

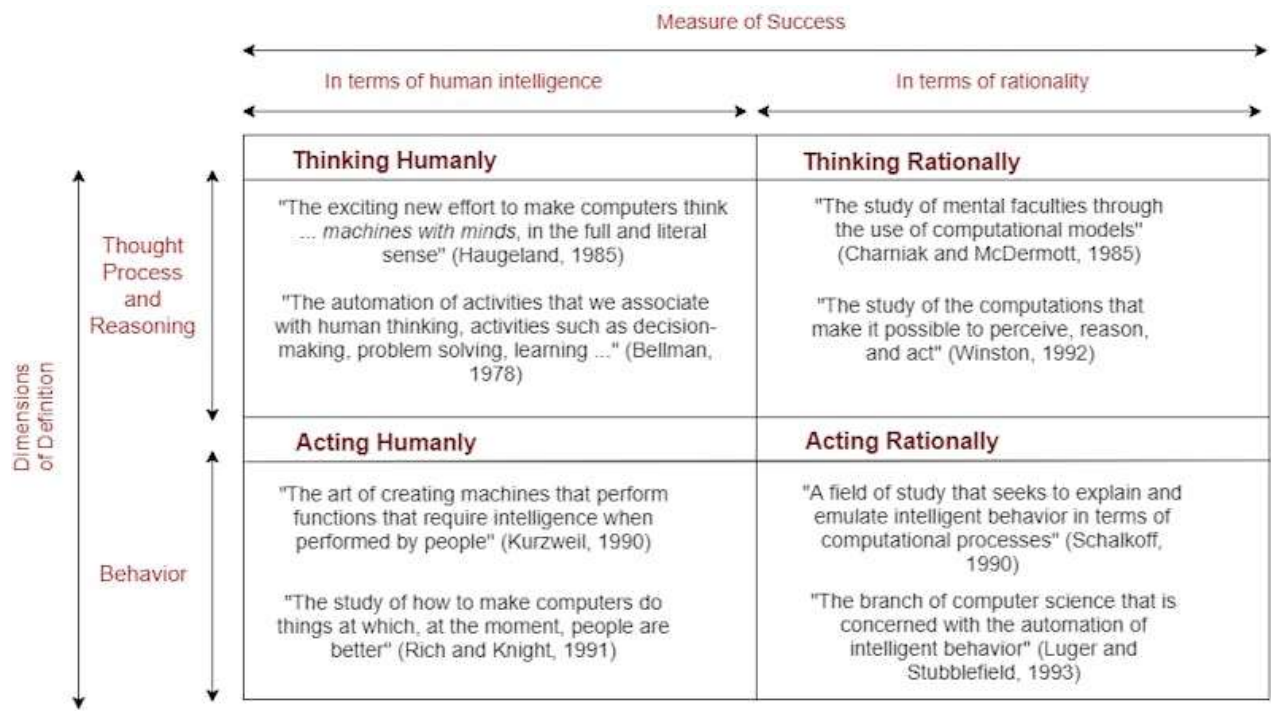


INTRODUCTION TO ARTIFICIAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE

Artificial intelligence, as a computer science discipline, works to develop machines that execute duties that require human cognitive abilities. The human-related operations encompass learning combined with reasoning alongside problem-solving and perception and decision-making paths.

“It is a branch of computer science by which we can create intelligent machines that can behave like a human, think like humans, and be able to make decisions.”



ARTIFICIAL INTELLIGENCE

- **Systems that can think humanly**

- For following this approach, ***one first needs to understand how humans think***. Knowing the internal working of human brain can be achieved by introspection or psychological experiments. This, in itself, is a vast interdisciplinary field, ***known as Cognitive Science***.

- **Systems that can act humanly**

- *This definition came into being when **Alan Turing proposed the Turing Test**. A system passes this test, if it can fool a human interrogator by depicting intelligent behavior. By intelligent behavior, we mean achieving human level performance in cognitive tasks.* Such a system would require to have major components of A.I., including natural language processing, knowledge representation, automated reasoning, machine learning, robotics and computer vision. Seeing the underlying complications, no major effort has been made in trying to make such a machine.

