Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual
CS261
Artificial Intelligence

Preparing the scientific material

A.Prof. Ahmed El-Nagar

T.A. Osama Hefny

T.A. Ehab Ibrahim

What We'll Build in This Course

Course Objectives

- **Provide students with a solid foundation in classical and modern AI concepts.**
- **Develop problem-solving skills using AI techniques.**
- **Introduce AI applications in real-world domains.**
- **Prepare students for research or industry roles in AI.**

Intended Learning Outcomes (ILOs)

By the end of this course, students will be able to:

1. **Explain fundamental AI concepts, history, and ethical considerations.**
2. **Apply search algorithms to solve structured problems.**
3. **Design knowledge-based systems using logic and reasoning.**
4. **Implement machine learning algorithms for classification and prediction.**
5. **Apply probabilistic reasoning and handle uncertainty.**
6. **Use AI techniques in natural language processing (NLP), computer vision, and robotics.**
7. **Critically evaluate ethical, legal, and social implications of AI.**

Weekly Breakdown

Week 1: Introduction to AI

- History, definition, applications, and scope.
- AI vs Human Intelligence.
- Lab: Install Python, Jupyter, AI libraries (NumPy, Pandas, Scikit-learn).

Week 2: Intelligent Agents

- Types of agents, environments, agent architecture.
- Lab: Build a simple rule-based agent.

Week 3: Search Strategies (Uninformed Search)

- BFS, DFS, Uniform Cost Search.
- Lab: Implement search algorithms (maze-solving).

Week 4: Search Strategies (Informed Search & Optimization)

- A*, Greedy, Hill-Climbing, Genetic Algorithms.
- Lab: Solve an 8-puzzle problem using A*.

Week 5: Knowledge Representation

- Logic (propositional, predicate), semantic networks, frames.
- Lab: Prolog basics / Knowledge base implementation in Python.

Week 6: Reasoning and Inference

- Forward chaining, backward chaining, resolution.
- Lab: Expert system prototype.

Week 7: Machine Learning (Supervised Learning)

- Linear regression, decision trees, k-NN, SVM.
- Lab: Train and evaluate classifiers on a dataset.

Week 8: Machine Learning (Unsupervised Learning & Neural Networks)

- Clustering (k-means, hierarchical), introduction to neural networks.
- Lab: Implement k-means and a simple neural network.

Week 9: Probabilistic Reasoning & Bayesian Networks

- **Probability, Bayes theorem, Hidden Markov Models.**
- **Lab: Implement Naive Bayes for text classification.**

Week 10: Natural Language Processing (NLP)

- **Text processing, sentiment analysis, chatbots.**
- **Lab: Build a simple chatbot using NLP techniques.**

Week 11: Computer Vision & Perception

- **Image representation, edge detection, CNN basics.**
- **Lab: Image classification using a pre-trained CNN.**

Week 12: Course Review & Project Presentations

- **Student project demos (AI applications).**
- **Comprehensive revision for exam.**

Contents

Beni-Suif University
College of Computers and AI
Department of Computer Science

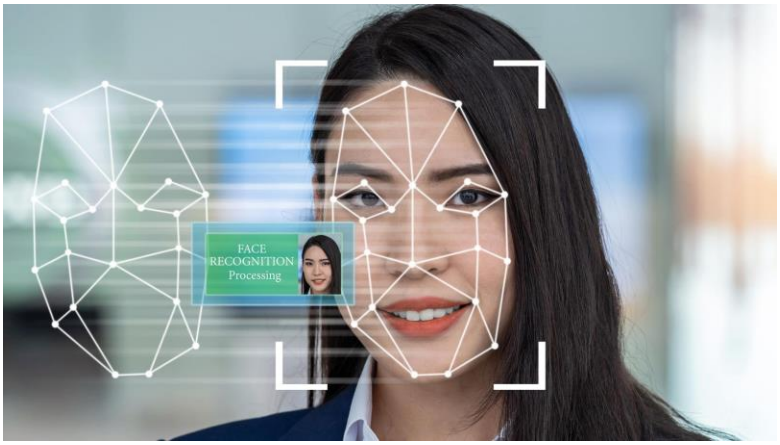
Lab Manual

Introduction to Artificial Intelligence

Lab - 1
Introduction to Neural Network

What is Artificial Intelligence?

- In simple terms: **Making computers think like humans.**
- It's about creating machines that can perform tasks that usually require human intelligence.
- **Examples:** Playing chess, recognizing faces in photos, recommending movies on Netflix, or understanding voice commands (like Siri or Alexa).

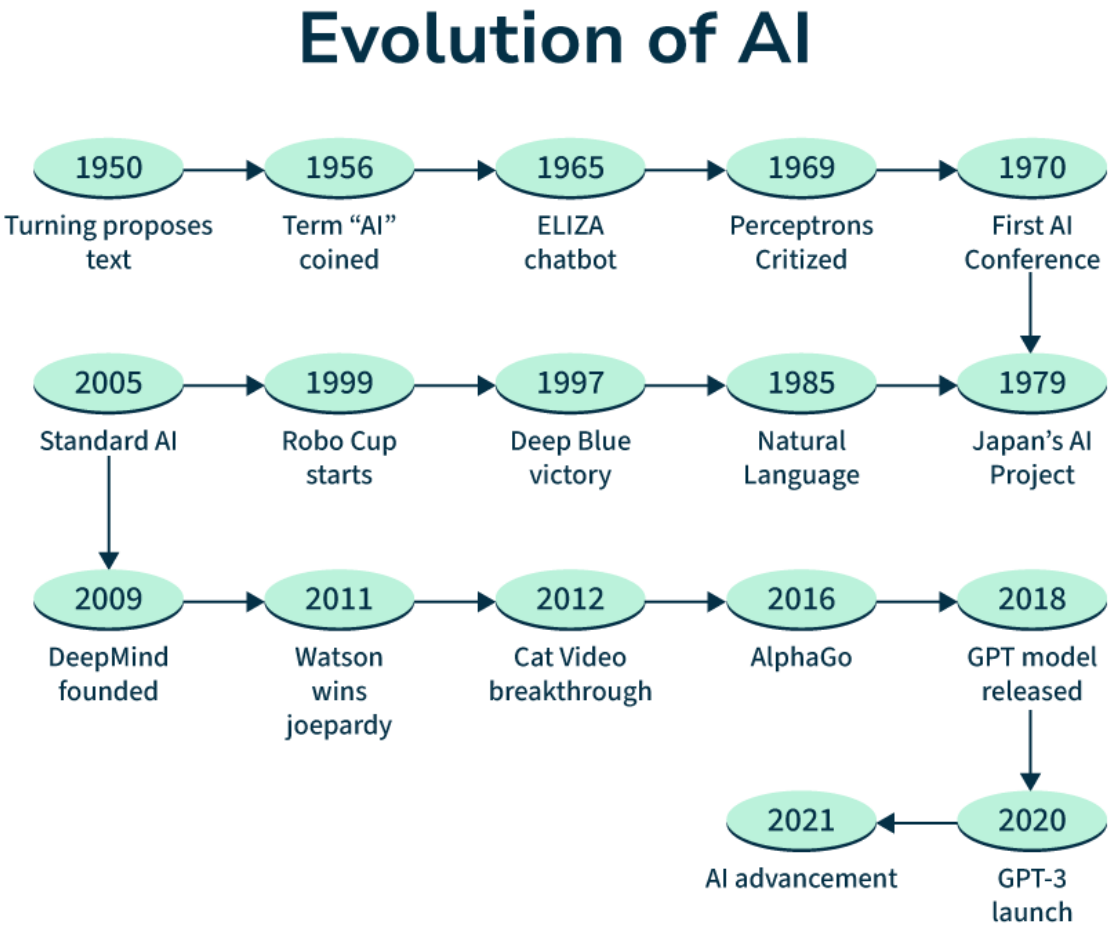


amazon alexa

Brief History of AI

Timeline:

- **1950s**: The idea is born. Alan Turing asks, "Can machines think?"
- **1956**: The term "Artificial Intelligence" is officially coined.
- **1997**: AI beats the world chess champion (Deep Blue).
- **2010s-Present**: AI is everywhere! Self-driving cars, smart assistants, and generative AI like ChatGPT.



AI vs. Human Intelligence

Artificial Intelligence (AI)	Human Intelligence
Processes information using algorithms and data patterns.	Processes information using reasoning, emotions, intuition, and experiences.
Learns from large datasets; performance depends on data quality.	Learns from real-world experiences, context, and social interactions.
Executes tasks quickly, consistently, and without fatigue.	Limited by physical and mental fatigue, but adaptable and creative.
Excels at repetitive tasks, pattern recognition, and computation.	Excels at abstract thinking, creativity, and complex problem-solving.
Lacks self-awareness, emotions, and consciousness.	Possesses self-awareness, emotions, consciousness, and empathy.
Cannot truly understand meaning—interprets patterns statistically.	Understands meaning, context, humor, irony, and subtle cues naturally.
Highly scalable—can operate across many machines simultaneously.	Limited to a single biological brain.
Requires power, hardware, maintenance, and updates.	Biological—self-healing and energy-efficient.
Makes decisions based solely on programmed rules and learned data.	Makes decisions influenced by ethics, morals, values, and judgment.
Can be biased depending on training data.	Can be biased due to personal experiences and beliefs.

Where Do We See AI? (Applications)

Healthcare

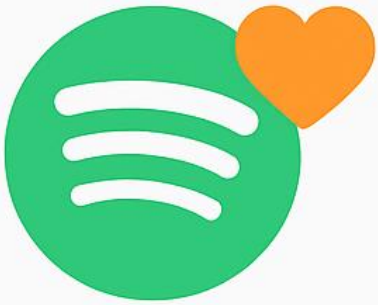
Helping doctors diagnose diseases from X-rays.

Finance

Detecting credit card fraud.

Transportation

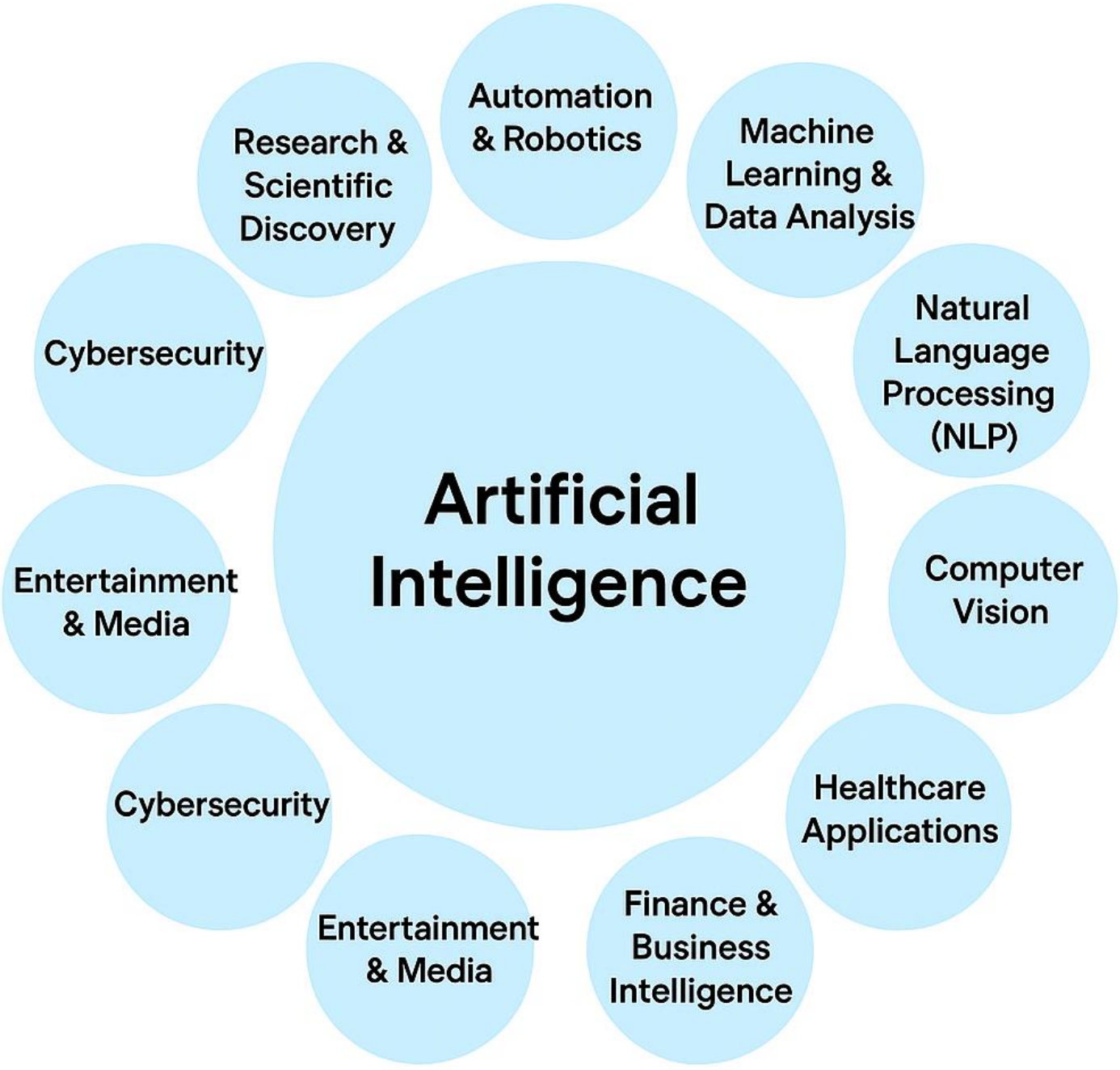
Powering the navigation in Google Maps.



Entertainment

Recommending your next favorite song on Spotify.

The Scope of AI



LAB: Setting Up Our Tools

Let's Get Our Hands Dirty!

Tools:

1. **Python: The programming language we will use (it's like English but for computers!).**
2. **Jupyter Notebook: An interactive "lab notebook" for writing and testing code step-by-step.**
3. **Libraries: Pre-built toolkits so we don't have to reinvent the wheel.**

installation steps

Step 1: Installing Python & Jupyter

Instructions:

1. **Go to python.org and download Python (version 3.x recommended).**
2. **IMPORTANT: Check the box that says "Add Python to PATH" during installation.**
3. **Open your terminal (command prompt).**
4. **Type this command and press Enter to install Jupyter.**

```
pip install jupyter
```

5. **To launch Jupyter, type:**

```
jupyter notebook
```

Step 2: Installing AI libraries:

- **NumPy: For doing math with lists of numbers (arrays).**
- **Pandas: For organizing data into tables (like Excel in Python).**
- **Scikit-learn: The main toolbox for simple Machine Learning algorithms.**

```
pip install numpy panadas scikit-learn
```

Our Very First Code!

Let's open a new Jupyter Notebook and write a simple program to make sure everything works.

```
# Import the numpy toolbox and give it a short nickname 'np'
import numpy as np

# Create a simple list of numbers
my_list = [1, 2, 3, 4, 5]

# Convert it into a numpy array (makes math easy!)
my_array = np.array(my_list)

# Print the average of the numbers in the array
print("The average is:", np.mean(my_array))

# Print each number multiplied by 2
print("Multiplied by 2:", my_array * 2)
```

Expected output:

```
The average is: 3.0
Multiplied by 2: [ 2  4  6  8 10 ]
```

Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Intelligent Agents

Lab - 2
Intelligent Agents

Lab - 2
Intelligent Agents

1. An overview

This lab introduces one of the most fundamental ideas in AI: the concept of an intelligent agent. By the end of this session, you will understand what makes something an 'agent', how agents interact with their environments, and the five main agent types. You will also write your first AI agent in Python.

Learning Objectives

After completing this lab, you should be able to:

- Define what an intelligent agent is and explain the perceive → decide → act loop.
- Use the PEAS framework to describe any agent's task environment.
- Identify the six properties used to classify environments.
- Name and compare the five types of agent programs.
- Implement a simple reflex agent and a model-based reflex agent in Python.
- Measure and compare agent performance using a scoring system.

2. Theory

What is an Intelligent Agent?

An agent is anything that perceives its environment through sensors and acts on it through actuators. The word 'agent' comes from the Latin agere — 'to do'. The key idea is simple: the agent observes the world and does something about it.

> **Key Idea**
> A rational agent always tries to do the right thing given what it knows. It chooses actions that maximize its performance measure based on its percept history and built-in knowledge.

The diagram below shows the basic loop that every agent follows, from a simple thermostat to a self-driving car:

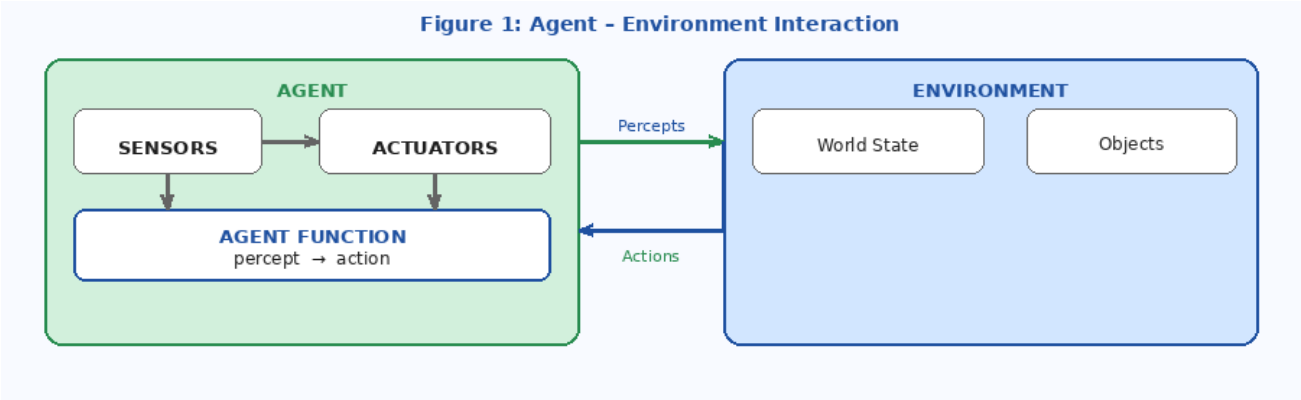


Figure 1 — The agent perceives the environment (via sensors) and acts on it (via actuators).

