1.Clean and label a real-world dataset (e.g., food delivery, movie reviews) using AI-based data annotation tools to prepare it for training. Apply pre-processing steps like normalization, missing value handling, and encoding to improve dataset quality.

1. **Dataset Selection:** Choose a real-world dataset (e.g., *food delivery reviews*, *movie sentiment dataset*, etc.).

2. **Data Import:** Load the dataset using tools like `pandas.read_csv()` or from a database/API.

3. **AI-based Data Annotation:** Use AI annotation tools (e.g., **Label Studio**, **Amazon SageMaker Ground Truth**, or **Snorkel**) to:

    - Automatically generate or verify labels (e.g., sentiment = positive/negative).

    - Manually correct misclassified samples if needed.

4. **Data Cleaning:** Remove duplicate records.
   Handle missing values:

    - Drop rows/columns with too many nulls.

    - Fill missing values using mean/median/mode or model-based imputation.

5. **Text Cleaning (if applicable):** Remove special characters, stopwords, and extra spaces.Convert text to lowercase.Apply tokenization and lemmatization.

6. **Encoding Categorical Features:**Apply **Label Encoding** for binary columns.Apply **One-Hot Encoding** for multi-category columns.

7. **Normalization / Scaling:** Use **Min-Max Scaling** or **Standardization** to normalize numerical values.

8. **Data Splitting:** Split dataset into **training**, **validation**, and **test sets** (e.g., 70-20-10 ratio).

9. **Validation and Export:** Validate cleaned data.Save the processed dataset as `cleaned_data.csv` or a similar format.

2.Use Breadth First Search (BFS) to find the shortest path for a delivery robot navigating a city grid.Implement Depth First Search (DFS) to explore all connected parts of a website structure (like crawling pages).

1. Breadth First Search (BFS) – Shortest Path for Delivery Robot in City Grid:

1. **Input:**
    - `grid` representing the city (0 = free path, 1 = obstacle).
    - `start` and end coordinates.
2. **Initialize:**A **queue** to store nodes and their path.A **visited** matrix to mark visited cells.
3. **Enqueue** the `start` position with an empty path and mark it visited.
4. **While** the queue is not empty:
   a. Dequeue the current cell `(x, y)` and its path.
   b. If `(x, y)` is the **destination**, return the path as the shortest route.
   c. For each possible **movement** (up, down, left, right):
    - Check if the new cell is **within bounds**, not an obstacle, and not visited.
    - Mark it visited and enqueue with updated path.
5. **If no path found**, return “No possible route.”

🟦 2. Depth First Search (DFS) – Website Structure Exploration (Web Crawler)

1. **Input:**
    - A **graph** where each node is a web page and edges are hyperlinks.
    - A **starting URL**.
2. **Initialize:**A **stack** (or recursion) for DFS traversal.A **visited** set to avoid revisiting pages.
3. **Push** the starting URL onto the stack and mark it visited.

4. **While** the stack is not empty:
   a. Pop the top URL as `current_page`.
   b. **Process** (or log) the current page (e.g., print or store).
   c. For each **linked page** in `current_page`:If not visited, push it onto the stack and mark visited.

5. **Continue** until all reachable pages are visited.

3.Implement Hill Climbing to solve a simplified travel route optimization problem. Use A ^ * algorithm to simulate a GPS navigation system that finds the most efficient path in a city map.

1. Hill Climbing Algorithm – Travel Route Optimization

Algorithm:

1. **Input:** A set of possible routes between cities.A **cost function** (e.g., total distance or time).

2. **Initialize:** Start with a **random route** as the current solution.Compute its cost (heuristic value).

3. **Generate Neighbors:**Create nearby routes by **swapping cities** or making small changes to the path.

4. **Evaluate Neighbors:**Calculate the cost for each neighboring route.

5. **Move to Better State:**If a neighbor has **lower cost (better route)**, move to that neighbor.Otherwise, **stop** (local optimum reached).

6. **Repeat:**Continue steps 3–5 until no better neighbor exists or a stopping condition (like iteration limit) is reached.

7. **Output:**The best route found and its total cost.

2. A* (A-Star) Algorithm – GPS Navigation System

Algorithm:

1. **Input:**A city map as a **graph** (nodes = intersections, edges = roads with distances).A **heuristic function h(n)** estimating cost from node n to goal.

2. **Initialize:**

    - **Open list** (priority queue) with start node.
    - **Closed list** (visited nodes).
    - `g(n)` = actual cost from start to n.
    - `f(n) = g(n) + h(n)` (total estimated cost).