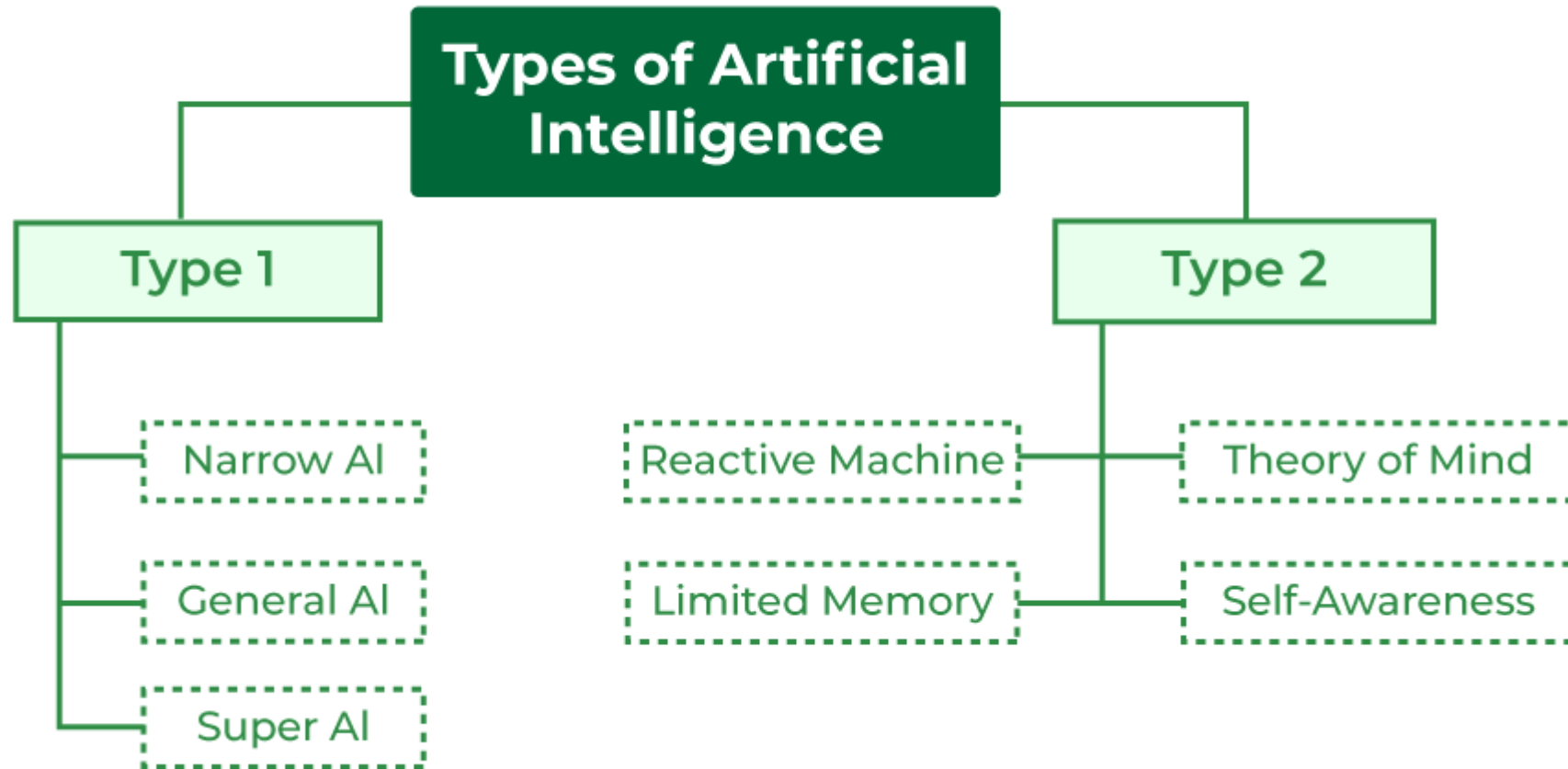


ARTIFICIAL INTELLIGENCE

- Systems that can think rationally
 - *The aim of this approach is to build upon programs that represent “right thinking”, to create intelligent **systems**. This “right thinking” or irrefutable reasoning processes, is defined in coding (in mathematical terms) using logic or laws of thought.*
 1. *Not all knowledge can be expressed with logical notations (especially when knowledge is not 100% certain).*
 2. *It can lead to computational blow up, as without guidance, there are many reasoning steps that can be tried.*
- Systems that can act rationally
 - *This approach involves creating systems that act in a way which maximizes its chances of achieving its goal, given the available information. These systems are known as **Rational Agents**, such that they perceive the*