PEAS: Describing a Task Environment

Before designing any agent, we need to clearly describe its world. The PEAS framework gives us a standard way to do this with four questions:

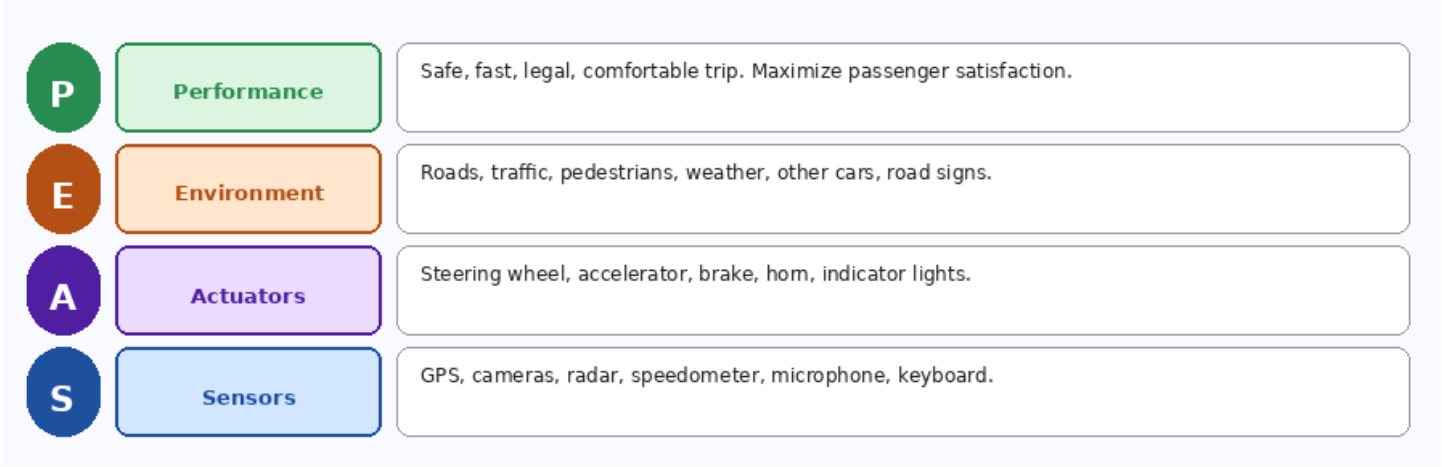


Figure 2 — PEAS applied to a self-driving taxi. Always fill in PEAS before writing any code.

Try this yourself: pick any everyday system (a spam filter, a hospital alarm, a recommendation engine) and fill in P, E, A, S before moving on.

Properties of Environments

Not all environments are the same. An environment can be described using six yes/no properties. The combination of these properties tells you which agent architecture you need:

Property	Option A	Option B
Observability	Fully Observable — agent sees everything	Partially Observable — limited view
Determinism	Deterministic — same action = same result	Stochastic — randomness involved
Episodes	Episodic — each action is independent	Sequential — actions affect future
Time	Static — world waits while agent thinks	Dynamic — world changes while thinking
State Space	Discrete — finite, countable states	Continuous — infinite states/actions
Agents	Single-agent	Multi-agent — others also acting

Quick Example
The Vacuum Cleaner World (our lab environment) is: Fully Observable, Deterministic, Sequential, Static, Discrete, Single-Agent. This makes it a good starting point — everything is simple and visible.

Five Types of Agent Programs
Russell & Norvig define five agent architectures. Think of them as levels, each one adds something to the previous:

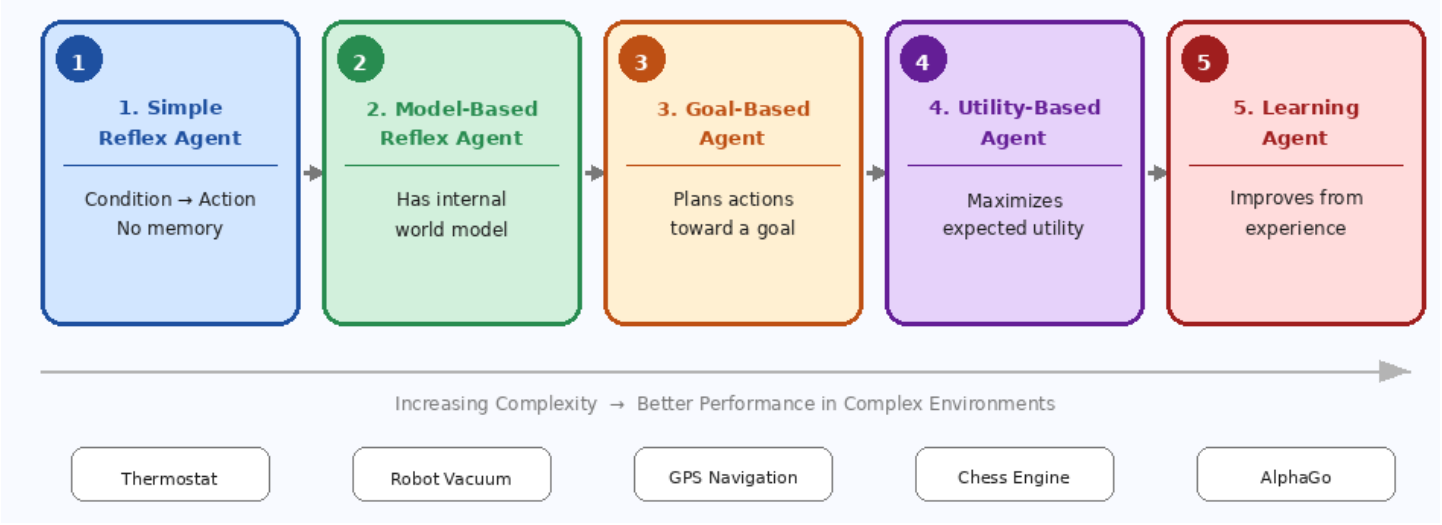


Figure 3 — The five agent types. Each level adds capability but also complexity.

Type 1: Simple Reflex Agent
The most basic agent. It looks at the current situation and applies a rule. That's it. No memory, no thinking ahead.

- Works only when the environment is fully observable.
- Everyday example: A thermostat. If temperature < 18°C → turn on heater.
- Problem: It can get stuck in loops because it forgets what it already did.

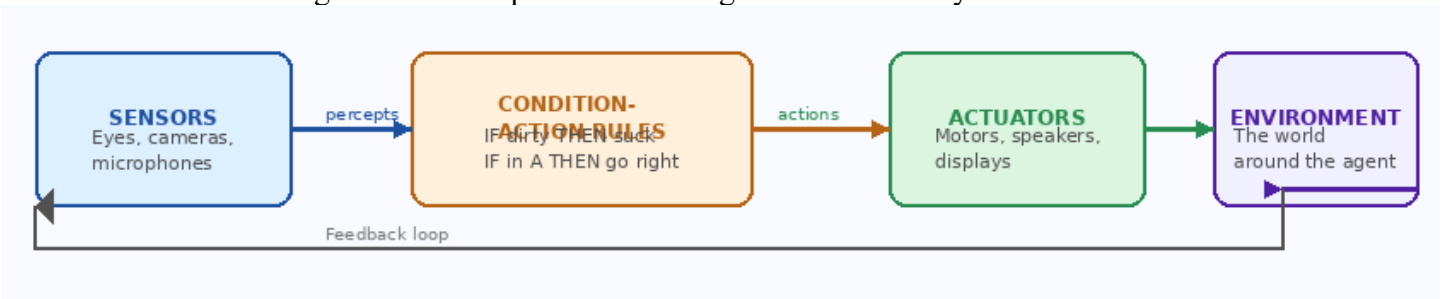


Figure 4 — Simple Reflex Agent. Percept goes in, action comes out. No memory.

Type 2: Model-Based Reflex Agent
Same as simple reflex, but now the agent keeps an internal model of the world — a kind of memory. It uses this to make better decisions when it can't see everything.

- Handles partially observable environments.
- Everyday example: A robot vacuum that remembers which rooms it already cleaned.

Type 3: Goal-Based Agent
This agent doesn't just react — it plans. It knows where it wants to go (the goal) and searches for a sequence of actions to get there.

- Everyday example: Google Maps routing — finding the best path to a destination.
- Requires search or planning algorithms (we'll cover these in Week 3!).

Type 4: Utility-Based Agent
Sometimes multiple paths lead to the goal. A utility-based agent picks the best one by assigning a score (utility) to each possible outcome.

- Everyday example: A chess engine — not just 'win', but 'win with the highest probability.'

Type 5: Learning Agent
This agent improves over time by learning from experience. It has four parts: a performance element (acts), a learning element (improves), a critic (gives feedback), and a problem generator (explores new situations).

- Everyday example: Netflix recommendations — gets better the more you watch.

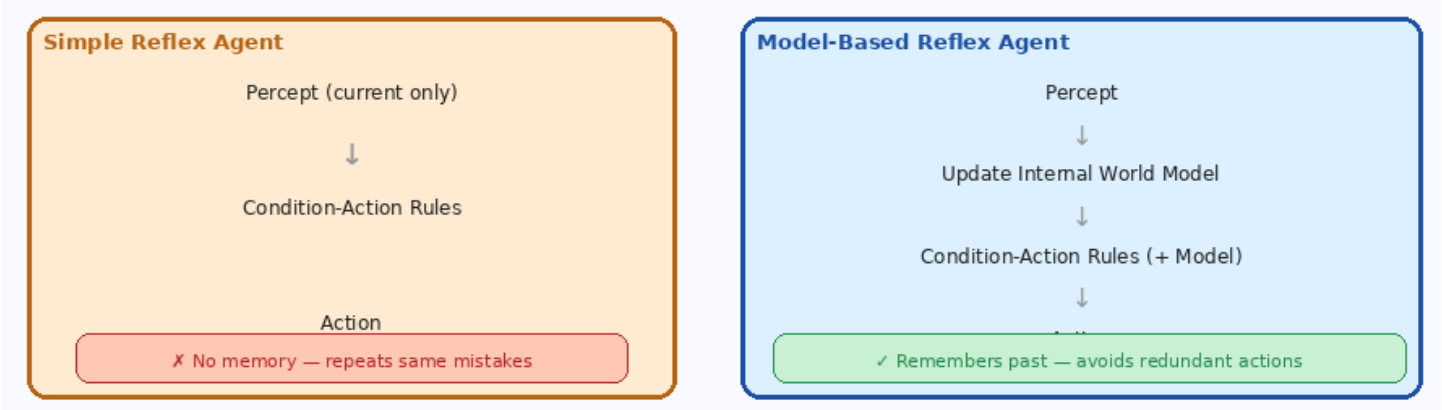


Figure 5 — Simple Reflex vs Model-Based Agent. The model gives the agent a 'memory' of the world.

3. Lab

The Environment

We'll use a classic AI environment from the textbook: a two-room vacuum cleaner world. The setup is simple on purpose; it lets us focus on agent design rather than complicated code.

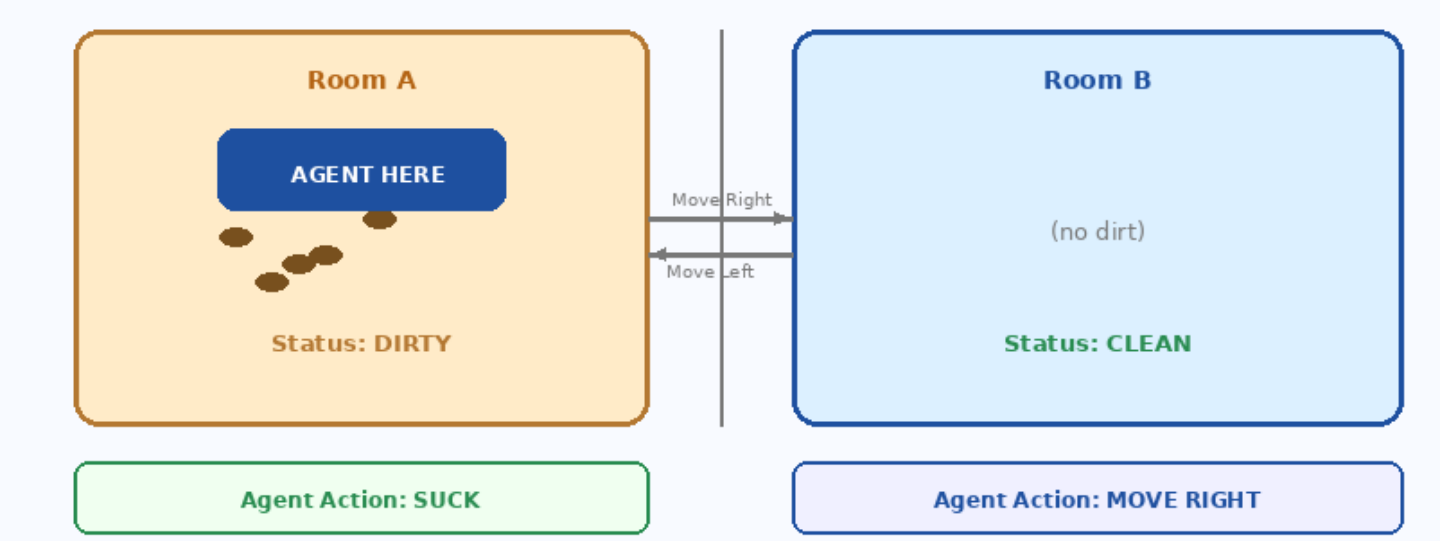


Figure 6 — The Vacuum World. Agent is in Room A (Dirty). It must decide: Suck, or Move?

Lab Goal — Exercise 1

Implement a simple reflex agent that cleans both rooms. Then in Exercise 2, upgrade it to a model-based agent. Compare performance scores and explain the difference.

PEAS for the Vacuum World

Before writing any code, let's apply the PEAS framework:

P — Performance	+10 points for each room cleaned. –1 point for each move or unnecessary suck.
E — Environment	Two rooms: A and B. Each is either Clean or Dirty.
A — Actuators	Three actions: Suck, Move Left, Move Right.
S — Sensors	Current location (A or B) and room status (Clean or Dirty).

Exercise 1: Simple Reflex Agent

Step 1 — The Environment (copy this code first)

This is the world our agent will live in. Read it carefully before writing the agent.

```
import random

class VacuumEnvironment:

    def __init__(self):

        # Randomly set each room as Clean or Dirty at the start

        self.rooms = {

            'A': random.choice(['Clean', 'Dirty']),

            'B': random.choice(['Clean', 'Dirty'])

        }

        self.agent_location = random.choice(['A', 'B'])

        self.performance = 0


    def get_percept(self):
```

```
    # The agent sees its current location and that room's status
    return (self.agent_location, self.rooms[self.agent_location])
def execute_action(self, action):
    if action == 'Suck':
        if self.rooms[self.agent_location] == 'Dirty':
            self.rooms[self.agent_location] = 'Clean'
            self.performance += 10   # reward for cleaning!
        else:
            self.performance -= 1    # penalty: wasted a suck
    elif action == 'Right':
        self.agent_location = 'B'
        self.performance -= 1        # small cost to move
    elif action == 'Left':
        self.agent_location = 'A'
        self.performance -= 1
def is_all_clean(self):
    return all(s == 'Clean' for s in self.rooms.values())
def show(self, step, action):
    a = self.rooms['A']
    b = self.rooms['B']
    loc = self.agent_location
    print(f'Step {step:2} | Action: {action:8} | A={a:5} B={b:5}
        | Agent@{loc} | Score: {self.performance}')
```

Step 2 — Write the Simple Reflex Agent

Now write the agent function. It receives a percept (location, status) and returns an action. Use only if/elif statements — no memory allowed!

```
def simple_reflex_agent(percept):
    location, status = percept
    # Rule 1: If the current room is dirty — clean it
    if status == 'Dirty':
        return 'Suck'
    # Rule 2: If in Room A and it's clean — go to B
    elif location == 'A':
        return 'Right'
    # Rule 3: If in Room B and it's clean — go back to A
    else:
        return 'Left'
# --- Run the simulation ---
def run(agent_fn, steps=12):
    env = VacuumEnvironment()
    print(f'Start | A={env.rooms["A"]:5} B={env.rooms["B"]:5}
          | Agent @ {env.agent_location}')
    for step in range(1, steps + 1):
        percept = env.get_percept()
        action  = agent_fn(percept)
        env.execute_action(action)
        env.show(step, action)
        if env.is_all_clean():
            print(f'Both rooms clean at step {step}!')
```

```
            break
    print(f'Final score: {env.performance}\n')
    return env.performance
run(simple_reflex_agent)
```

🔍 **Student Task 1**

Run the code 5 times. Write down the initial state and final score each time. Then answer: Why does the agent keep moving after both rooms are clean? What is the problem?

Exercise 2: Model-Based Reflex Agent

The simple reflex agent never stops moving after both rooms are clean — it wastes points going back and forth. Fix this by giving the agent a memory (internal model).

```
class ModelBasedAgent:
    def __init__(self):
        # Internal model: what we believe about each room
        self.model = {'A': 'Unknown', 'B': 'Unknown'}

    def __call__(self, percept):
        location, status = percept

        # Step 1: Update our model with what we see right now
        self.model[location] = status

        # Step 2: Decide action using rules + our model
        if status == 'Dirty':
            return 'Suck'                    # always clean first
```

```
        other = 'B' if location == 'A' else 'A'
        if self.model[other] != 'Clean':
            return 'Right' if location == 'A' else 'Left'  # go check other room

        return 'NoOp'                    # both clean — do nothing!


# --- Compare both agents ---
print('=== Simple Reflex Agent ===')
score1 = run(simple_reflex_agent, steps=15)

print('=== Model-Based Agent ===')
model_agent = ModelBasedAgent()
score2 = run(model_agent, steps=15)

print(f'Improvement: {score2 - score1:+d} points')
```

🔍 **Student Task 2**

Run both agents 10 times. Record the scores. Calculate the average for each. Which agent is better? Why does the model-based agent stop moving when both rooms are clean?

4. Summary

Here is everything we covered this week:

✦ What is an Agent?

Anything that perceives its environment through sensors and acts on it through actuators.

✦ Rational Agent

Chooses actions that maximize its performance measure given its percept history and knowledge.

✦ PEAS Framework

P = Performance | E = Environment | A = Actuators | S = Sensors

✦ Environment Properties

Observable, Deterministic, Episodic, Static, Discrete, Single-agent

✦ 5 Agent Types

Simple Reflex → Model-Based → Goal-Based → Utility-Based → Learning

✦ Key Lab Takeaway

Model-based agents outperform simple reflex agents because they remember the world state and avoid redundant actions.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Search Strategies

Uninformed Search: BFS, DFS, Uniform Cost Search

Lab - 3
Search Strategies

Uninformed Search: BFS, DFS, Uniform Cost Search

Lab - 3
Search Strategies
Uninformed Search: BFS, DFS, Uniform Cost Search

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Represent a problem as a state space graph (states, actions, transitions, cost).
- Implement BFS, DFS, and Uniform Cost Search from scratch.
- Understand when each algorithm is the right choice.
- Measure and compare: path length, nodes explored, memory used.

