.lower()
    text = " ".join([w for w in text.split() if w not in
                     stop_words])
    return text

data['cleaned_review'] = data['review'].apply(clean_
                                              text)
encoder = LabelEncoder()

data['sentiment_encoded'] = encoder.fit_transform(data
                                                  ['sentiment'])
```

```
data['review_length'] = data['review'].apply(lambda x:
                                    len(x.split()))
scalar = MinMaxScaler()
data['review_length_norm'] = scaler.fit_transform(data
                                    [['review_length']])
data.to_csv("imdb_cleaned.csv", index=False)
print("\nProcessed Dataset sample:\n", data.head())
```

OUTPUT:

Processed dataset sample:

	review	sentiment
0	One of the other reviewers has ...	positive
1	A wonderful little production. The ...	Positive
2	I thought this was a wonderful way ...	negative

		sentiment_encoded
0	One reviewers mentioned watching ...	1
1	wonderful little production filming ...	1
2	thought wonderful way spend time ...	0

	review_length	review_length_norm
0	307	0.122371
1	162	0.064071
2	166	0.065693

PROGRAM:

```
from collections import deque

def bfs_shortest_path(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    directions = [(0,1), (1,0), (0,-1), (-1,0)]

    queue = deque([(start, [start])])
    visited = set()
    visited.add(start)

    while queue:
        (x, y), path = queue.popleft()
        if (x,y) == goal:
            return path
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols
            and grid[nx][ny] == 0 and (nx, ny) not
                                    in visited:
                visited.add((nx, ny))
                queue.append((nx, ny), path + [(nx, ny)])

    return None


def dfs_website(graph, start, visited = None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end = " -> ")
```

```
        for neighbor in graph.get(start, []):
            if neighbor not in visited:
                dfs_website(graph, neighbor, visited).


def main():
    print("choose algorithm:")
    print("1. BFS (Delivery Robot shortest Path)")
    print("2. DFS (Website Crawling)")
    choice = input("Enter your choice (1 or 2): ")

    if choice == "1":
        grid = [
            [0, 0, 0, 1, 0],
            [1, 1, 0, 1, 0],
            [0, 0, 0, 0, 0],
            [0, 1, 1, 1, 1],
            [0, 0, 0, 0, 0]
        ]
        start = (0, 0)
        goal = (4, 4)
        path = bfs_shortest_path(grid, start, goal)
        if path:
            print("Shortest Path found by BFS:",
                  path)
        else:
            print("No path found"):
```

```
elif choice == "2":
    website_graph = {
      "Home": ["About", "Products", "Contact"],
      "About": ["Team", "Careers"],
      "Products": ["Product1", "Product2"],
      "Contact": ["Email", "Phone"],
      "Team": [],
      "Careers": [],
      "Product1": [],
      "Product2": [],
      "Email": [],
      "Phone": []
    }
    print("DFS website crawling starting from 'Home': ")
    dfs_website(website_graph, "Home")
    print("END")
else:
    print("Invalid choice!")

if __name__ == "__main__":
    main()
```

OUTPUT :

Choose Algorithm:

1. BFS

2. DFS

Enter your choice: 1

Shortest path found by BFS: [(0,0), (0,1), (0,2), (1,2), (2,2), (2,1), (2,0), (3,0), (4,0), (4,1)]

PROGRAM:

```
import heapq
import random

def hill_climbing(cities, distance_matrix, max_iterations
                  =1000):

    current_route = cities[:]
    random.shuffle(current_route)

    def route_cost(route):
        return sum(distance_matrix[route[i]]
                   [route[i+1]] for i in range(len(route)-1))

    current_cost = route_cost(current_route)

    for _ in range(max_iterations):
        i, j = random.sample(range(len(cities)), 2)
        neighbor = current_route[:]
        neighbor[i], neighbor[j] = neighbor[j],
                                   neighbor[i]
        neighbor_cost = route_cost(neighbor)

        if neighbor_cost < current_cost:
            current_route, current_cost = neighbor,
                                          neighbor_cost.

    return current_route, current_cost

def a_star(graph, start, goal):
    def heuristic(a, b):
        (x1, y1) = a
        (x2, y2) = b
        return abs(x1 - x2) + abs(y1 - y2)
```