TYPES OF ARTIFICIAL INTELLIGENCE



AI TYPE 1: BASED ON CAPABILITIES

1.Weak AI or Narrow AI: This type of AI can be used to solve certain problems and focus on a particular kind of job. It is only effective when it is used in a specific area and does not produce the same results when applied in other areas. It applies to smart products, known as virtual assistants such as Siri, systems engaged in image recognition, and IBM's Watson.

2.General AI: General AI, also known as Strong AI, on the other hand, refers to machines capable of achieving any action that a man is capable of accomplishing. It is planning to attain human-like features such as intelligence characteristics like reasoning and learning processes. This is another type of AI that is still under research and has not been developed to its realization.

3.Super AI: An advanced artificial intelligence in which every domain is superior to that of humans in terms of their decision-making power, problem-solving skills, learning capabilities, as well as their feelings and emotions. It is the final stage of AI development, and it does not currently exist in the world.



AI TYPE 2: BASED ON FUNCTIONALITY

1.Reactive Machines: This type of AI processes the current input data and does not have any previous experience. They follow pre-defined rules. Some of the most widely known examples include IBM's chess machine known as Deep Blue and the Go-playing computer termed Google's AlphaGo.

2.Limited Memory: Many of them use the earlier information to establish something for a limited period. Some of the concrete samples include self-driving cars that follow other vehicles, the speed, and the road condition of the environment.

3.Theory of Mind: The purpose of this AI is to comprehend the feelings, desires or even gestures of people. As previously stated, it is still part of the theoretical research and has not been fully realized.

4.Self-Awareness: The final type of artificial intelligence that remains at the level of theory is even more superior to human intelligence as it would have consciousness and feelings. People would consider this level of AI as a significant level of advancement in technology as well as in knowledge.



PROBLEMS OF AI

Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory and computation. Many points on the spectrum of intelligence versus cost are viable, from insects to humans. AI seeks to understand the computations required from intelligent behaviour and to produce computer systems that exhibit intelligence. Aspects of intelligence studied by AI include perception, communication using human languages, reasoning, planning, learning and memory. The following questions are to be considered before we can step forward:

1. What are the underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level human intelligence can be modelled?
4. When will it be realized when an intelligent program has been built?



APPLICATIONS OF AI

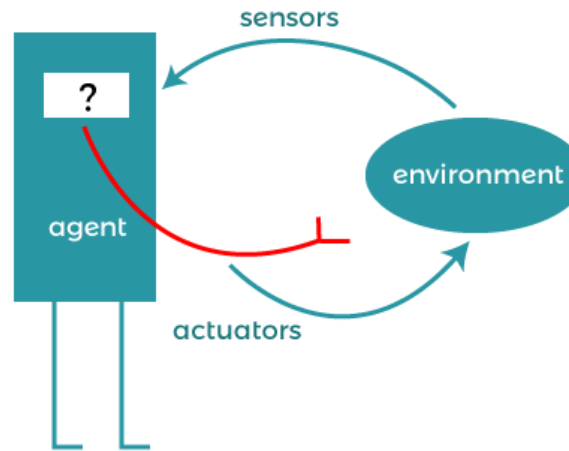
AI has applications in all fields of human study, such as finance and economics, environmental engineering, chemistry, computer science, and so on. Some of the applications of AI are listed below:

- Perception
 - Machine vision
 - Speech understanding
 - Touch (tactile or haptic) sensation
- Robotics
- Natural Language Processing
 - Natural Language Understanding
 - Speech Understanding
 - Language Generation
 - Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing



AI AGENTS

A rational agent may take the form of a person, firm, machine, or software to make decisions. It works with the best results after considering past and present perceptions (perceptual inputs of the agent at a given instance). An AI system is made up of an agent and its environment. Agents work in their environment, and the environment may include other agents.