2. Theory

The Search Problem

Every search problem has four parts: a start state, a goal state, actions (what moves are possible), and a transition model (where each action leads). The algorithms below differ only in how they choose which node to expand next.

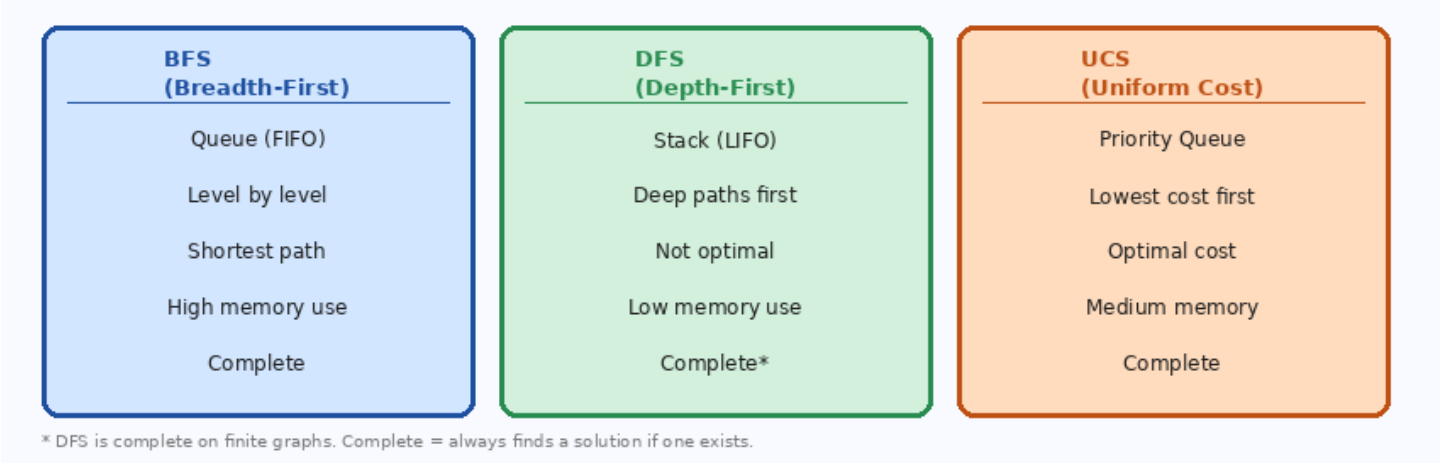


Figure 1 — BFS, DFS and UCS compared. The data structure each uses determines the search order.

Algorithm	When to use it
BFS	When you need the shortest path (fewest steps). All edge costs must be equal.
DFS	When memory is tight and any valid path is acceptable. Fast but not optimal.
UCS	When edge costs differ and you need the cheapest path. Generalizes BFS.

3. Lab – Maze Solver

The Maze Environment

We represent the maze as a 2D grid. 0 = open path, 1 = wall. The agent starts at (0,0) and must reach (4,4). Each step costs 1 unit.

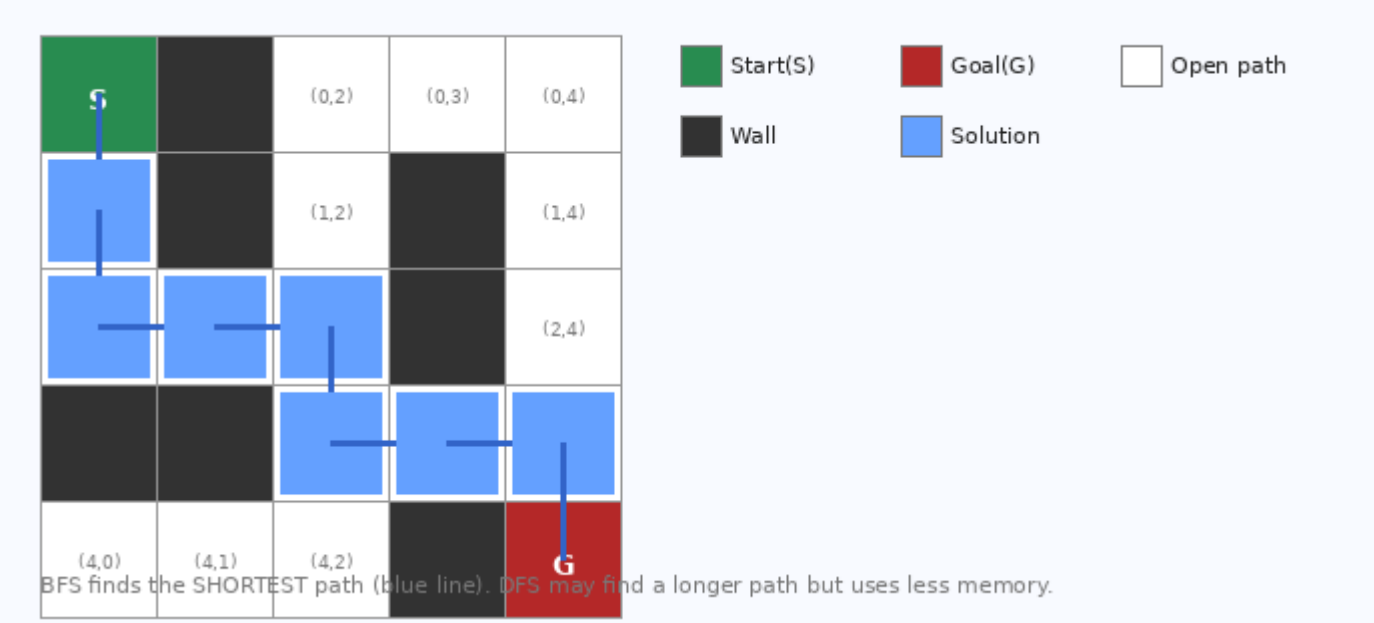


Figure 2 — 5×5 maze. Blue line shows the BFS solution path

Shared Setup Code

Copy this once. All three agents will use the same maze and graph helpers.

```
from collections import deque
import heapq

# 0 = open, 1 = wall
MAZE = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 0, 0],
    [0, 0, 0, 1, 0],
]
START = (0, 0)
GOAL  = (4, 4)
ROWS, COLS = 5, 5

def get_neighbors(pos):
    """Return valid adjacent cells (up, down, left, right)"""
    r, c = pos
```

```
    moves = [(-1,0),(1,0),(0,-1),(0,1)]   # up, down, left, right
    result = []
    for dr, dc in moves:
        nr, nc = r+dr, c+dc
        if 0 <= nr < ROWS and 0 <= nc < COLS and MAZE[nr][nc] == 0:
            result.append((nr, nc))
    return result

def reconstruct_path(came_from, goal):
    """Walk back through came_from dict to build the path"""
    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = came_from[node]
    return list(reversed(path))

def show_path(path):
    grid = [row[:] for row in MAZE]
    for r, c in path:
        grid[r][c] = '*'
    for row in grid:
        print(' '.join(str(x) for x in row))
    print(f'Path length: {len(path)-1} steps')
```

Exercise 1: BFS

BFS explores level by level using a queue (FIFO). It always finds the shortest path when all steps cost the same.

```
def bfs(start, goal):
    queue     = deque([[start]])      # each item is a PATH, not just a node
    visited   = {start}
    explored  = 0

    while queue:
        path = queue.popleft()        # take from front (FIFO)
        node = path[-1]              # current position
        explored += 1

        if node == goal:
            print(f'BFS: {explored} nodes explored')
            return path

        for neighbor in get_neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])  # extend path

    return None   # no path found

path = bfs(START, GOAL)
```

```
if path:
    print('BFS Solution:')
    show_path(path)
```

> 🔍 **Student Task 1**
>
> **Run BFS. Record: How many nodes were explored? How long is the path? Change the maze (add/remove walls) and run again. Does BFS always find the shortest path?**

Exercise 2: DFS

DFS dives as deep as possible using a stack (LIFO). It uses less memory than BFS but may find a longer path or get stuck exploring dead-ends first.

```
def dfs(start, goal):
    stack    = [[start]]             # stack of paths
    visited  = {start}
    explored = 0

    while stack:
        path = stack.pop()           # take from TOP (LIFO) — this is the only
        node = path[-1]             # difference from BFS!
        explored += 1

        if node == goal:
            print(f'DFS: {explored} nodes explored')
            return path

        for neighbor in get_neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append(path + [neighbor])

    return None

path = dfs(START, GOAL)
if path:
    print('DFS Solution:')
    show_path(path)
```

> 🔍 **Student Task 2**
>
> **Compare BFS vs DFS: nodes explored and path length. Why does DFS explore fewer nodes here? Is its path shorter or longer? Change deque→stack to stack→deque and confirm the behavior swap.**

Exercise 3: Uniform Cost Search

UCS uses a priority queue sorted by total path cost. For our maze all steps cost 1, so results equal BFS. To see the real difference, we add variable costs.

```
def ucs(start, goal):
    # heap entries: (total_cost, path)
    heap     = [(0, [start])]
    visited  = {}
    explored = 0

    while heap:
        cost, path = heapq.heappop(heap)   # lowest cost first
        node = path[-1]

        if node in visited:
            continue
        visited[node] = cost
        explored += 1

        if node == goal:
            print(f'UCS: {explored} nodes explored, total cost: {cost}')
            return path, cost

        for neighbor in get_neighbors(node):
            if neighbor not in visited:
                # Change step_cost to test variable costs
                step_cost = 1
                heapq.heappush(heap, (cost + step_cost, path + [neighbor]))

    return None, float('inf')

path, cost = ucs(START, GOAL)
if path:
    print(f'UCS Solution (cost={cost}):')
    show_path(path)
```

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Informed Search & Optimization

A*, Greedy, Hill-Climbing, Genetic Algorithms

Lab - 4

Informed Search & Optimization

A*, Greedy, Hill-Climbing, Genetic Algorithms

Lab - 4

Informed Search & Optimization

A*, Greedy, Hill-Climbing, Genetic Algorithms

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Understand how heuristics guide search and why h must be admissible.
- Implement A* from scratch using f(n) = g(n) + h(n).
- Calculate the Manhattan distance heuristic for the 8-puzzle.
- Compare A* against BFS on the same puzzle instance.
- Understand Hill-Climbing and why it can get stuck.

2. Theory

Why Informed Search?
Uninformed search (BFS/DFS) has no idea where the goal is — it explores blindly. Informed search uses a heuristic h(n): an estimate of how far node n is from the goal. A good heuristic dramatically reduces the number of nodes explored.

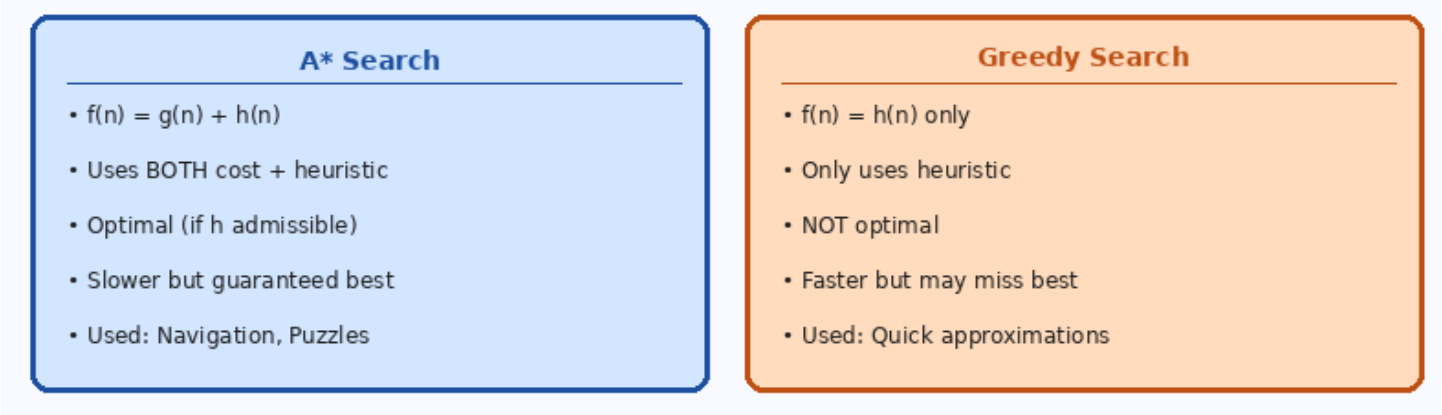


Figure 2 — A* combines actual cost g(n) with heuristic h(n). Greedy only uses h(n) and can miss the best path.

Term	Meaning
g(n)	Cost from start to current node n (actual, known)
h(n)	Estimated cost from n to goal (the heuristic)
f(n) = g(n) + h(n)	A*'s priority — always expand the lowest f(n) node
Admissible heuristic	h(n) NEVER overestimates the true cost — guarantees optimal solution
Manhattan distance	Sum of |row_diff| + |col_diff| for all tiles — admissible for 8-puzzle

1. Lab – 8-Puzzle with A*

The 8-Puzzle

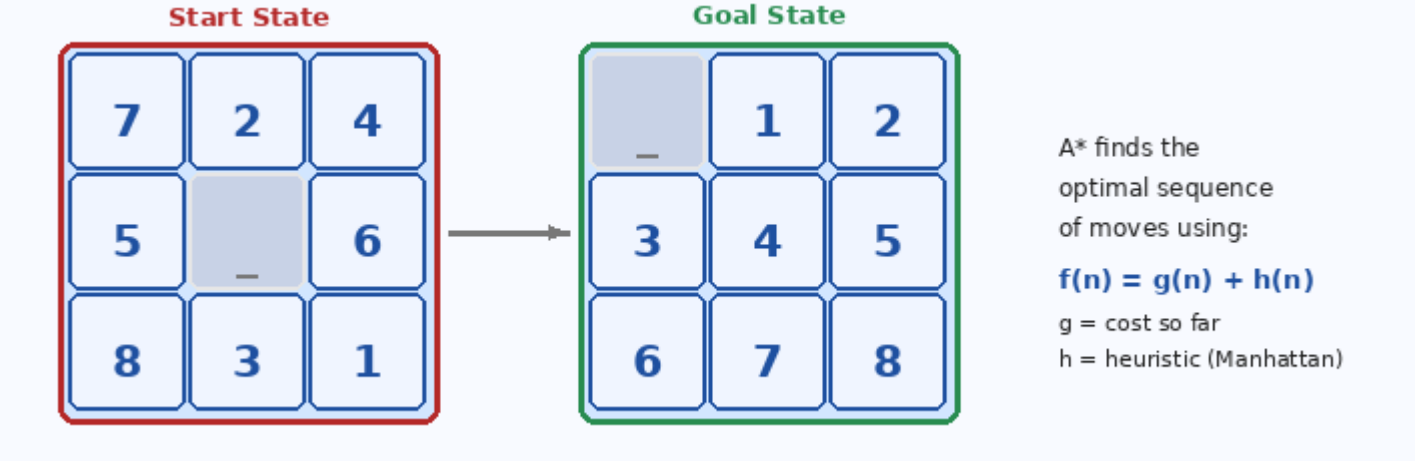


Figure 2 — The 8-puzzle. Slide tiles to reach the goal state. The blank (0) is the only moveable space.

The state is a tuple of 9 numbers. 0 is the blank. Legal moves slide a tile into the blank's position.

Setup — State Representation

```
import heapq

# States are tuples of 9 numbers (read row by row)
START = (7, 2, 4,
         5, 0, 6,
         8, 3, 1)

GOAL  = (0, 1, 2,
         3, 4, 5,
         6, 7, 8)

def get_moves(state):
    """Return all states reachable by sliding one tile"""
    state = list(state)
    blank = state.index(0)          # find the blank
    row, col = blank // 3, blank % 3
    moves = []
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        nr, nc = row+dr, col+dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            new_state = state[:]
            target = nr*3 + nc
            new_state[blank], new_state[target] = new_state[target],
new_state[blank]
            moves.append(tuple(new_state))
    return moves
```

```
def print_state(state):
    for i in range(0, 9, 3):
        row = state[i:i+3]
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()
```

Exercise 1: Manhattan Distance Heuristic

This is the heart of A*. For each tile, calculate how many rows + columns away it is from where it should be.

```
def manhattan(state, goal=GOAL):
    """
    For each tile (except blank), sum up the Manhattan distance
    from its current position to its goal position.
    """
    total = 0
    for tile in range(1, 9):           # skip the blank (0)
        current_idx = state.index(tile)
        goal_idx    = goal.index(tile)
        # Convert flat index to (row, col)
        cur_r,  cur_c  = current_idx // 3, current_idx % 3
        goal_r, goal_c = goal_idx    // 3, goal_idx    % 3
        total += abs(cur_r - goal_r) + abs(cur_c - goal_c)
    return total

# Quick test
print('Heuristic for START:', manhattan(START))   # should be > 0
print('Heuristic for GOAL: ', manhattan(GOAL))    # should be 0
```

🔍 Student Task 1

Calculate the Manhattan distance for START by hand. Pick tile 7: it's at position (0,0) but should be at (2,1). Distance = |0-2|+|0-1| = 3. Do this for all tiles and verify your function

Exercise 2: A* Search

```
def astar(start, goal):
    # heap: (f_score, g_score, state, path)
    h0 = manhattan(start)
    heap = [(h0, 0, start, [start])]
    visited = {}
    explored = 0

    while heap:
        f, g, state, path = heapq.heappop(heap)

        if state in visited:
            continue
        visited[state] = g
```

```
        explored += 1

        if state == goal:
            print(f'A*: solved in {g} moves, explored {explored} nodes')
            return path

        for next_state in get_moves(state):
            if next_state not in visited:
                new_g = g + 1
                new_h = manhattan(next_state)
                new_f = new_g + new_h
                heapq.heappush(heap, (new_f, new_g, next_state,
path+[next_state]))

    return None   # unsolvable

solution = astar(START, GOAL)
if solution:
    print(f'Total moves: {len(solution)-1}')
    print('\nFinal state:')
    print_state(solution[-1])
```

> 🔍 **Student Task 2**
>
> **Run A*. How many nodes does it explore? Now replace manhattan () with a constant 0 (making A* behave like BFS). Run again and compare node counts. This shows how much the heuristic helps.**

Exercise 3: Compare Heuristics

Try a weaker heuristic: count misplaced tiles (tiles not in their goal position). It's still admissible but less informed than Manhattan.

```
def misplaced_tiles(state, goal=GOAL):
    """Count tiles not in their correct position (excluding blank)"""
    return sum(1 for i in range(9) if state[i] != goal[i] and state[i] != 0)

def astar_with_heuristic(start, goal, h_fn):
    """Same A* but accepts any heuristic function"""
    heap = [(h_fn(start), 0, start, [start])]
    visited = {}
    explored = 0
    while heap:
        f, g, state, path = heapq.heappop(heap)
        if state in visited: continue
        visited[state] = g
        explored += 1
        if state == goal:
            return len(path)-1, explored
        for ns in get_moves(state):
            if ns not in visited:
```

```
                ng = g + 1
                heapq.heappush(heap, (ng+h_fn(ns), ng, ns, path+[ns]))
    return None, None

moves_m, exp_m = astar_with_heuristic(START, GOAL, manhattan)
moves_t, exp_t = astar_with_heuristic(START, GOAL, misplaced_tiles)

print(f'Manhattan    : {moves_m} moves, {exp_m} nodes explored')
print(f'Misplaced    : {moves_t} moves, {exp_t} nodes explored')
print(f'Winner       : Manhattan (explores {exp_t - exp_m} fewer nodes)')
```

Challenge
Implement a greedy best-first search: remove g from f(n), using only h(n). Compare the number of nodes it explores vs A*. Does it always find the optimal solution?