```
open_set = []
heapq.heappush(open_set, (0, start))
came_from = {}
g_score = {start: 0}
f_score = {start: heuristic(start, goal)}

while open_set:
    _, current = heapq.heappop(open_set)
    if current == goal:
        path = []
        while current in came_from:
            path.append(current)
            current = came_from[current]
        path.append(start)
        return path[::-1]

    for neighbor, cost in graph.get(current, []):
        tentative_g = g_score[current] + cost
        if tentative_g < g_score.get(neighbor,
                                     float('inf')):
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g
            f_score[neighbor] = tentative_g +
                                heuristic(neighbor, goal)
            heapq.heappush(open_set, (f_score[neighbor],
                                      neighbor))
return None
```

```
if __name__ == "__main__":
    cities = [0, 1, 2, 3]
    distance_matrix = [
                [0, 10, 15, 20],
                [10, 0, 35, 25],
                [15, 30, 0, 30],
                [20, 25, 30, 0]
            ]
    best_route, best_cost = hill_climbing(cities,
                                          distance_matrix)
    print("Best route (Hill climbing):", best_route)
    print("Best cost:", best_cost)
    graph = {
        (0,0): [((1,0), 1), ((0,1), 1)],
        (1,0): [((2,0), 1), ((1,1), 1), ((0,0), 1)],
        (0,1): [((0,0), 1), ((1,1), 1)],
        (1,1): [((1,0), 1), ((0,1), 1), ((2,1), 1)],
        (2,0): [((1,0), 1), ((2,1), 1)],
        (2,1): [((2,0), 1), ((1,1), 1)]
    }
    path = a_star(graph, (0,0), (2,1))
    print("Shortest path (A*):", path)
```

OUTPUT :

Best route (Hill climbing): [3, 1, 0, 2]
Best cost: 50
Shortest path (A*): [(0, 0), (0, 1), (1, 1), (2, 1)]

RESULT :

Hill climbing successfully optimized the travel route with reduced total distance. A* algorithm efficiently computed the best GPS path from the source to the destination.

PROGRAM :

```
def chatbot():
    print("Welcome to the college Helpdesk chatbot!")
    print("Ask me anything about admissions, courses, fees, library, or campus. \n")
    responses = {
        "admission": "Admissions are open from June to August. You can apply online through our college website.",
        "course": "We offer undergraduate and postgraduate courses in Engineering, Arts, Science, and Commerce.",
        "fee": "Tuition fees vary by course. Please specify your course for detailed fee structure.",
        "library": "The library is open from 8 AM to 8 PM on weekdays and has a vast collection of books and e-resources.",
        "campus": "Our campus has modern facilities including sports grounds, labs and cafeterias.",
        "scholarship": "Scholarships are available for meritorous and financially needy students.",
        "hostel": "Hostel accomodations are available for both boys and girls with all basic amenities.".
    }
```

```
while True:
    user_input = input("You: ").lower()
    if user_input in ["exit", "quit", "bye"]:
        print("chatbot: Thank you for using the
                College Helpdesk chatbot. Goodbye!")
        break

    response_given = False
    for keyword in responses:
        if keyword in user_input:
            print(f"chatbot: {responses[keyword]}")
            response_give = True
            break

    if not response_given:
        print("chatbot: Sorry, I didn't understand
            that. Can you please ask something else
            or be more specific?")

if __name__ == "__main__":
    chatbot()
```

OUTPUT:

Welcome to the college Helpdesk chatbot!
Ask me anything about admissions, courses, fees, library, or campus.

You : admission
chatbot : Admissions are open from June to August. You can apply through our college website

You : library
chatbot : The library is open from 8 AM to 8PM on weekday and has a vast collection of books and re-resource

You : bye
chatbot : Thank you for using the college Helpdesk chatbot. Goodbye!

NS
4/5

RESULT:

The chatbot successfully identified user intents like greetings, queries, thanks and farewells. It provided accurate, rule-based responses, simulating real-time conversations.

PROGRAM:

```
def health_inference_system(symptoms):
    symptoms = symptoms.lower()
    advice = []
    if "fever" in symptoms:
        advice.append("Drink plenty of fluids
            and take rest.").
        advice.append("Monitor temperature
            regularly.")
        advice.append("Consult a doctor if fever
            persists more than 3 days.")

    if "cough" in symptoms:
        advice.append("Drink warm fluids like
            ginger tea.")
        advice.append("Avoid cold drinks and dusty
            environments.")
        advice.append("Consult a doctor if cough lasts
            for more than 2 weeks.")

    if "headache" in symptoms:
        advice.append("Take a short break and relax
            in a quiet place.")
        advice.append("Drink water to stay hydrated.")
        advice.append("Sleep for at least 7-8 hrs a day.")

    if "fatigue" in symptoms or "tired" in symptoms:
        advice.append("Ensure balanced diet with enough
            proteins.")
        advice.append("Get adequate sleep.")
        advice.append("Take light exercises or short
            walks daily.")
```

```
    if "stomach" in symptoms or "abdominal
                    pain" in symptoms:
        advice.append("Avoid spicy/oily foods and
                    drink plenty of water.")
        advice.append("Eat smaller, more frequent
                    meals.")
        advice.append("Consult a doctor if pain is
                    severe or persistent.")
    if not advice:
        advice.append("No specific advice found.
                    Consider consulting a doctor for proper
                    diagnosis.")
    return advice

print("Welcome to the preventive Health tips System!")
print("
user_symptoms = input("Enter your symptoms
                    (comma seperated): ")
suggestions = health_inference_system(user_symptoms)
print("\n --- Preventive health Tips--- ")
for i, tip in enumerate(suggestions, start=1):
    print(f"{i}. {tip}")
```

OUTPUT:

Welcome to the Preventive Health Tips System!
Enter your symptoms (comma seperated): fever, cough.

--- Preventive Health Tips ---
1. Drink plenty of fluids and take rest.
2. Monitor Temperature regularly.
3. Consult a doctor if fever persists more than 3 days.
4. Drink warm fluids like ginger tea.
5. Avoid cold drinks and dusty environments.
6. Consult a doctor if cough lasts for more than 2 weeks.

RESULT:

Forward chaining successfully derived new facts {C, D, F} and updated the knowledge base. Backward chaining confirmed that the goal fact "F" can be logically derived from the initial facts and rules.

PROGRAM :

```
Pmport numpy as np
Pmport matplotlPb.pyplot. as plt
def solve(n, col=0, board=None, res=None):
    if board Ps None:
        board = np.zeros((n,n), Pnt)
        res = []
    if colS == n:
        res.append(aboard.copy())
        return res
    for r Pn range(n):
        Pf (board[r][:col] == 0).all().and \
        . all(board[r-i][col-i] ==0 for i Pn range(1,
                                    mPn(r, col)+1))and \
          all(board[r+i][col+i] ==0 for P in range(1,
                                    mPn(n-1-r, col)+1)):
            board[r][col] = 1
            solve(n, col+1, board, res)
            board[r][col] = 0
    return res

def shaw(sol, n, i):
    plt.PmshaO(np.add.outer(range(n), range(n))%2,
                                    emap = "gray")
    for r Pn range(n):
        for c in range(n):
            Pf sol[r][c]:
                plt.text(c, r, "rM", ha="eenter", va="center",
                    fontsPze=28, color="red")
    plt.tPtle(f"SolutPon {i+1} (n={n})"); plt.axis("off")
                                    plt.shaoc)
```

```
try:
    n = int(input("Enter n (1-9): "))
    if not 1 <= n <= 9:
        raise ValueError
except ValueError:
    print("Please enter an integer between 1 and 9.")
    return SystemExit

s = solve(n)
print(f"{len(s)} solutions found")
for i, s in enumerate(s): show(s, n, i)
```

OUTPUT :

```
Enter n (1-9): 5
10 solutions found
```

RESULT :

The program successfully generated all valid configurations of N-Queens for board sizes 1 to 9. It showed that N = 1 has 1 solution, N = 2 and N = 3 have no solutions, and higher values of N yield multiple solutions.