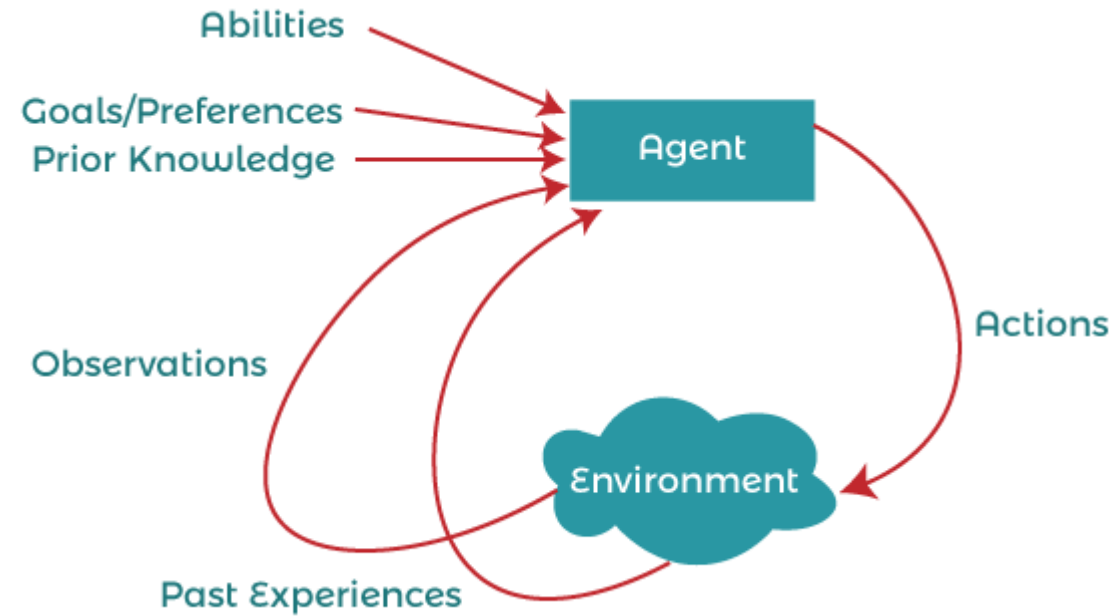


agent = architecture + agent program

- A software agent has keystrokes, file contents, received network packages that act as sensors and are displayed on the screen, files, sent network packets to act as actuators.
- The human agent has eyes, ears, and other organs that act as sensors, and hands, feet, mouth, and other body parts act as actuators.
- A robotic agent consists of cameras and infrared range finders that act as sensors and various motors that act as actuators.



AI AGENTS



Types of agents

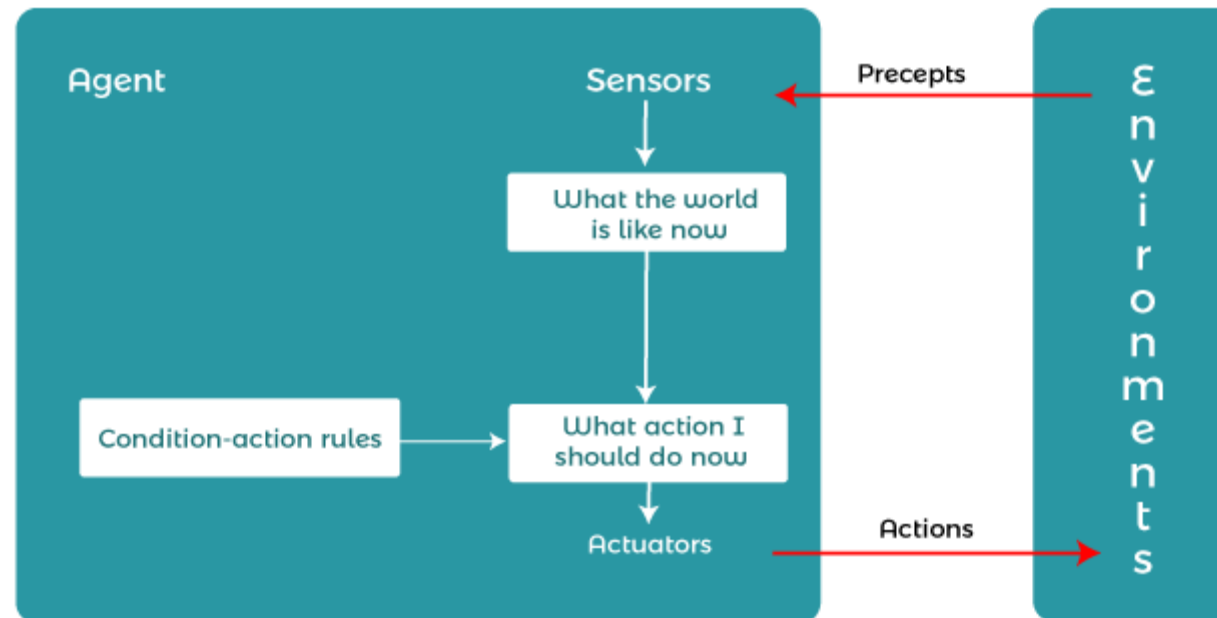
Agents can be divided into four classes based on their perceived intelligence and ability:

- Simple reflex agent
- Model-based reflex agent
- Target-based agent
- Utility-based agent
- Learning agent



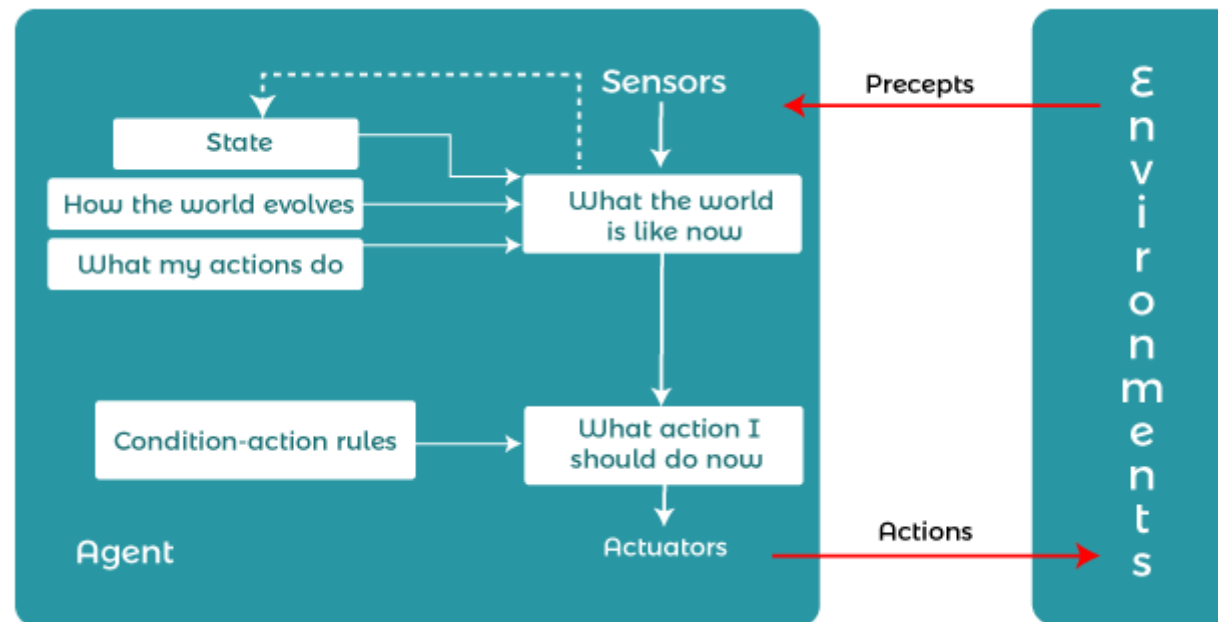
AI AGENTS

Simple reflex agent: Simple reflex agents ignore the rest of the concept history and act only based on the current concept. Concept history is the history of all that an agent has believed to date. The agent function is based on the condition-action rule. A condition-action rule is a rule that maps a state, that is, a condition, to an action. If the condition is true, then action is taken; otherwise, not.