Exercise 4: Hill-Climbing

Hill-climbing is a local search algorithm. Instead of storing all paths, it only keeps the current state and moves to the best neighbor. Think of it like hiking: always walk uphill.

```
def hill_climbing(start, goal):
    current = start
    steps = 0

    while current != goal:
        neighbors = get_moves(current)
        # Pick the neighbor with the lowest heuristic (closest to goal)
        best = min(neighbors, key=manhattan)

        if manhattan(best) >= manhattan(current):
            print(f'Stuck at local minimum after {steps} steps!')
            print('Current state:')
            print_state(current)
            return None

        current = best
        steps += 1

    print(f'Hill-climbing solved in {steps} steps')
    return current

# Hill-climbing often gets STUCK — try it!
result = hill_climbing(START, GOAL)
```

Student Task 3

Run hill-climbing. Does it solve the puzzle or get stuck? Why does it fail? What property of A* prevents this problem? (Hint: A* keeps all explored paths.)

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Knowledge Representation

Logic, Semantic Networks, Frames, Knowledge Bases

Lab - 5

Knowledge Representation

Logic, Semantic Networks, Frames, Knowledge Bases

Lab - 5

Knowledge Representation

Logic, Semantic Networks, Frames, Knowledge Bases

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Distinguish propositional logic from first-order predicate logic.
- Build a semantic network in Python and query inherited properties.
- Implement a knowledge base with facts and IF-THEN rules.
- Write a simple forward-chaining reasoner over the knowledge base.

2. Theory

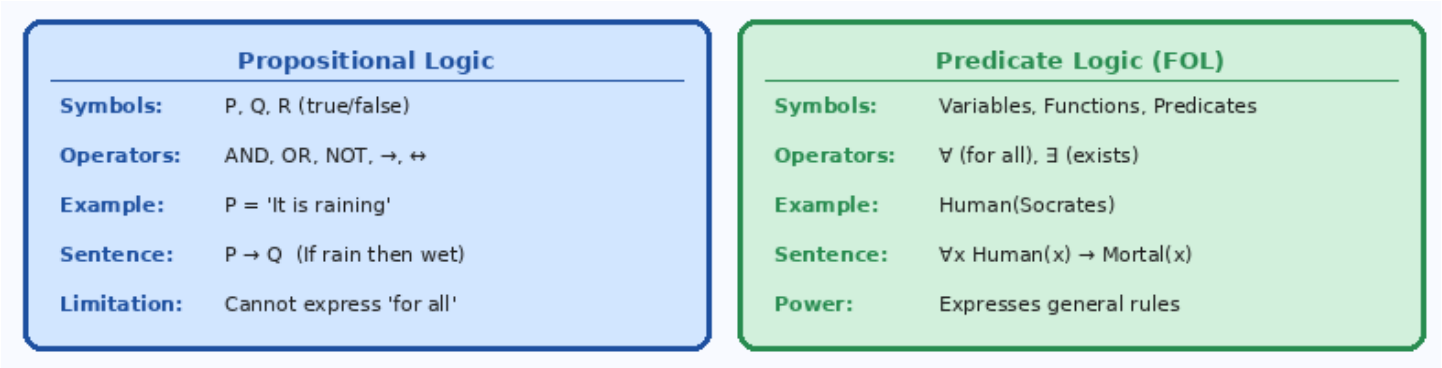


Figure 3 — Propositional vs Predicate Logic. FOL is far more expressive

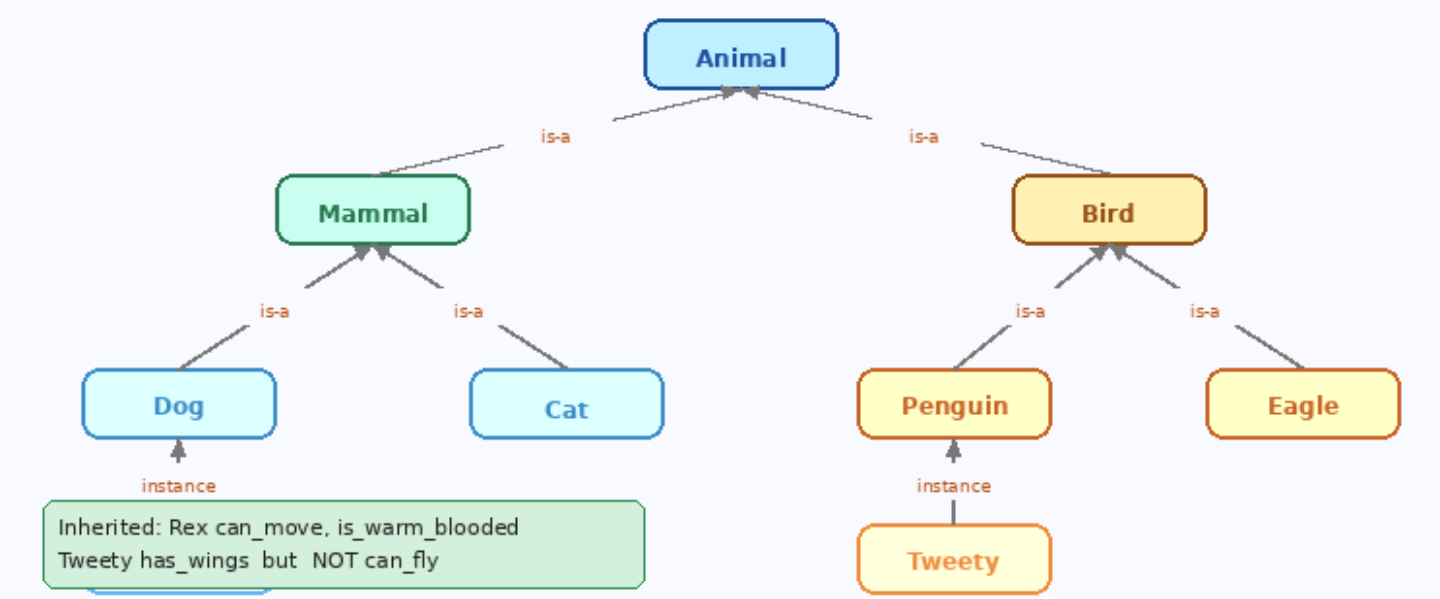


Figure 2 — Semantic Network. Nodes = concepts, edges = relationships. Properties are inherited upward.

Representation	Key Idea
Propositional Logic	True/false symbols + AND/OR/NOT/→. No variables.
Predicate Logic (FOL)	Variables, functions, quantifiers (∀, ∃). Full generality.
Semantic Network	Graph of concepts linked by 'is-a' and 'has-property' edges.
Frames	Structured record: object + slots (attributes) + default values.

3. Lab

Exercise 1: Semantic Network in Python

We implement the animal network from Figure 2. Each node is a dictionary of properties. Inheritance means checking parent nodes when a property isn't found locally.

```
# Semantic Network — animal taxonomy

network = {
    'Animal':  {'parent': None,     'can_move': True, 'alive': True},
    'Mammal':  {'parent': 'Animal', 'warm_blooded': True, 'has_hair': True},
    'Bird':    {'parent': 'Animal', 'has_wings': True, 'lays_eggs': True},
    'Dog':     {'parent': 'Mammal', 'has_tail': True, 'sound': 'bark'},
    'Cat':     {'parent': 'Mammal', 'has_tail': True, 'sound': 'meow'},
    'Penguin': {'parent': 'Bird',   'can_fly': False, 'can_swim': True},
    'Eagle':   {'parent': 'Bird',   'can_fly': True},
    # Instances
    'Rex':     {'parent': 'Dog',    'name': 'Rex', 'age': 3},
    'Tweety':  {'parent': 'Penguin','name': 'Tweety'},
}

def get_property(node, prop):
    """Look up a property, inheriting from parents if not found locally"""
    current = node
    while current is not None:
        data = network.get(current, {})
        if prop in data:
            return data[prop]
        current = data.get('parent')   # move up the hierarchy
    return None   # not found anywhere

# --- Test queries ---
print('Can Rex move?  ', get_property('Rex', 'can_move'))      # True (from
Animal)
print('Rex sound?     ', get_property('Rex', 'sound'))          # bark (from Dog)
print('Tweety fly?    ', get_property('Tweety', 'can_fly'))     # False (from
Penguin)
print('Tweety wings?  ', get_property('Tweety', 'has_wings'))   # True (from Bird)
```

> 🔍 **Student Task 1**
>
> **Add a new animal: 'Bat' (parent: Mammal, can_fly: True). Add an instance 'Bruce' (parent: Bat). Then query: does Bruce have hair? Can Bruce fly? Which node provides each answer?**

Exercise 2: Frames

Frames extend semantic networks with structured slots and default values. They are like classes in Python.

```
class Frame:
    def __init__(self, name, parent=None, **slots):
        self.name   = name
        self.parent = parent
        self.slots  = slots

    def get(self, slot):
        """Check own slots first, then inherit from parent frame"""
        if slot in self.slots:
            return self.slots[slot]
        if self.parent:
            return self.parent.get(slot)
        return None

    def set(self, slot, value):
        self.slots[slot] = value

# Build the frame hierarchy
animal  = Frame('Animal', can_move=True, alive=True)
mammal  = Frame('Mammal', parent=animal, warm_blooded=True)
dog     = Frame('Dog',    parent=mammal, sound='bark', legs=4)
rex     = Frame('Rex',    parent=dog,    name='Rex', age=3)

print(rex.get('warm_blooded'))   # True — inherited from Mammal
print(rex.get('sound'))          # 'bark' — inherited from Dog
print(rex.get('name'))           # 'Rex' — own slot

# Override a slot for a specific instance
rex.set('sound', 'woof')         # Rex barks differently!
print(rex.get('sound'))          # 'woof'
```

Exercise 3: Knowledge Base with Rules

Now we build a proper KB: a set of facts (known truths) and rules (IF...THEN logic). A forward-chaining engine derives new facts automatically.

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()    # known true statements
        self.rules = []       # list of (conditions, conclusion) tuples

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, conditions, conclusion):
        """conditions = list of facts that must ALL be true"""
        self.rules.append((conditions, conclusion))

    def ask(self, fact):
        return fact in self.facts
```

```
    def forward_chain(self):
        """Keep applying rules until no new facts are derived"""
        changed = True
        while changed:
            changed = False
            for conditions, conclusion in self.rules:
                if all(c in self.facts for c in conditions):
                    if conclusion not in self.facts:
                        self.facts.add(conclusion)
                        print(f'  Derived: {conclusion}')
                        changed = True

# --- Build a weather/clothing KB ---
kb = KnowledgeBase()

# Initial facts
kb.add_fact('raining')
kb.add_fact('cold')
kb.add_fact('going_outside')

# Rules
kb.add_rule(['raining'],                    'wet_outside')
kb.add_rule(['cold', 'going_outside'],       'need_jacket')
kb.add_rule(['wet_outside', 'going_outside'], 'need_umbrella')
kb.add_rule(['need_jacket', 'need_umbrella'], 'fully_prepared')

print('Initial facts:', kb.facts)
print('\nRunning forward chaining...')
kb.forward_chain()
print('\nFinal facts:', kb.facts)
print('\nAm I fully prepared?', kb.ask('fully_prepared'))
```

🔍 Student Task 2

Add three more facts and two more rules of your own to the KB. Examples: 'have_car', 'traffic_jam', 'need_extra_time'. Run forward_chain() again and trace which rules fire.

Exercise 4: Logic Sentences

Translate these English sentences into propositional logic, then add them as rules to your KB.

```
# Translate these into KB rules:
#
# 1. 'If it is sunny and warm, go to the beach.'
# 2. 'If you have a fever OR a cough, you might be sick.'
# 3. 'If you study hard AND attend lectures, you will pass.'
#
# Template:
# kb.add_rule(['condition1', 'condition2'], 'conclusion')

kb2 = KnowledgeBase()

# Add your facts and rules here:
# kb2.add_fact('sunny')
# kb2.add_fact('warm')
# kb2.add_rule(['sunny', 'warm'], 'go_to_beach')

# TODO: Complete the other two rules

kb2.forward_chain()
```

Bonus — Predicate Logic

Extend your KB to handle variables. For example, add a function that takes a subject (like 'Socrates') and checks if it is human, then infers it is mortal. This is a step toward Prolog.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Reasoning & Inference

Forward Chaining, Backward Chaining, Expert System

Lab - 6

Reasoning & Inference

Forward Chaining, Backward Chaining, Expert System

Lab - 6

Reasoning & Inference

Forward Chaining, Backward Chaining, Expert System

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Understand the difference between forward and backward chaining.
- Implement both reasoning strategies on the same knowledge base.
- Build a complete expert system with explanation capability.
- Understand when each strategy is more appropriate.

2. Theory

Figure 4 — Forward chaining starts from facts. Backward chaining starts from the goal.

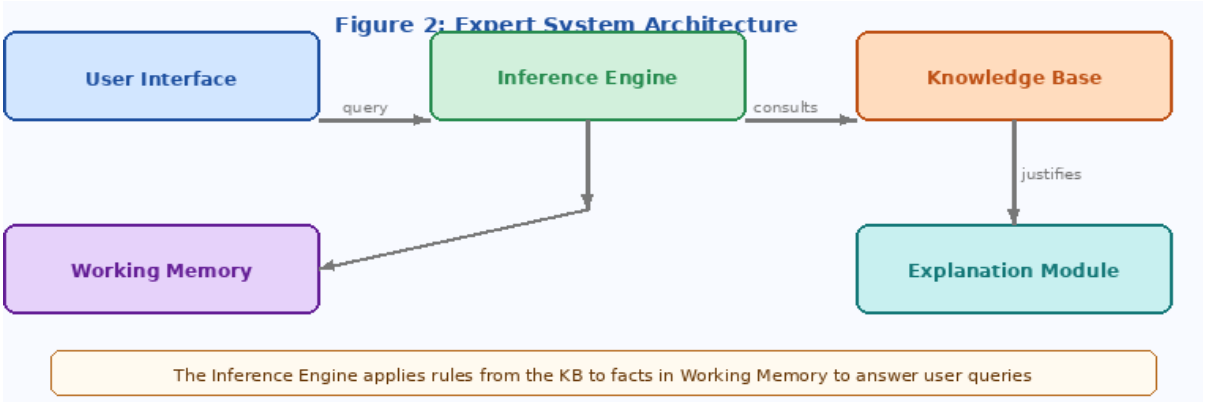


Figure 2 — Expert System architecture. The Inference Engine is the brain that applies rules.

Strategy	Use When...
Forward Chaining	You have facts and want to see what you can conclude (data-driven).
Backward Chaining	You have a hypothesis and want to check if it's provable (goal-driven).

3. Lab

The Knowledge Base

Our expert system diagnoses three plant diseases based on observable symptoms. This mirrors real expert systems used in agriculture, medicine, and fault-detection.

```
# ============================================================
#  Plant Disease Expert System
# ============================================================

# Each rule: ( list_of_conditions, conclusion, explanation )
RULES = [
    # Powdery Mildew
    (['white_powder_on_leaves', 'dry_conditions'],
     'powdery_mildew',
     'White powder + dry conditions indicate Powdery Mildew.'),

    # Root Rot
    (['yellow_leaves', 'soggy_soil', 'wilting'],
     'root_rot',
     'Yellow leaves + soggy soil + wilting indicate Root Rot.'),

    # Leaf Blight
    (['brown_spots', 'leaf_edges_dying'],
     'leaf_blight',
     'Brown spots + dying edges indicate Leaf Blight.'),

    # Intermediate rules
    (['overwatering'],             'soggy_soil',
     'Overwatering causes soggy soil.'),
    (['high_humidity', 'no_air'],  'dry_conditions',
     'High humidity + poor airflow create mildew conditions.'),
    (['root_rot', 'leaf_blight'],  'serious_infection',
     'Multiple diseases — serious infection detected.'),
]

# Starting symptoms (what we observe)
INITIAL_FACTS = {
    'white_powder_on_leaves',
    'high_humidity',
    'no_air',
}
```

Exercise 1: Forward Chaining Engine

Forward chaining: start from observed symptoms, apply every matching rule, accumulate new facts. Keep going until nothing new can be derived.

```
def forward_chain(facts, rules):
    """
    Start from facts, fire every applicable rule,
    collect new conclusions. Repeat until stable.
    """
    facts    = set(facts)     # make a copy
    log      = []             # audit trail
    changed  = True

    while changed:
        changed = False
        for conditions, conclusion, explanation in rules:
            if all(c in facts for c in conditions):
                if conclusion not in facts:
                    facts.add(conclusion)
                    log.append((conditions, conclusion, explanation))
                    changed = True

    return facts, log

print('=== Forward Chaining Diagnosis ===')
print('Observed symptoms:', INITIAL_FACTS)
print()

final_facts, trace = forward_chain(INITIAL_FACTS, RULES)

print('Reasoning trace:')
for conditions, conclusion, explanation in trace:
    cond_str = ' + '.join(conditions)
    print(f'  [{cond_str}]  =>  {conclusion}')
    print(f'    Reason: {explanation}')

print()
diseases = ['powdery_mildew', 'root_rot', 'leaf_blight', 'serious_infection']
print('Diagnoses:')
for d in diseases:
    status = 'DETECTED' if d in final_facts else 'not found'
    print(f'  {d}: {status}')
```

🔍 Student Task 1

Run the forward chainer. Which disease is detected? Now add 'overwatering', 'yellow_leaves', and 'wilting' to INITIAL_FACTS and run again. What new diseases appear? Which rule fires last?

Exercise 2: Backward Chaining Engine

Backward chaining: start with a hypothesis (e.g. 'Does the plant have powdery_mildew?'). Find rules whose conclusion matches. Then try to prove all the rule's conditions recursively.