3. **While** the open list is not empty:
    a. Select the node n with the **lowest f(n)**.
    b. If n is the **goal**, return the path (reconstructed using parent links).
    c. Move n to the **closed list**.
    d. For each **neighbor** of n:

    - Compute tentative cost `g_new = g(n) + distance(n, neighbor)`.
    - If neighbor not in open/closed list or `g_new` < existing cost:
        - Update `g(neighbor)`, `f(neighbor)`, and **set parent** = n.
        - Add neighbor to open list.

4. **If** goal not found → no valid path exists.

5. **Output:**Optimal route and its total cost (distance/time).

4.Design and implement a basic chatbot that answers frequently asked student queries in a college helpdesk system.Ensure the chatbot provides different responses based on user input patterns using keyword mapping.

Algorithm Steps

1. **Start**

2. **Define Knowledge Base:**Create a dictionary (or JSON file) that maps **keywords** to **responses**.
    Example:

json

```json
{
  "admission": "Admissions are open till July 31. Apply online at college.edu/apply",
  "fees": "You can pay your fees online through the student portal.",
  "courses": "We offer B.Tech, M.Tech, MBA, and MCA programs.",
  "hostel": "Hostel facilities are available for both boys and girls.",
  "library": "The library is open from 8 AM to 8 PM on all working days."
}
```

3. **Accept User Input:**Take the user's query as a string.Convert it to **lowercase** for uniform matching.

4. **Keyword Matching:**For each keyword in the knowledge base:

   - If the keyword is found in the user's query → fetch the mapped response.
   - Break after finding the first match (or show multiple matches if needed).

5. **Handle Unknown Queries:**If no keyword matches, display a default response like:
   "I'm sorry, I didn't understand. Please contact the college office for more info."

6. **Repeat Interaction:**Keep asking for queries until the user types "exit" or "quit".

7. **End**

5.Build a simple inference system that suggests preventive health tips based on user-reported symptoms.Apply logical rules (IF-THEN) to infer suitable actions or advice from the input.

Algorithm Steps

1. **Start**

2. **Define Knowledge Base:**Create a set of **IF-THEN rules** linking symptoms to health advice.
   Example:

```
IF symptom = "fever" THEN advice = "Drink plenty of fluids and rest."
IF symptom = "cough" THEN advice = "Avoid cold drinks and consider consulting a doctor if pers
IF symptom = "headache" THEN advice = "Ensure proper hydration and sleep."
IF symptom = "fatigue" THEN advice = "Take balanced meals and exercise moderately."
```

3. **Accept User Input:**Ask the user to **report their symptoms** (comma-separated or one by one).Convert input to **lowercase** for uniform matching.

4. **Infer Advice:**For each reported symptom:

   - Match it against the knowledge base rules.
   - If a rule exists → **collect the corresponding advice**.
   - If no match → provide a generic suggestion: *"Consult a doctor if symptoms persist."*

5. **Display Results:**Show all inferred preventive tips to the user.

6. **Repeat Interaction:**Ask if the user wants to **report additional symptoms**.

7. **End**

6.Develop a solution for the N-Queens problem that visually displays valid arrangements for any n between 1 and 9.
Demonstrate how AI can solve constraint-based puzzle problems effectively.

Algorithm Steps (Backtracking Approach)

1. **Input:**n = size of the chessboard (1 ≤ n ≤ 9).

2. **Initialize:** An empty n×n board (2D array or list of lists).A list to store **all valid solutions**.

3. **Define Safety Function:** Check if placing a queen at `(row, col)` is safe:

   - No other queen in the same column.

            - No other queen in the **upper-left diagonal**.
            - No other queen in the **upper-right diagonal**.

4. **Backtracking Recursive Function:** `place_queen(row)`
   a. **Base Case:**
    - If `row == n`, add the current board to **solutions** and return.
      b. **Recursive Step:**
    - For each column `col` in 0..(n-1):
        - If `(row, col)` is **safe**:
            - Place queen at `(row, col)`.
            - Recursively call `place_queen(row + 1)`.
            - Remove queen (backtrack) for next column.
5. **Call Recursive Function:**
    - Start with `row = 0`.
6. **Display Solutions:**
    - For each solution:
        - Print or visually represent the board (`Q` for queen, `.` for empty).
        - Optionally, use **matplotlib** or **ASCII art** for visualization.
7. **End**