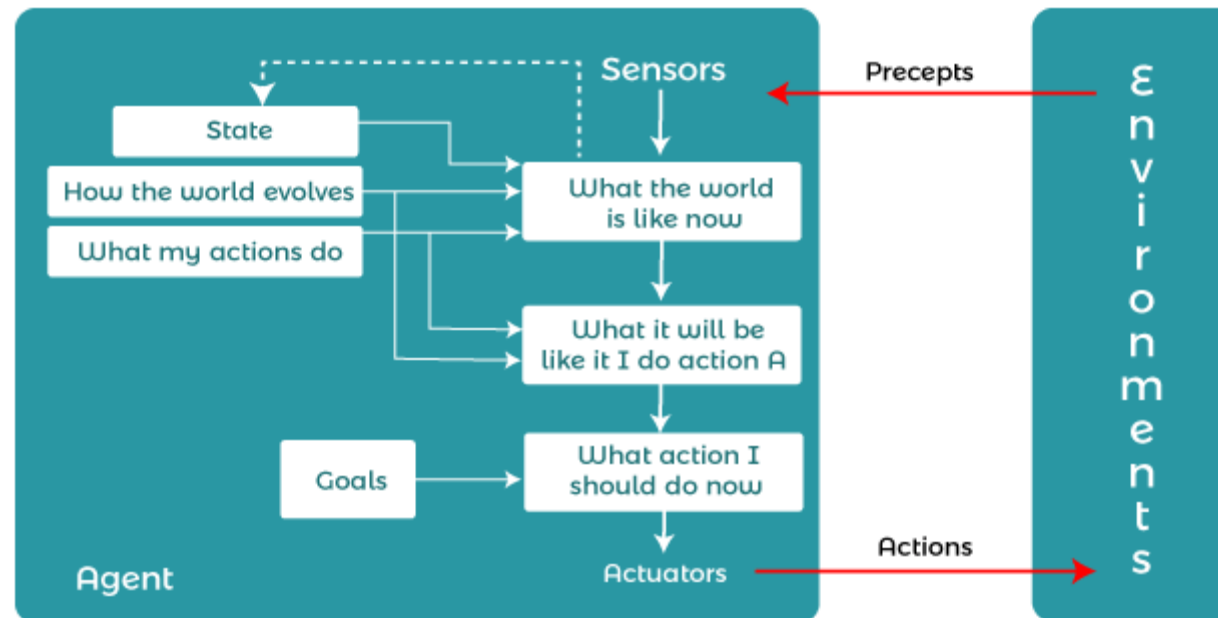
AI AGENTS

Model-based reflex agents: It works by searching for a rule whose position matches the current state. A model-based agent can handle a partially observable environment using a model about the world. The agent has to keep track of the internal state, adjusted by each concept, depending on the concept history. The current state is stored inside the agent, which maintains some structure describing the part of the world that cannot be seen.