```
def backward_chain(goal, facts, rules, depth=0, visited=None):
    """
    Recursively try to prove 'goal' from facts and rules.
    Returns (True/False, explanation_steps).
    """
    if visited is None: visited = set()
    indent = '  ' * depth

    # Base case: goal is already a known fact
    if goal in facts:
        print(f'{indent}✓ {goal}  (known fact)')
        return True, [f'{goal} is a known fact']

    if goal in visited:
        return False, ['(circular — already tried)']
    visited.add(goal)

    # Try every rule whose conclusion matches our goal
    for conditions, conclusion, explanation in rules:
        if conclusion == goal:
            print(f'{indent}Trying rule: {conditions} => {goal}')
            # Try to prove all conditions of this rule
            all_proved = True
            proof = [explanation]
            for cond in conditions:
                proved, sub_proof = backward_chain(
                    cond, facts, rules, depth+1, visited.copy())
                if not proved:
                    all_proved = False
                    break
                proof.extend(sub_proof)
            if all_proved:
                print(f'{indent}✓ Proved: {goal}')
                return True, proof

    print(f'{indent}✗ Cannot prove: {goal}')
    return False, []

print('=== Backward Chaining ===')
for hypothesis in ['powdery_mildew', 'root_rot', 'serious_infection']:
    print(f'\nCan we prove: {hypothesis}?')
    proved, proof = backward_chain(hypothesis, INITIAL_FACTS, RULES)
    print(f'Result: {"YES" if proved else "NO"}')
```

> 🔍 **Student Task 2**
> **Add 'overwatering' to INITIAL_FACTS and re-run backward chaining for 'root_rot'. Trace the reasoning: what sub-goals does it try? Draw the proof tree on paper.**

Exercise 3: Interactive Expert System

Now connect everything into an interactive system that asks the user about symptoms and gives a diagnosis with explanations.

```
ALL_SYMPTOMS = [
    'white_powder_on_leaves', 'dry_conditions', 'yellow_leaves',
    'soggy_soil', 'wilting', 'brown_spots', 'leaf_edges_dying',
    'overwatering', 'high_humidity', 'no_air'
]

def run_expert_system():
    print('=== Plant Disease Expert System ===')
    print('Answer yes/no for each symptom:\n')

    observed = set()
    for symptom in ALL_SYMPTOMS:
        ans = input(f'  Does the plant show: {symptom}? (y/n): ').strip().lower()
        if ans == 'y':
            observed.add(symptom)

    print('\n--- Running diagnosis ---')
    final_facts, trace = forward_chain(observed, RULES)

    diseases = ['powdery_mildew', 'root_rot', 'leaf_blight', 'serious_infection']
    found = [d for d in diseases if d in final_facts]

    if found:
        print(f'\nDiagnosis: {', '.join(found)}')
        print('\nExplanation:')
        for _, conclusion, explanation in trace:
            if conclusion in found or conclusion in final_facts:
                print(f'  • {explanation}')
    else:
        print('\nNo disease detected with the given symptoms.')

run_expert_system()
```

Extension

Add a new disease of your choice (e.g. aphid infestation). Define 2–3 new symptoms and write the rule. Test it by running the interactive system.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Machine Learning: Supervised Learning

Linear Regression, Decision Trees, k-NN, SVM

Lab - 7

Machine Learning: Supervised Learning

Linear Regression, Decision Trees, k-NN, SVM

Lab - 7

Machine Learning: Supervised Learning

Linear Regression, Decision Trees, k-NN, SVM

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Load and explore a real dataset using pandas.
- Train four classifiers: Decision Tree, k-NN, SVM, Logistic Regression.
- Evaluate using accuracy, confusion matrix, and classification report.
- Compare classifiers and explain why they differ on the same data.

2. Theory

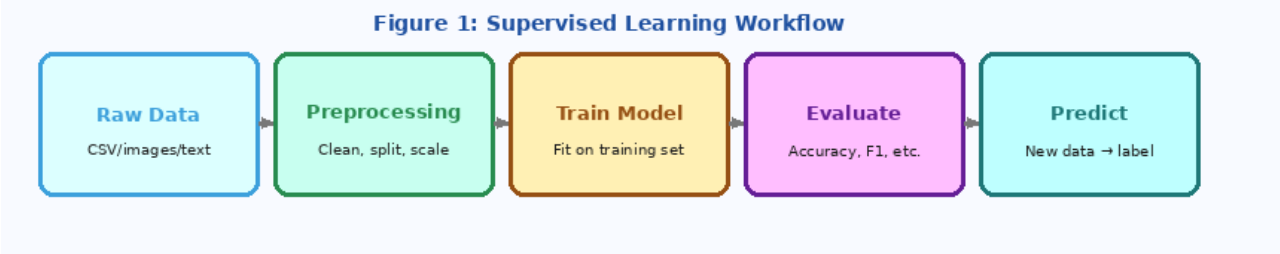


Figure 5 — Supervised Learning Workflow. Every lab exercise follows these five steps.

Key rule: split your data. Never train and test on the same rows — this gives falsely high accuracy. We use an 80/20 train/test split.

3. Lab

Setup & Data Exploration

The Iris dataset has 150 flower samples across 3 species. Each sample has 4 measurements (sepal/petal length and width). This is the 'Hello World' of ML.

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from sklearn.preprocessing import StandardScaler


# Load the dataset
iris = load_iris()
X = iris.data            # features: 4 measurements
y = iris.target          # labels: 0=setosa, 1=versicolor, 2=virginica
feature_names = iris.feature_names
target_names  = iris.target_names

# Quick exploration
df = pd.DataFrame(X, columns=feature_names)
df['species'] = [target_names[i] for i in y]

print('Dataset shape:', X.shape)
print('Classes:', target_names)
print('\nFirst 5 rows:')
print(df.head())
print('\nBasic statistics:')
print(df.describe())

# Split: 80% train, 20% test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
print(f'\nTraining samples: {len(X_train)}')
print(f'Testing samples:  {len(X_test)}')
```

🔍 **Student Task 1**

Run the exploration code. How many samples per class? Which feature has the largest range? Print df['species'].value_counts() to check class balance.

Exercise 1: Decision Tree

A decision tree splits the data on the feature that best separates the classes (using Gini impurity or information gain). It's easy to visualize and explain.

```
from sklearn.tree import DecisionTreeClassifier, export_text

# Train
dt = DecisionTreeClassifier(max_depth=3, random_state=42)
dt.fit(X_train, y_train)

# Evaluate
y_pred_dt = dt.predict(X_test)
acc_dt = accuracy_score(y_test, y_pred_dt)
print(f'Decision Tree Accuracy: {acc_dt:.3f}')

# Visualize the tree structure
tree_text = export_text(dt, feature_names=feature_names)
print('\nTree structure:')
print(tree_text)

# Feature importance
print('Feature importances:')
for name, imp in zip(feature_names, dt.feature_importances_):
    bar = '#' * int(imp * 30)
    print(f'  {name:30} {imp:.3f} {bar}')
```

🔍 **Student Task 2**

Change max_depth to 1, 2, 5, None (unlimited). Record accuracy for each. What happens when the tree is too deep? What is this problem called?

Exercise 2: k-Nearest Neighbors

k-NN classifies a point by voting among its k closest training examples. It stores all training data and makes no assumptions about the data shape.

```
from sklearn.neighbors import KNeighborsClassifier

# Scale features — IMPORTANT for k-NN (distances are affected by scale)
scaler  = StandardScaler()
X_train_s = scaler.fit_transform(X_train)  # fit on train only!
X_test_s  = scaler.transform(X_test)        # apply same scale to test

# Train with k=3
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train_s, y_train)

y_pred_knn = knn.predict(X_test_s)
acc_knn = accuracy_score(y_test, y_pred_knn)
print(f'k-NN (k=3) Accuracy: {acc_knn:.3f}')

# Find the best k
print('\nAccuracy vs k:')
for k in [1, 3, 5, 7, 9, 11]:
    knn_k = KNeighborsClassifier(n_neighbors=k)
    knn_k.fit(X_train_s, y_train)
    acc = accuracy_score(y_test, knn_k.predict(X_test_s))
    bar = '#' * int(acc * 30)
    print(f'  k={k:2}: {acc:.3f} {bar}')
```

🔍 **Student Task 3**

What happens to accuracy as k increases? Why does k=1 often overfit? Why does very large k underfit? Find the best k for this dataset.

Exercise 3: SVM and Logistic Regression

```
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# SVM
svm = SVC(kernel='rbf', C=1.0, random_state=42)
svm.fit(X_train_s, y_train)
acc_svm = accuracy_score(y_test, svm.predict(X_test_s))
print(f'SVM Accuracy: {acc_svm:.3f}')

# Logistic Regression
lr = LogisticRegression(max_iter=200, random_state=42)
lr.fit(X_train_s, y_train)
acc_lr = accuracy_score(y_test, lr.predict(X_test_s))
print(f'Logistic Regression Accuracy: {acc_lr:.3f}')
```

Exercise 4: Full Comparison + Confusion Matrix

Now compare all four classifiers on the same test set and visualize where each one makes mistakes.

```
models = {
    'Decision Tree': dt,
    'k-NN (k=5)':    KNeighborsClassifier(5).fit(X_train_s, y_train),
    'SVM':           svm,
    'Log. Regr.':    lr,
}

print('=== Classifier Comparison ===')
print(f'{'Model':20} | {'Accuracy':10} | {'Best at':20}')
print('-' * 58)

best_model, best_acc = None, 0
for name, model in models.items():
    # Use scaled features for models that need it
    X_t = X_test_s if name != 'Decision Tree' else X_test
    preds = model.predict(X_t)
    acc   = accuracy_score(y_test, preds)
    if acc > best_acc: best_acc, best_model = acc, name
    print(f'{name:20} | {acc:.3f}      | (see report below)')

print(f'\nWinner: {best_model} ({best_acc:.3f})')

# Detailed report for best model
print(f'\nDetailed report for {best_model}:')
best = models[best_model]
X_t = X_test_s if best_model != 'Decision Tree' else X_test
print(classification_report(y_test, best.predict(X_t),
                            target_names=target_names))

# Confusion matrix
print('Confusion Matrix:')
cm = confusion_matrix(y_test, best.predict(X_t))
print('           Pred setosa  Pred versic  Pred virgin')
for i, row in enumerate(cm):
    print(f'True {target_names[i]:10}', row)
```

🔍 Student Task 4

Look at the confusion matrix. Which two species get confused with each other? Inspect the dataset — are their feature values similar? This explains the misclassifications.

💡 Bonus Challenge

Try the same classifiers on the Wine dataset (from sklearn.datasets import load_wine). Does the winner change? Why might a different dataset favor a different algorithm?

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Unsupervised Learning & Neural Networks

K-Means, Hierarchical Clustering, Intro to Neural Networks

Lab - 8

Unsupervised Learning & Neural Networks

K-Means, Hierarchical Clustering, Intro to Neural Networks

Lab - 8

Unsupervised Learning & Neural Networks

K-Means, Hierarchical Clustering, Intro to Neural Networks

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Understand the difference between supervised and unsupervised learning.
- Implement k-means clustering from scratch using only NumPy.
- Use hierarchical clustering and interpret a dendrogram.
- Build a neural network with one hidden layer and train it with backpropagation.
- Understand what weights, activations, and epochs mean.

2. Theory

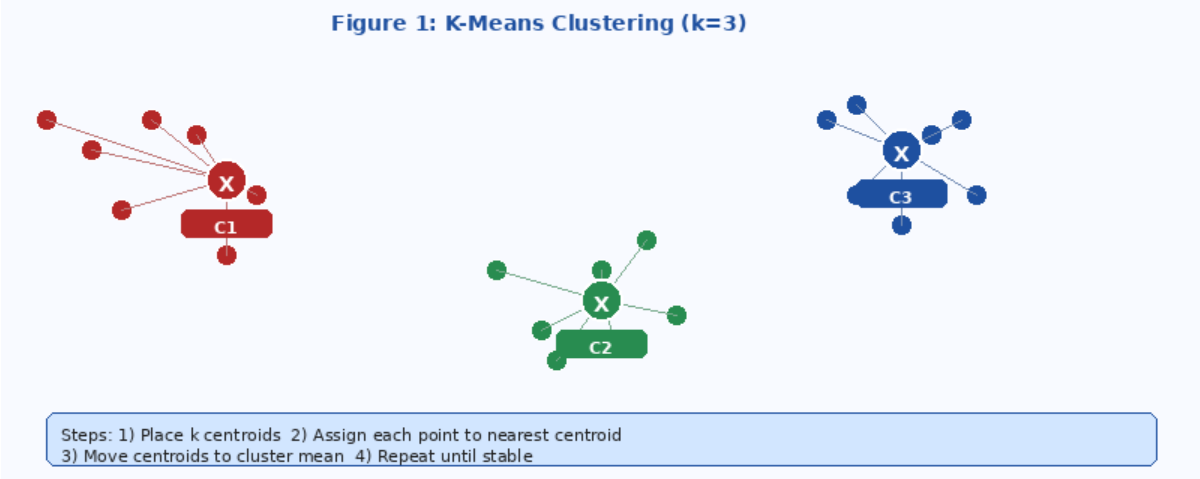


Figure 6 — K-Means Clustering. Three clusters found automatically — no labels were given.

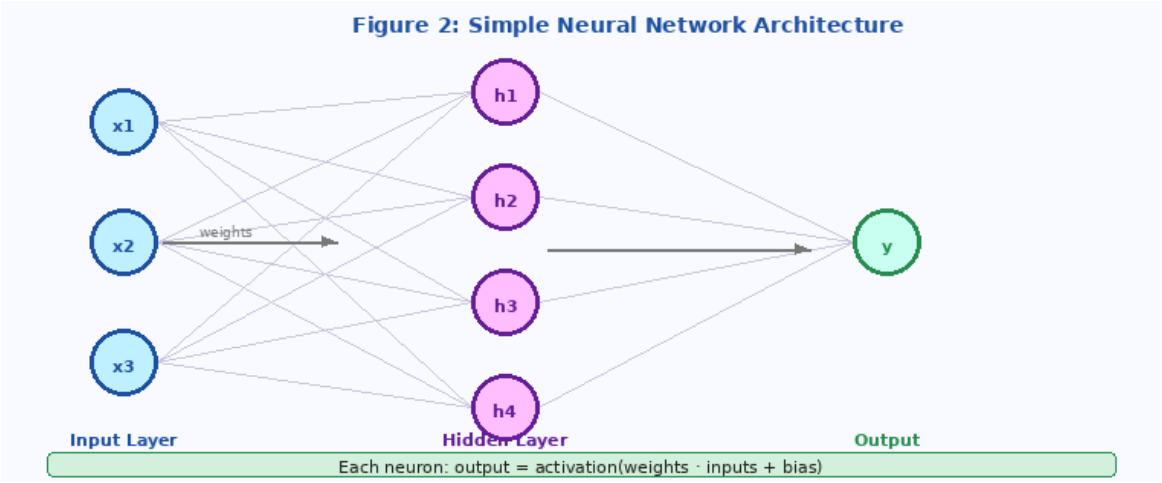


Figure 2 — A simple neural network. Signals flow left to right through layers of neurons.

Term	Meaning
Unsupervised Learning	No labels. The algorithm discovers structure in raw data.
Centroid	The center point of a cluster — updated each iteration in k-means.
Neuron/Node	Computes: output = activation (weights · inputs + bias)
Activation Function	Adds non-linearity. Common choices: sigmoid, ReLU, tanh.
Backpropagation	How a neural net learns: compute error, propagate it backward, update weights.
Epoch	One complete pass through the entire training dataset.

3. Lab

Clustering

Exercise 1: K-Means from Scratch

We implement k-means using only NumPy — no sklearn. This forces you to understand each step of the algorithm.

```
import numpy as np
import matplotlib
matplotlib.use('Agg')   # non-interactive backend
import matplotlib.pyplot as plt

# Generate synthetic 2D data with 3 natural clusters
np.random.seed(42)
cluster1 = np.random.randn(50, 2) + [0, 0]
cluster2 = np.random.randn(50, 2) + [5, 5]
cluster3 = np.random.randn(50, 2) + [0, 8]
X = np.vstack([cluster1, cluster2, cluster3])

def kmeans(X, k, max_iters=100):
    """
    K-Means from scratch.
    Returns: labels (which cluster each point belongs to),
             centroids (final cluster centers),
             history (centroids at each step for visualization)
    """
    # Step 1: Initialize centroids — pick k random points from X
    idx = np.random.choice(len(X), k, replace=False)
    centroids = X[idx].copy()
    history   = [centroids.copy()]

    for iteration in range(max_iters):
        # Step 2: Assign each point to the nearest centroid
        distances = np.array([
            np.linalg.norm(X - c, axis=1) for c in centroids
        ])                              # shape: (k, n_points)
        labels = np.argmin(distances, axis=0)

        # Step 3: Move each centroid to the mean of its cluster
        new_centroids = np.array([
            X[labels == i].mean(axis=0) if np.any(labels == i) else centroids[i]
            for i in range(k)
        ])

        history.append(new_centroids.copy())

        # Step 4: Stop if centroids didn't move
        if np.allclose(centroids, new_centroids):
            print(f'Converged after {iteration+1} iterations')
            break
        centroids = new_centroids
```

```
    return labels, centroids, history

labels, centroids, history = kmeans(X, k=3)
print(f'Cluster sizes: {np.bincount(labels)}')
print(f'Final centroids:\n{centroids.round(2)}')
```

> 🔍 **Student Task 1**
>
> **Run k-means with k=2, 3, 4, 5. For each k, count the cluster sizes. Which k makes sense for this data? What goes wrong with k=5?**

Exercise 2: Inertia and Choosing k (Elbow Method)

```
def compute_inertia(X, labels, centroids):
    """Sum of squared distances from each point to its centroid"""
    total = 0
    for i, c in enumerate(centroids):
        cluster_points = X[labels == i]
        total += np.sum((cluster_points - c) ** 2)
    return total

print('Elbow Method — Inertia vs k:')
print(f'{'k':4} | {'Inertia':12} | Bar chart')
print('-' * 50)

inertias = []
for k in range(1, 9):
    labels_k, centroids_k, _ = kmeans(X, k)
    inertia = compute_inertia(X, labels_k, centroids_k)
    inertias.append(inertia)
    bar = '#' * int(inertia / 500)
    print(f'{k:4} | {inertia:12.1f} | {bar}')

print('\nLook for the ELBOW — where inertia stops dropping sharply.')
print('That k is usually the right number of clusters.')
```

> 🔍 **Student Task 2**
>
> **Which k is the elbow? Does it match the 3 clusters we built into the data? Try generating data with 4 or 5 clusters and see if the elbow shifts.**

Exercise 3: Hierarchical Clustering

Hierarchical clustering builds a tree (dendrogram) — you don't need to choose k in advance. You cut the tree at the desired level.

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import load_iris

iris = load_iris()
X_iris = iris.data
y_iris = iris.target

# Try different numbers of clusters
for n_clusters in [2, 3, 4]:
    agg = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
    pred = agg.fit_predict(X_iris)

    # How well do clusters match true species?
    # (We cheat and compare to known labels to evaluate)
    from sklearn.metrics import adjusted_rand_score
    ari = adjusted_rand_score(y_iris, pred)
    print(f'n_clusters={n_clusters}: ARI={ari:.3f}')

# ARI = 1.0 means perfect match, 0 means random
print('\nBest k is the one with highest ARI (closer to 1.0)')
```

Neural Networks

Exercise 4: Neural Network from Scratch

We build a 2-layer neural network using only NumPy. The network learns the XOR function — a classic problem that linear models cannot solve.

```
import numpy as np

# XOR: output is 1 only when inputs differ
X_xor = np.array([[0,0],[0,1],[1,0],[1,1]])
y_xor = np.array([[0],[1],[1],[0]])

def sigmoid(z):       return 1 / (1 + np.exp(-z))
def sigmoid_d(z):     return sigmoid(z) * (1 - sigmoid(z))

class SimpleNN:
    def __init__(self, input_size, hidden_size, output_size, lr=0.1):
        # Random weight initialization
        np.random.seed(42)
        self.W1 = np.random.randn(input_size, hidden_size) * 0.5
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.5
        self.b2 = np.zeros((1, output_size))
        self.lr = lr

    def forward(self, X):
        """Forward pass: compute predictions"""
        self.z1 = X  @ self.W1 + self.b1
```

```
        self.a1 = sigmoid(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = sigmoid(self.z2)
        return self.a2

    def backward(self, X, y):
        """Backpropagation: compute gradients and update weights"""
        m = X.shape[0]

        # Output layer error
        dz2 = self.a2 - y
        dW2 = self.a1.T @ dz2 / m
        db2 = dz2.mean(axis=0, keepdims=True)

        # Hidden layer error
        dz1 = (dz2 @ self.W2.T) * sigmoid_d(self.z1)
        dW1 = X.T @ dz1 / m
        db1 = dz1.mean(axis=0, keepdims=True)

        # Update weights
        self.W2 -= self.lr * dW2;  self.b2 -= self.lr * db2
        self.W1 -= self.lr * dW1;  self.b1 -= self.lr * db1

    def loss(self, y_pred, y_true):
        return np.mean((y_pred - y_true) ** 2)  # MSE
```

```
# --- Train the network ---
nn = SimpleNN(input_size=2, hidden_size=4, output_size=1, lr=1.0)

print('Training XOR network...')
print(f'{'Epoch':8} | {'Loss':10}')
print('-' * 22)

for epoch in range(10001):
    y_pred = nn.forward(X_xor)
    nn.backward(X_xor, y_xor)
    if epoch % 1000 == 0:
        l = nn.loss(y_pred, y_xor)
        bar = '#' * int((1-l) * 20)
        print(f'{epoch:8} | {l:.6f}  {bar}')

# --- Test ---
print('\nFinal predictions:')
preds = nn.forward(X_xor)
for i, (inp, pred, true) in enumerate(zip(X_xor, preds, y_xor)):
    result = 'OK' if abs(pred[0] - true[0]) < 0.1 else 'MISS'
    print(f'  Input {inp} → Pred: {pred[0]:.3f}  True: {true[0]}  [{result}]')
```

🔍 Student Task 3

Run the network and watch the loss decrease. Try: hidden_size=2 vs 8, lr=0.1 vs 2.0, epochs=1000 vs 20000. What happens when learning rate is too high? Too low?

Exercise 5: Sklearn Neural Network on Iris

Now use sklearn's MLPClassifier for a real multi-class problem. This uses the same ideas but handles all the details automatically.

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

# MLP = Multi-Layer Perceptron (feedforward neural network)
mlp = MLPClassifier(
    hidden_layer_sizes=(8, 4),   # 2 hidden layers: 8 then 4 neurons
    activation='relu',
    max_iter=500,
    random_state=42
)
mlp.fit(X_train, y_train)

preds = mlp.predict(X_test)
print(f'Neural Network Accuracy: {accuracy_score(y_test, preds):.3f}')
print(f'Training loss (final): {mlp.loss_:.4f}')
print(f'Epochs taken: {mlp.n_iter_}')
print()
print(classification_report(y_test, preds, target_names=iris.target_names))

# Compare with Week 7 classifiers
print('Compare to Week 7 results!')
```

Final Challenge

Try hidden_layer_sizes= (100,50,25) — a deeper network. Does it improve accuracy? Does it need more epochs? This is the start of deep learning.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Probabilistic Reasoning

Probability, Bayes' Theorem, HMMs, Naive Bayes

Lab - 9

Probabilistic Reasoning

Probability, Bayes' Theorem, HMMs, Naive Bayes

Lab - 9

Probabilistic Reasoning

Probability, Bayes' Theorem, HMMs, Naive Bayes

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Apply Bayes' theorem to compute posterior probabilities by hand.
- Understand the 'naive' independence assumption and why it works in practice.
- Implement Naive Bayes training (probability tables) from a text dataset.
- Implement Naive Bayes prediction with Laplace smoothing.
- Evaluate the classifier using accuracy and a confusion matrix.
- Understand Hidden Markov Models at a conceptual level.

2. Theory

Bayes' Theorem

Bayes' theorem tells us how to update our belief about an event (A) after seeing new evidence (B). It is the mathematical foundation of probabilistic AI.

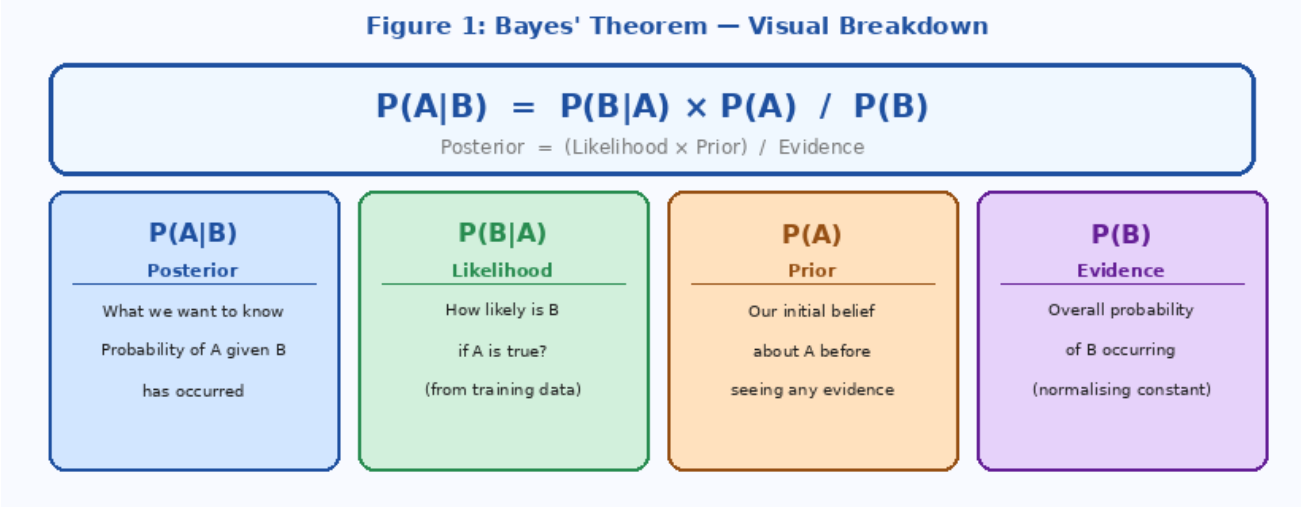


Figure 7 — Bayes' theorem and what each term means.

Worked example: A spam filter. We want to know P (spam | contains 'free'). From training data, we know: 80% of spam emails contain 'free', 5% of non-spam do. 30% of all emails are spam.

```
# P(spam | 'free')  =  P('free' | spam) × P(spam)  /  P('free')
#
# P('free') = P('free'|spam)×P(spam) + P('free'|ham)×P(ham)
#           = 0.80 × 0.30 + 0.05 × 0.70
#           = 0.24 + 0.035 = 0.275
#
# P(spam | 'free') = 0.80 × 0.30 / 0.275 = 0.873
#
# Conclusion: If an email contains 'free', there is an 87% chance it is spam.

p_free_given_spam = 0.80
p_free_given_ham  = 0.05
p_spam            = 0.30
p_ham             = 1 - p_spam

p_free  = p_free_given_spam * p_spam + p_free_given_ham * p_ham
p_spam_given_free = (p_free_given_spam * p_spam) / p_free

print(f'P(spam | free) = {p_spam_given_free:.3f}')    # 0.873
```

Naive Bayes for Text

When classifying a document, we have many words. Naive Bayes multiplies the individual word probabilities together — that is the 'naive' independence assumption. Despite being wrong in theory, it works surprisingly well in practice.

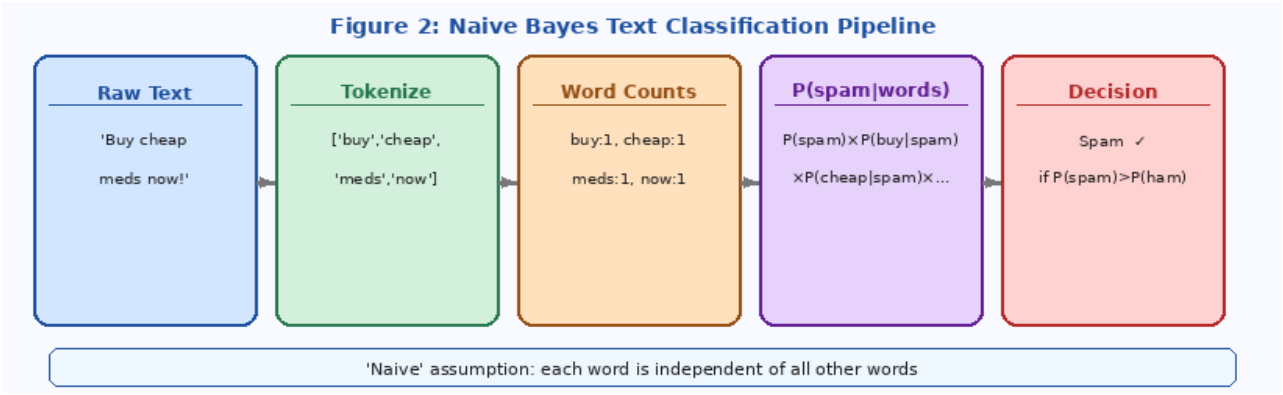


Figure 2 — Naive Bayes pipeline. Each word contributes independently to the final score.

Hidden Markov Models (HMM) — Concept

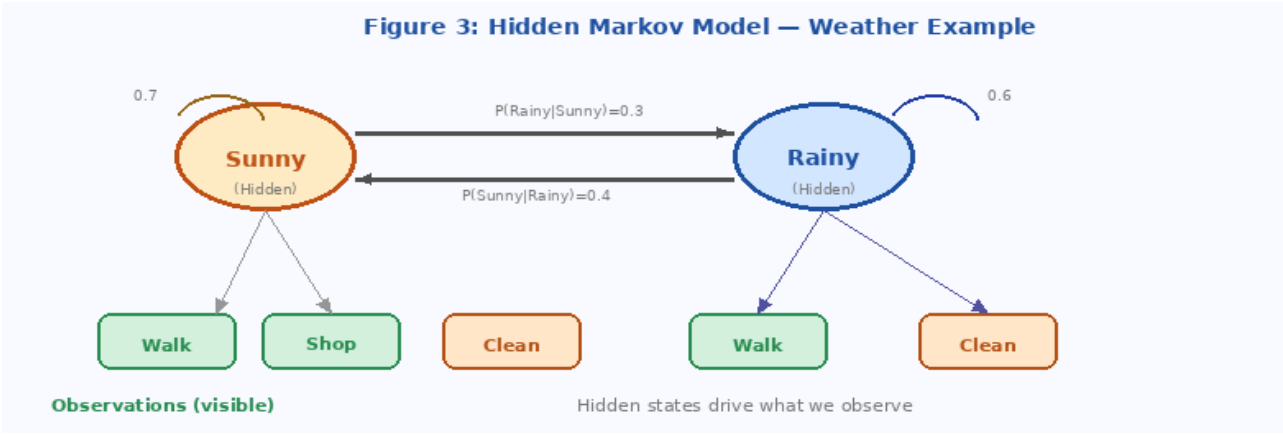


Figure 3 — HMM for weather. Hidden states (Sunny/Rainy) generate observable activities.

In an HMM, we observe a sequence of outputs (e.g. what someone does each day) but cannot directly see the underlying states (e.g. the weather). The Viterbi algorithm finds the most likely sequence of hidden states. HMMs are used in speech recognition, gesture recognition, and bioinformatics.

3. Lab

The Dataset

We use a small labelled dataset of email subjects. Each is labelled 'spam' or 'ham' (legitimate). In a real project you would load thousands of examples from a CSV.

```
# Our small training corpus
TRAINING_DATA = [
    # (text, label)
    ('win free money now click here',          'spam'),
    ('congratulations you won a prize',         'spam'),
    ('buy cheap meds online free shipping',     'spam'),
    ('limited offer claim your free gift',      'spam'),
    ('make money fast guaranteed results',      'spam'),
    ('urgent your account needs attention',     'spam'),
    ('dear friend I need your help transfer',   'spam'),

    ('meeting tomorrow at 10am in room 3',      'ham'),
    ('please review the attached report',        'ham'),
    ('see you at the conference next week',      'ham'),
    ('lunch at noon in the cafeteria',           'ham'),
    ('project deadline moved to Friday',         'ham'),
    ('team standup call at 9am tomorrow',        'ham'),
    ('feedback on your presentation was great',  'ham'),
]

TEST_DATA = [
    ('free money transfer click now',            'spam'),
    ('meeting cancelled see you next week',      'ham'),
    ('win a free gift click here',               'spam'),
    ('project update attached for review',       'ham'),
    ('urgent claim your prize now',              'spam'),
]
```

Exercise 1: Training the Classifier

Training means counting words per class, then converting those counts into probabilities. We use Laplace smoothing to avoid zero probabilities for unseen words.

```
from collections import defaultdict, Counter
import math

class NaiveBayesClassifier:
    def __init__(self):
        self.class_counts   = Counter()          # how many docs per class
        self.word_counts    = defaultdict(Counter)  # word counts per class
        self.vocabulary     = set()              # all unique words
        self.total_docs     = 0

    def tokenize(self, text):
        """Lowercase and split into words"""
        return text.lower().split()

    def train(self, data):
        """
```

```
        Count everything we need for Bayes:
        - How many documents per class  (for prior P(class))
        - How many times each word appears in each class  (for likelihood)
        """
        for text, label in data:
            words = self.tokenize(text)
            self.class_counts[label] += 1
            self.total_docs          += 1
            for word in words:
                self.word_counts[label][word] += 1
                self.vocabulary.add(word)

        print(f'Training complete:')
        print(f'  Documents: {self.total_docs}')
        print(f'  Classes:   {dict(self.class_counts)}')
        print(f'  Vocabulary size: {len(self.vocabulary)} words')

        # Show top spam words
        top_spam = self.word_counts['spam'].most_common(5)
        print(f'  Top spam words: {top_spam}')

    def predict(self, text):
        """
        For each class, compute log P(class) + sum of log P(word|class).
        We use log-probabilities to avoid underflow (multiplying
        many tiny numbers gives 0 in floating point).
        """
        words = self.tokenize(text)
        V = len(self.vocabulary)    # vocabulary size for Laplace smoothing
        scores = {}

        for label in self.class_counts:
            # Prior: log P(class)
            log_score = math.log(self.class_counts[label] / self.total_docs)

            # Likelihood: log P(word | class) for each word
            total_words_in_class = sum(self.word_counts[label].values())
            for word in words:
                word_count = self.word_counts[label][word]
                # Laplace smoothing: add 1 to numerator, V to denominator
                # This prevents log(0) for words not seen in training
                p_word = (word_count + 1) / (total_words_in_class + V)
                log_score += math.log(p_word)

            scores[label] = log_score

        # Return the class with the highest log-probability
        return max(scores, key=scores.get), scores

    def evaluate(self, test_data):
        correct = 0
        results = []
        for text, true_label in test_data:
            pred_label, scores = self.predict(text)
            correct += (pred_label == true_label)
            results.append((text, true_label, pred_label,
                            'OK' if pred_label == true_label else 'WRONG'))
        return correct / len(test_data), results
```

Exercise 2: Run and Evaluate

```
# --- Train ---
clf = NaiveBayesClassifier()
clf.train(TRAINING_DATA)

# --- Test on individual examples ---
print('\n=== Predictions ===')
test_sentences = [
    'win free money now',
    'see you at the meeting tomorrow',
    'free gift claim your prize',
    'please review the attached document',
    'guaranteed results buy now',
]

for sentence in test_sentences:
    label, scores = clf.predict(sentence)
    spam_score = scores.get('spam', -999)
    ham_score  = scores.get('ham',  -999)
    confidence = 'HIGH' if abs(spam_score - ham_score) > 3 else 'LOW'
    print(f'  "{sentence}"')
    print(f'    → {label.upper()} (confidence: {confidence})')

# --- Full evaluation ---
print('\n=== Evaluation on Test Set ===')
accuracy, results = clf.evaluate(TEST_DATA)

print(f'{'Text':45} | {'True':5} | {'Pred':5} | Status')
print('-' * 70)
for text, true_l, pred_l, status in results:
    short = text[:42] + '..' if len(text) > 42 else text
    print(f'{short:45} | {true_l:5} | {pred_l:5} | {status}')

print(f'\nAccuracy: {accuracy:.1%}
({int(accuracy*len(TEST_DATA))}/{len(TEST_DATA)} correct)')
```

> 🔍 **Student Task 1**
> **Run the classifier. Which test emails are misclassified? Look at their words — why does the model get them wrong? Try adding more training examples that fix the mistakes.**

Exercise 3: Inspect the Model

One of Naive Bayes' big advantages is transparency. You can inspect exactly what the model has learned.

```
def inspect_model(clf, word):
    """Show how a word influences the classification"""
    V = len(clf.vocabulary)
    print(f'\nWord: "{word}"')
    for label in clf.class_counts:
        count = clf.word_counts[label][word]
        total = sum(clf.word_counts[label].values())
        p = (count + 1) / (total + V)
        print(f'  P("{word}" | {label}) = ({count}+1)/({total}+{V}) = {p:.4f}')

inspect_model(clf, 'free')
inspect_model(clf, 'meeting')
inspect_model(clf, 'click')
inspect_model(clf, 'review')

# Print top 5 most 'spammy' and most 'hammy' words
print('\nMost indicative spam words:')
spam_words = clf.word_counts['spam'].most_common(8)
for word, count in spam_words:
    print(f'  {word}: appeared {count} times in spam')

print('\nMost indicative ham words:')
ham_words = clf.word_counts['ham'].most_common(8)
for word, count in ham_words:
    print(f'  {word}: appeared {count} times in ham')
```

🔍 **Student Task 2**

What is Laplace smoothing? Remove the +1 and +V from the predict () method (set them to 0). What happens when you classify a sentence with a word that never appeared in training?