AI AGENTS

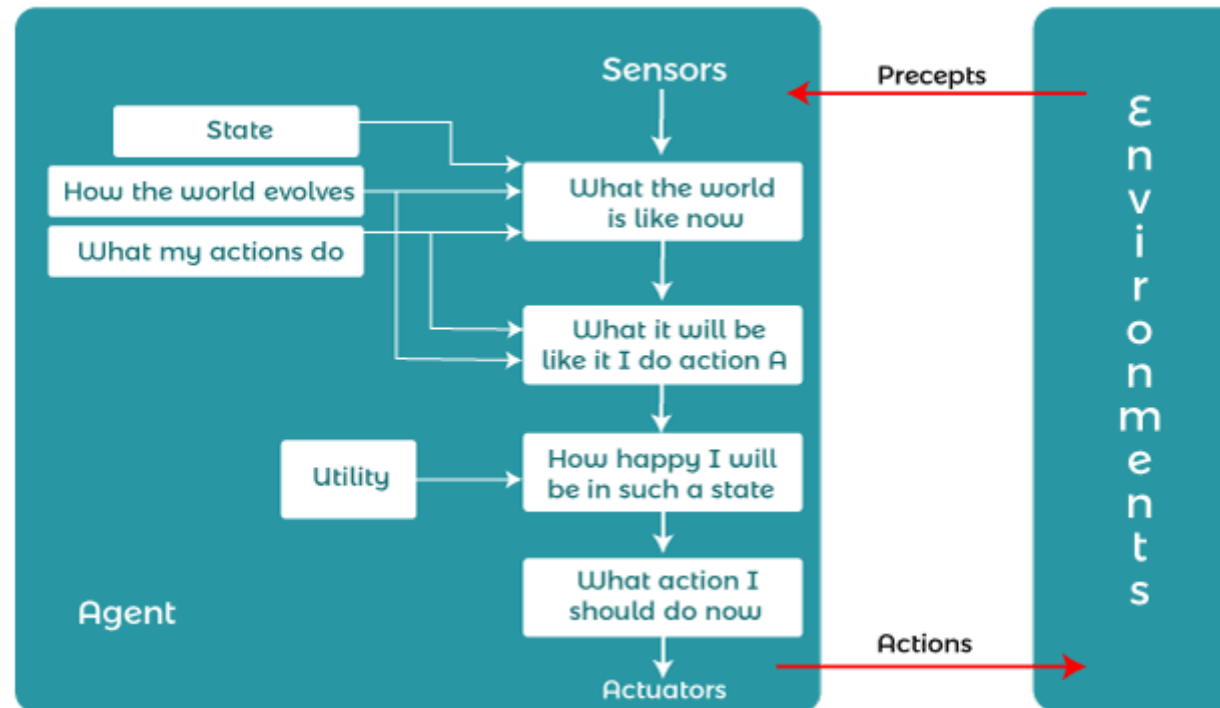
Goal-based agents: These types of agents make decisions based on how far they are currently from their goals (details of desired conditions). Their every action is aimed at reducing its distance from the target. This gives the agent a way to choose from a number of possibilities, leading to a target position. The knowledge supporting their decisions is clearly presented and can be modified, which makes these agents more flexible.



AI AGENTS

Utility-based agents: The agents which are developed having their end uses as building blocks are called utility-based agents. When there are multiple possible alternatives, then to decide which one is best, utility-based agents are used. They choose actions based on a **preference (utility)** for each state. Sometimes achieving the desired goal is not enough. We may look for a quicker, safer, cheaper trip to reach a destination. Agent happiness should be taken into consideration.

Utility describes how "**happy**" the agent is. Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected utility. A utility function maps a state onto a real number which describes the associated degree of happiness.

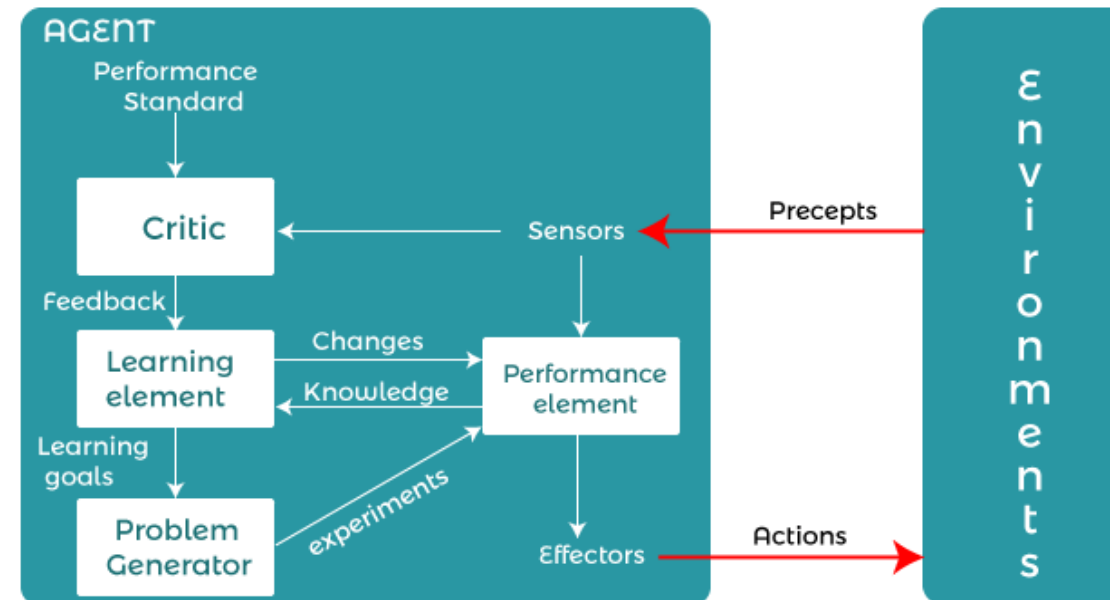


AI AGENTS

Learning Agent: A learning agent in AI is the type of agent that can learn from its past experiences or it has learning capabilities. It starts to act with basic knowledge and then is able to act and adapt automatically through learning.