Exercise 4: Confusion Matrix & Metrics

```
def confusion_matrix(clf, data):
    """
    Build and display a confusion matrix:
        True Positive  (TP): predicted spam, actually spam
        True Negative  (TN): predicted ham,  actually ham
        False Positive (FP): predicted spam, actually ham  (bad!)
        False Negative (FN): predicted ham,  actually spam (bad!)
    """
    tp = tn = fp = fn = 0
    for text, true_label in data:
        pred_label, _ = clf.predict(text)
        if pred_label == 'spam' and true_label == 'spam': tp += 1
        elif pred_label == 'ham'  and true_label == 'ham':  tn += 1
        elif pred_label == 'spam' and true_label == 'ham':  fp += 1
        else:                                               fn += 1

    print('\nConfusion Matrix:')
    print(f'                Predicted Spam  Predicted Ham')
    print(f'  Actual Spam        {tp:3}             {fn:3}')
    print(f'  Actual Ham         {fp:3}             {tn:3}')

    precision = tp / (tp + fp) if (tp+fp) > 0 else 0
    recall    = tp / (tp + fn) if (tp+fn) > 0 else 0
    f1        = 2*precision*recall/(precision+recall) if (precision+recall)>0 else
0

    print(f'\nPrecision: {precision:.2f}  (of predicted spam, how many were really
spam)')
    print(f'Recall:    {recall:.2f}  (of actual spam, how many did we catch)')
    print(f'F1 Score:  {f1:.2f}  (harmonic mean of precision and recall)')

confusion_matrix(clf, TEST_DATA)
```

Bonus Challenge

Extend the classifier to handle a third class: 'phishing' emails (contain: 'verify account', 'click link', 'password expired'). Add 5 training examples for this class and re-evaluate.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Natural Language Processing

Text Processing, Sentiment Analysis, Chatbots

Lab - 10

Natural Language Processing

Text Processing, Sentiment Analysis, Chatbots

Lab - 10

Natural Language Processing

Text Processing, Sentiment Analysis, Chatbots

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Build a full text preprocessing pipeline (tokenize, normalize, remove stopwords, stem).
- Implement TF-IDF weighting from scratch to measure word importance.
- Build a rule-based sentiment analyzer using a lexicon.
- Build an intent-matching chatbot with context memory.
- Understand the difference between rule-based and ML-based NLP.

2. Theory

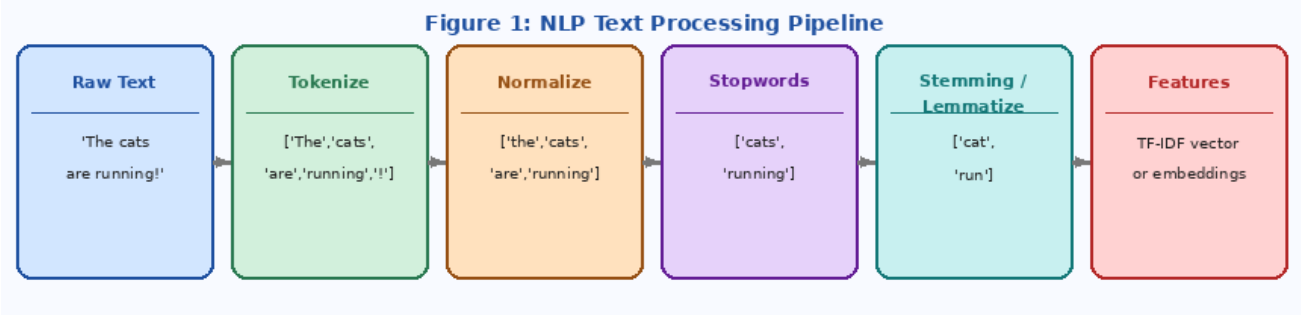


Figure 8 — Text preprocessing pipeline. Every NLP task starts with these steps.

Step	What it does	Example
Tokenization	Split text into individual tokens (words/punctuation)	'Hello world!' → ['Hello','world','!']
Normalization	Lowercase, remove punctuation	'Hello World!' → 'hello world'
Stop-word removal	Remove very common words that carry little meaning	'the cats are running' → 'cats running'
Stemming	Cut words to their root form (rough)	'running' → 'run', 'cats' → 'cat'
Lemmatization	Reduce to dictionary base form (precise)	'better' → 'good', 'was' → 'be'
TF-IDF	Score words by how unique they are to a document	'AI' in AI paper scores high; 'the' scores near 0

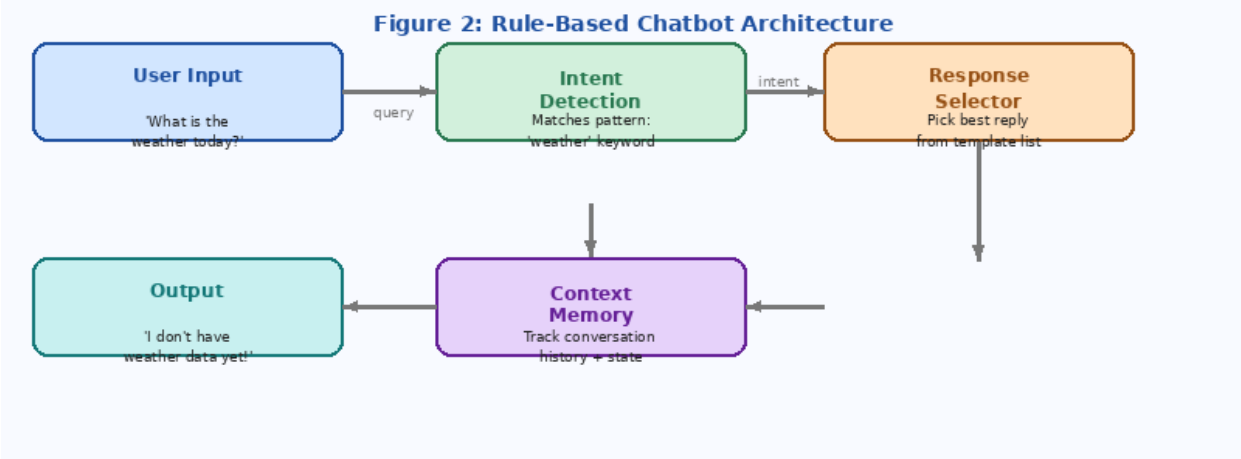


Figure 2 — Chatbot architecture. Intent detection maps user input to a response category

3. Lab

Exercise 1: Text Preprocessing Pipeline

We build the pipeline from Figure 1, step by step. Every later exercise depends on this.

```
import re
import math
from collections import Counter, defaultdict

# ── Step 1: Tokenize ──
def tokenize(text):
    """Lowercase and split into word tokens, stripping punctuation"""
    text = text.lower()
    text = re.sub(r'[^a-z0-9\s]', '', text)   # keep only letters, digits, spaces
    return text.split()

# ── Step 2: Stop-word removal ──
STOPWORDS = {
    'a','an','the','is','are','was','were','be','been','being',
    'have','has','had','do','does','did','will','would','could',
    'should','may','might','shall','can','i','you','he','she','we',
    'they','it','this','that','these','those','and','or','but','in',
    'on','at','to','for','of','with','by','from','up','as','into',
}

def remove_stopwords(tokens):
    return [t for t in tokens if t not in STOPWORDS]

# ── Step 3: Simple stemmer ──
def simple_stem(word):
    """Trim common suffixes — a crude but effective stemmer"""
    for suffix in ['ing', 'tion', 'ness', 'ment', 'able', 'ful', 'ed', 'er', 'ly',
's']:
        if word.endswith(suffix) and len(word) - len(suffix) > 2:
            return word[:-len(suffix)]
    return word

# ── Full pipeline ──
def preprocess(text, stem=True):
    tokens = tokenize(text)
    tokens = remove_stopwords(tokens)
    if stem:
        tokens = [simple_stem(t) for t in tokens]
    return tokens

# --- Test the pipeline ---
samples = [
    'The cats are running very quickly in the garden',
    'Natural Language Processing is a fascinating field of AI',
    'I am planning to visit the conference next week',
]
for s in samples:
    print(f'Input:  {s}')
    print(f'Output: {preprocess(s)}')
    print()
```

🔍 **Student Task 1**

Run the pipeline on three sentences of your own. Does the stemmer always produce real words? What is the difference between stemming and lemmatization? (Hint: 'better' → stem='bett', lemma='good')

Exercise 2: TF-IDF from Scratch

TF-IDF (Term Frequency – Inverse Document Frequency) scores how important a word is to a specific document. Common words score low; rare words that appear a lot in one document score high.

```
def compute_tfidf(corpus):
    """
    corpus = list of strings (documents)
    Returns: list of dicts  {word: tfidf_score}  — one per document
    """
    N = len(corpus)

    # Tokenize all documents
    tokenized = [preprocess(doc, stem=False) for doc in corpus]

    # ── IDF: how rare is each word across all documents? ──
    # IDF = log(N / (number of documents containing the word))
    doc_freq = Counter()
    for doc_tokens in tokenized:
        for word in set(doc_tokens):    # set: count each word once per doc
            doc_freq[word] += 1

    idf = {word: math.log(N / df) for word, df in doc_freq.items()}

    # ── TF: how often does each word appear in THIS document? ──
    results = []
    for doc_tokens in tokenized:
        tf = Counter(doc_tokens)
        total_words = len(doc_tokens)
        tfidf = {}
        for word, count in tf.items():
            tf_score   = count / total_words       # normalised frequency
            tfidf[word] = tf_score * idf.get(word, 0)
        results.append(tfidf)

    return results

# Test corpus
corpus = [
    'machine learning is a subset of artificial intelligence',
    'deep learning uses neural networks with many layers',
    'natural language processing helps computers understand text',
    'computer vision allows machines to interpret images',
    'reinforcement learning trains agents using rewards and penalties',
]

tfidf_scores = compute_tfidf(corpus)
```

```
print('Top 3 keywords per document (by TF-IDF):')
for i, (doc, scores) in enumerate(zip(corpus, tfidf_scores)):
    top = sorted(scores.items(), key=lambda x: x[1], reverse=True)[:3]
    keywords = [f'{w}({s:.3f})' for w, s in top]
    print(f'  Doc {i+1}: {', '.join(keywords)}')
```

> 🔍 **Student Task 2**
> **Why does the word 'learning' get a low TF-IDF score even though it appears often? Add a new document about 'quantum computing' — notice that 'quantum' gets a high score immediately. Why?**

Exercise 3: Sentiment Analyzer

We score sentiment by looking up words in a lexicon of positive and negative words. This is the same approach used in early social media analysis tools.

```
POSITIVE_WORDS = {
    'good','great','excellent','amazing','wonderful','fantastic','love',
    'best','happy','enjoy','perfect','awesome','beautiful','brilliant',
    'helpful','recommend','fast','easy','friendly','clean','clear',
    'useful','efficient','pleased','impressive','outstanding','fun',
}

NEGATIVE_WORDS = {
    'bad','terrible','awful','horrible','worst','hate','disappointing',
    'poor','broken','slow','difficult','confusing','useless','waste',
    'boring','ugly','rude','expensive','error','failed','problem',
    'annoying','frustrating','unreliable','overpriced','misleading',
}

INTENSIFIERS = {'very':1.5, 'extremely':2.0, 'really':1.3, 'quite':1.2,
                'somewhat':0.7, 'barely':0.4, 'not':-1.0}

def analyse_sentiment(text):
    tokens  = tokenize(text)
    score   = 0.0
    multiplier = 1.0
    detail  = []

    for i, token in enumerate(tokens):
        if token in INTENSIFIERS:
            multiplier = INTENSIFIERS[token]
            continue
        elif token in POSITIVE_WORDS:
            contribution = 1.0 * multiplier
            score += contribution
            detail.append(f'+{contribution:.1f}({token})')
        elif token in NEGATIVE_WORDS:
            contribution = -1.0 * multiplier
            score += contribution
            detail.append(f'{contribution:.1f}({token})')
        multiplier = 1.0   # reset after each word
```

```
    if   score >  0.5: label = 'POSITIVE'
    elif score < -0.5: label = 'NEGATIVE'
    else:              label = 'NEUTRAL'

    return label, score, detail

reviews = [
    'This product is very good and extremely helpful',
    'Terrible customer service, very slow and really frustrating',
    'It is okay, not great but not bad either',
    'Amazing quality, I love this, totally recommend it',
    'Completely broken, worst purchase ever, do not buy',
]

for review in reviews:
    label, score, detail = analyse_sentiment(review)
    print(f'{label:9} ({score:+.1f})  {" ".join(detail)}')
    print(f'  → "{review}"')
```

> 🔍 **Student Task 3**
> **Try the sentence: 'not good at all'. Does it classify correctly? Trace through the code — what happens with the word 'not' and the INTENSIFIERS multiplier? How could you improve negation handling?**

Exercise 4: Intent-Matching Chatbot

Now we combine the preprocessing pipeline with pattern matching and context memory to build a chatbot that can hold a basic conversation.

```
import random

# ── Intent patterns and responses ──
INTENTS = [
    {
        'name': 'greeting',
        'keywords': ['hello','hi','hey','good morning','good afternoon'],
        'responses': ['Hello! How can I help you today?',
                      'Hi there! What can I do for you?',
                      'Hey! Great to see you. How can I assist?'],
    },
    {
        'name': 'farewell',
        'keywords': ['bye','goodbye','see you','take care','exit','quit'],
        'responses': ['Goodbye! Have a great day!',
                      'See you later!', 'Take care!'],
        'end_chat': True,
    },
    {
        'name': 'about_ai',
        'keywords': ['artificial intelligence','ai','machine learning','deep
learning'],
```

```
        'responses': ['AI is the field of making machines intelligent.',
                      'Machine learning lets systems learn from data.',
                      'Deep learning uses neural networks with many layers.'],
    },
    {
        'name': 'sentiment_check',
        'keywords':
['feel','feeling','mood','happy','sad','frustrated','excited'],
        'responses': ['I understand how you feel.',
                      "I'm here to help. Tell me more.",
                      'Your feelings are valid. How can I help?'],
    },
    {
        'name': 'help',
        'keywords': ['help','assist','support','problem','issue','question'],
        'responses': ['I am here to help! Tell me your question.',
                      'Sure, what do you need help with?'],
    },
    {
        'name': 'weather',
        'keywords': ['weather','rain','sunny','temperature','forecast'],
        'responses': ['I do not have access to live weather data.',
                      'For weather, try a weather app or website.'],
    },
]

class Chatbot:
    def __init__(self, name='AI Assistant'):
        self.name    = name
        self.context = []     # conversation history
        self.turn    = 0

    def detect_intent(self, text):
        """Find the intent whose keywords best match the user's input"""
        tokens = set(preprocess(text, stem=False))
        best_intent = None
        best_score  = 0

        for intent in INTENTS:
            # Count how many keywords from this intent appear in the input
            score = sum(1 for kw in intent['keywords']
                        if any(kw in text.lower() for kw in intent['keywords']))
            # Better: check individual words
            score = sum(1 for kw in intent['keywords']
                        if kw in text.lower())
            if score > best_score:
                best_score  = score
                best_intent = intent

        return best_intent, best_score

    def respond(self, user_input):
        self.turn += 1
        self.context.append(('user', user_input))

        # Detect sentiment for empathetic response
        sentiment, score, _ = analyse_sentiment(user_input)

        # Detect intent
```

```
        intent, confidence = self.detect_intent(user_input)

        if intent is None or confidence == 0:
            response = "I'm not sure I understand. Could you rephrase that?"
        else:
            response = random.choice(intent['responses'])

        # Add empathy prefix for strong negative sentiment
        if sentiment == 'NEGATIVE' and score < -1.0:
            response = "I'm sorry to hear that. " + response

        self.context.append(('bot', response))
        end = intent.get('end_chat', False) if intent else False
        return response, end

    def chat(self):
        print(f'=== {self.name} ===')
        print('Type your message. Type "bye" to exit.\n')
        while True:
            user_input = input('You: ').strip()
            if not user_input: continue
            response, should_end = self.respond(user_input)
            print(f'Bot: {response}\n')
            if should_end: break

bot = Chatbot('StudyBot')
bot.chat()
```

🔍 Student Task 4

Test the chatbot with at least 10 different inputs. Find two cases where it responds poorly. Add a new intent (e.g. 'exam_help' or 'course_info') with at least 4 keywords and 3 responses.

Exercise 5: Conversation History & Context

```
def show_conversation_analysis(bot):
    """Analyse the conversation after it ends"""
    print('\n=== Conversation Analysis ===')
    print(f'Total turns: {bot.turn}')

    user_inputs = [text for role, text in bot.context if role == 'user']
    all_text    = ' '.join(user_inputs)

    # Overall sentiment of user across conversation
    label, score, _ = analyse_sentiment(all_text)
    print(f'Overall user sentiment: {label} (score: {score:+.1f})')

    # Most frequent words from user
    tokens = preprocess(all_text)
    word_freq = Counter(tokens).most_common(5)
    print(f'User\'s top keywords: {word_freq}')

    # Show the conversation
    print('\nFull transcript:')
    for role, text in bot.context:
        prefix = 'You:' if role == 'user' else 'Bot:'
        print(f'  {prefix:5} {text}')

# Run after chatting:
# show_conversation_analysis(bot)
```

Bonus Challenge

Upgrade the intent detector to use TF-IDF similarity instead of keyword counting. Represent each intent's keywords as a TF-IDF vector and compute cosine similarity against the user's input. Does it handle paraphrasing better?

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Computer Vision & Perception

Image Representation, Edge Detection, CNN Classification

Lab - 11

Computer Vision & Perception

Image Representation, Edge Detection, CNN Classification

Lab - 11

Computer Vision & Perception

Image Representation, Edge Detection, CNN Classification

1. An overview

Learning Objectives

After completing this lab, you should be able to:

- Load and manipulate an image as a NumPy array.
- Convert RGB to grayscale and understand pixel arithmetic.
- Apply convolution manually to detect edges using a Sobel kernel.
- Understand how a CNN learns spatial features layer by layer.
- Use a pre-trained ResNet model (no training required) to classify images.
- Interpret prediction confidence scores (softmax probabilities).

2. Theory

Images as Arrays

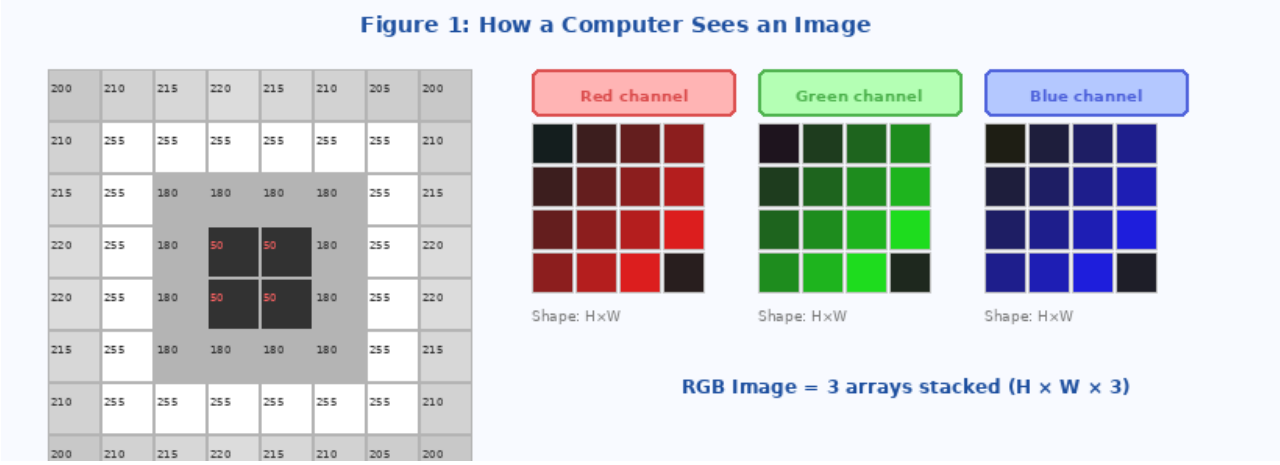


Figure 9 — Every image is just numbers. Grayscale = 1 channel; RGB = 3 channels stacked.

Concept	Details
Pixel	Smallest unit of an image. A number (0–255) representing brightness.
Grayscale image	2D array of shape (H, W). One value per pixel.
RGB image	3D array of shape (H, W, 3). Three channels: Red, Green, Blue.
Convolution	Slide a small filter (kernel) across the image to extract features.
Feature map	The output of a convolution — highlights specific patterns like edges.

Convolutional Neural Networks

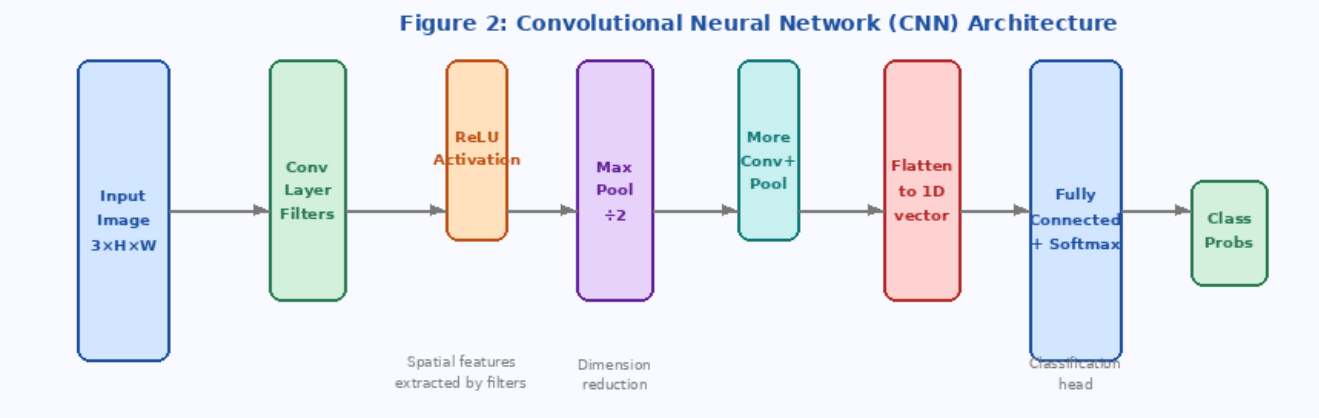


Figure 2 — CNN architecture. Early layers detect edges and textures; deeper layers detect objects.

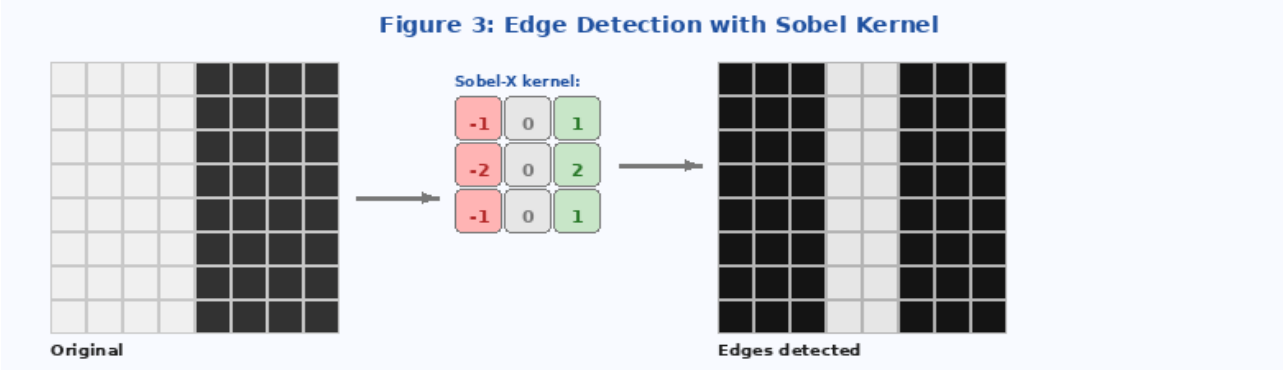


Figure 3 — Edge detection. The Sobel kernel highlights vertical boundaries between dark and light regions.

3. Lab

Exercise 1: Images as NumPy Arrays

Before working with CNNs, we must understand what an image looks like in memory. Every operation in computer vision is just array manipulation.

```
import numpy as np
from PIL import Image
import os

# ── Create a synthetic test image (no file needed) ──
# We'll build a simple 8x8 image with a bright square on dark background
def make_test_image(size=64):
    """Create a simple grayscale test image programmatically"""
    img_array = np.zeros((size, size), dtype=np.uint8)  # all black
    # Draw a white square in the middle
    margin = size // 4
    img_array[margin:size-margin, margin:size-margin] = 200
    # Draw a darker square inside that
    inner = size // 3
    img_array[inner:size-inner, inner:size-inner] = 80
    return img_array

gray = make_test_image(64)

print('=== Image as Array ===')
print(f'Shape:   {gray.shape}   (height x width)')
print(f'Dtype:   {gray.dtype}')
print(f'Min pixel: {gray.min()}  Max pixel: {gray.max()}')
print(f'Mean pixel: {gray.mean():.1f}')
print(f'\nTop-left 8x8 corner:')
print(gray[:8, :8])

# ── Basic pixel operations ──
# Invert: make dark pixels bright and vice versa
inverted = 255 - gray

# Threshold: make everything above 128 white, below 128 black
binary = (gray > 128).astype(np.uint8) * 255

# Crop: take only the top-left quadrant
crop = gray[:32, :32]

print(f'\nOriginal shape: {gray.shape}')
print(f'Cropped shape:  {crop.shape}')
print(f'Binary unique values: {np.unique(binary)}')

# ── Create and inspect an RGB image ──
rgb = np.zeros((64, 64, 3), dtype=np.uint8)
rgb[:, :32, 0] = 200   # left half: red channel
rgb[:, 32:, 2] = 200   # right half: blue channel

print(f'\nRGB image shape: {rgb.shape}  (H x W x channels)')
print(f'Pixel at (32,16): {rgb[32, 16]}   (R,G,B)')
print(f'Pixel at (32,48): {rgb[32, 48]}   (R,G,B)')
```

🔍 **Student Task 1**

Modify make_test_image to draw a diagonal stripe. Then: what is the shape of rgb[:,:,0]? What does it contain? Print rgb[:,:,0].max() — what does this tell you about the red channel?

Exercise 2: Manual Convolution and Edge Detection

A convolution slides a small matrix (kernel) over the image and computes a weighted sum at each position. This is the fundamental operation in every CNN layer.

```
def convolve2d(image, kernel):
    """
    Apply a 2D convolution manually.
    image:  2D NumPy array (grayscale)
    kernel: 2D NumPy array (e.g. 3x3)
    Returns: output feature map
    """
    H, W    = image.shape
    kH, kW  = kernel.shape
    pad_h   = kH // 2
    pad_w   = kW // 2

    # Pad the image so output has same size
    padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='edge')
    output = np.zeros_like(image, dtype=np.float32)

    for r in range(H):
        for c in range(W):
            # Extract the patch under the kernel
            patch = padded[r:r+kH, c:c+kW]
            # Element-wise multiply and sum
            output[r, c] = np.sum(patch * kernel)

    return output

# ── Sobel kernels ──
# Sobel-X detects vertical edges (changes left-to-right)
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]], dtype=np.float32)

# Sobel-Y detects horizontal edges (changes top-to-bottom)
sobel_y = np.array([[-1,-2,-1],
                    [ 0, 0, 0],
                    [ 1, 2, 1]], dtype=np.float32)

# ── Blur kernel (smoothing) ──
blur = np.ones((3,3), dtype=np.float32) / 9

# Apply to our test image
gray_float = gray.astype(np.float32)
```

```
edges_x  = convolve2d(gray_float, sobel_x)
edges_y  = convolve2d(gray_float, sobel_y)
edges    = np.sqrt(edges_x**2 + edges_y**2)  # combined edge magnitude
blurred  = convolve2d(gray_float, blur)

print('=== Convolution Results ===')
print(f'Original range:    {gray_float.min():.0f} to {gray_float.max():.0f}')
print(f'Edges-X range:     {edges_x.min():.0f} to {edges_x.max():.0f}')
print(f'Edge magnitude:    {edges.min():.0f} to {edges.max():.0f}')
print(f'After blur range:  {blurred.min():.0f} to {blurred.max():.0f}')

# Where are the strongest edges?
threshold = edges.mean() + edges.std()
edge_pixels = np.argwhere(edges > threshold)
print(f'\nStrong edge pixels found: {len(edge_pixels)}')
print(f'Edge positions (first 5): {edge_pixels[:5].tolist()}')
```

> 🔍 **Student Task 2**
> **Try the identity kernel: [[0,0,0],[0,1,0],[0,0,0]]. What does it produce? Try a sharpen kernel: [[-1,-1,-1],[-1,9,-1],[-1,-1,-1]]. Describe the difference between blur and sharpen in terms of what they do to pixel values.**

Exercise 3: Feature Maps (CNN Intuition)

A CNN's first layer learns kernels automatically from data. Here we manually show what different learned kernels would detect — giving intuition for what early CNN layers do.

```
# Simulate what a CNN's first layer might learn
kernels = {
    'Vertical edge':   np.array([[-1,0,1],[-2,0,2],[-1,0,1]], np.float32),
    'Horizontal edge': np.array([[-1,-2,-1],[0,0,0],[1,2,1]], np.float32),
    'Diagonal (/)':    np.array([[0,1,2],[-1,0,1],[-2,-1,0]], np.float32),
    'Diagonal (\\)':   np.array([[2,1,0],[1,0,-1],[0,-1,-2]], np.float32),
    'Blob/spot':       np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]], np.float32),
    'Blur':            np.ones((3,3), np.float32)/9,
}

print('Feature map statistics for each kernel:')
print(f'{'Kernel':20} | {'Max activation':15} | {'Active pixels':13} |
{'Detects'}')
print('-'*75)

for name, kernel in kernels.items():
    feature_map = convolve2d(gray_float, kernel)
    max_act  = feature_map.max()
    active   = np.sum(np.abs(feature_map) > 50)  # count strongly activated
    print(f'{name:20} | {max_act:15.1f} | {active:13} | high values = feature
present')

print()
print('In a real CNN, these kernels are LEARNED during training,')
print('not hand-crafted. Deeper layers combine early features')
print('into complex patterns: textures → parts → objects.')
```

🔍 **Student Task 3**

Create a test image with a diagonal line. Which kernel gives the highest max activation? Create an image with a small bright dot in the center — which kernel activates most? Explain why.

Exercise 4: Pre-Trained CNN with PyTorch

Training a CNN from scratch takes days and thousands of images. Instead, we load ResNet-18: a model pre-trained on ImageNet (1.2 million images, 1000 classes). We simply feed it images and read the predictions.

```
import torch
import torchvision.transforms as transforms
from torchvision import models
from PIL import Image
import urllib.request
import json

# ── Load the pre-trained model ──
print('Loading ResNet-18 (pre-trained on ImageNet)...')
model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
model.eval()   # inference mode — disables dropout and batch norm updates
print('Model loaded!')
print(f'Parameters: {sum(p.numel() for p in model.parameters()):,}')

# ── Image preprocessing ──
# ResNet expects: RGB, 224x224, normalized to ImageNet mean/std
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],   # ImageNet channel means
        std= [0.229, 0.224, 0.225]    # ImageNet channel stds
    ),
])

def load_and_preprocess(image_path):
    """Load an image file and prepare it for ResNet"""
    img = Image.open(image_path).convert('RGB')
    print(f'Original size: {img.size}')
    tensor = transform(img).unsqueeze(0)   # add batch dimension: (1,3,224,224)
    print(f'Tensor shape:  {tuple(tensor.shape)}')
    return tensor

def classify_image(image_path, top_k=5):
    """Run ResNet on an image and return top-k predictions"""
    tensor = load_and_preprocess(image_path)

    with torch.no_grad():           # no gradient computation needed
        logits = model(tensor)      # raw scores for 1000 classes
        probs  = torch.softmax(logits, dim=1)[0]   # convert to probabilities
```

```
    # Get top-k predictions
    top_probs, top_indices = torch.topk(probs, top_k)
    return top_probs.tolist(), top_indices.tolist()

# ── Download ImageNet class labels ──
labels_url = ('https://raw.githubusercontent.com/pytorch/hub/master'
              '/imagenet_classes.txt')
urllib.request.urlretrieve(labels_url, 'imagenet_classes.txt')
with open('imagenet_classes.txt') as f:
    LABELS = [line.strip() for line in f.readlines()]

# ── Create test images and classify ──
def create_solid_test_image(color, filename):
    """Save a solid-color image for basic testing"""
    arr = np.full((224, 224, 3), color, dtype=np.uint8)
    Image.fromarray(arr).save(filename)

# Create two test images
create_solid_test_image([34, 139, 34],  'test_green.png')   # forest green
create_solid_test_image([135, 206, 235],'test_sky.png')     # sky blue

print('\n=== Classification Results ===')
for fname, desc in [('test_green.png','Green image'),('test_sky.png','Sky-blue
image')]:
    print(f'\n{desc}:')
    probs, indices = classify_image(fname)
    for prob, idx in zip(probs, indices):
        bar = '#' * int(prob * 40)
        print(f'  {LABELS[idx]:30}  {prob:.3f}  {bar}')

print('\nNote: solid colors are not natural images, so confidence is low.')
print('Use a real photo of an animal or object for meaningful results.')
```

🔍 **Student Task 4**

Download any image from the internet (a dog, a car, a fruit) and save it as 'my_image.jpg'. Run classify_image('my_image.jpg'). What is the top-3 predictions? Is the top prediction correct? What is the confidence score?

Exercise 5: Understanding Model Internals

```
# ── Inspect what ResNet-18 looks like layer by layer ──
print('=== ResNet-18 Architecture ===')
for name, module in model.named_children():
    params = sum(p.numel() for p in module.parameters())
    print(f'  {name:15} | {type(module).__name__:25} | {params:>10,} params')

# ── Extract intermediate feature maps ──
feature_maps = {}

def hook_fn(name):
    def hook(module, input, output):
        feature_maps[name] = output.detach()
    return hook

# Register hooks on first and last conv layers
model.layer1.register_forward_hook(hook_fn('layer1'))
model.layer4.register_forward_hook(hook_fn('layer4'))

# Run a forward pass
dummy = torch.randn(1, 3, 224, 224)   # random image
with torch.no_grad():
    _ = model(dummy)

print('\n=== Feature Map Shapes ===')
for layer_name, fmap in feature_maps.items():
    print(f'  {layer_name}: {tuple(fmap.shape)}')
    print(f'    (batch=1, channels={fmap.shape[1]}, H={fmap.shape[2]},
W={fmap.shape[3]})')

print()
print('layer1: low-level features (edges, colours) — large spatial size')
print('layer4: high-level features (object parts)  — small spatial size')
```

Bonus Challenge

Load two images of the same category (e.g. two different dogs). Extract the layer4 feature map for each. Compute cosine similarity between them. Are features of same-class images more similar to each other than to images from different classes?