A learning agent has mainly four conceptual components, which are:

- **Learning element:** It is responsible for making improvements by learning from the environment
- **Critic:** The learning element takes feedback from critics which describe how well the agent is doing with respect to a fixed performance standard.
- **Performance element:** It is responsible for selecting external action
- **Problem Generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.



PEAS IN ARTIFICIAL INTELLIGENCE

PEAS stands for performance measure, environment, actuators, and sensors. PEAS defines AI models and helps determine the task environment for an intelligent agent.

- **Performance measure:** It defines the success of an agent. It evaluates the criteria that determines whether the system performs well.
- **Environment:** It refers to the external context in which an AI system operates. It encapsulates the physical and virtual surroundings, including other agents, objects, and conditions.
- **Actuators:** They are responsible for executing actions based on the decisions made. They interact with the environment to bring about desired changes.
- **Sensors:** An agent observes and perceives its environment through sensors. Sensors provide input data to the system, enabling it to make informed decisions.



PEAS IN ARTIFICIAL INTELLIGENCE

Agent	Performance measure	Environment	Actuators	Sensors
Vacuum cleaner	Cleanliness, security, battery	Room, table, carpet, floors	Wheels, brushes	Camera, sensors
Chatbot system	Helpful responses, accurate responses	Messaging platform, internet, website	Sender mechanism, typer	NLP algorithms
Autonomous vehicle	Efficient navigation, safety, time, comfort	Roads, traffic, pedestrians, road signs	Brake, accelerator, steer, horn	Cameras, GPS, speedometer
Hospital	Patient's health, cost	Doctors, patients, nurses, staff	Prescription, diagnosis, tests, treatments	Symptoms



PROBLEM, PROBLEM SPACE, SEARCH

A **problem** is a specific task or challenge that requires finding a solution or making a decision. In artificial intelligence, problems can vary in complexity and scope, ranging from simple tasks like arithmetic calculations to complex challenges such as image recognition, natural language processing, game playing, and optimization. Each problem has a defined set of initial states, possible actions or moves, and a goal state that needs to be reached or achieved.

The **problem space** is the set of all possible states, actions, and transitions that can be encountered while attempting to solve a specific problem. It represents the entire landscape of potential solutions and paths from the initial state to the goal state. In other words, the problem space defines all the possible configurations or arrangements of elements involved in the problem and the set of valid moves or actions that can be taken at each state. Each state in the problem space represents a specific configuration, and each action represents a possible move or step from one state to another.

Search is the process of exploring the problem space to find a sequence of actions or moves that lead to the goal state or a satisfactory solution. In AI, search algorithms are used to systematically navigate through the problem space and discover paths or solutions that satisfy the problem's constraints and objectives.



PROBLEM CHARACTERISTICS

- **Is the problem decomposable?**

The question of whether a problem is decomposable refers to whether it can be broken down into smaller, independent subproblems. Decomposable problems can be solved by tackling each subproblem individually, and their solutions can then be combined to solve the overall problem. Some problems, like complex integrals, can be decomposed into simpler subproblems, making it easier to find a solution using the divide-and-conquer approach. However, not all problems are decomposable, and some may require addressing as a whole without breaking them down into independent parts.

- **Can solution steps be ignored or undone?**

The reversibility of solution steps refers to whether they can be ignored or undone if they prove to be unwise or lead to a dead end. In some problems, certain solution steps can be ignored without affecting the final result. In recoverable problems, solution steps can be undone to explore alternative paths. For instance, in the 8-puzzle, moves can be undone to try different arrangements of tiles. On the other hand, some problems have irreversible solution steps, like in chess, where once a move is made, it cannot be undone.



PROBLEM, PROBLEM SPACE, SEARCH

- **Is the problem's universe predictable?**

The predictability of a problem's universe refers to whether the outcomes or states of the problem can be determined with certainty or if they involve uncertainty. Some problems have deterministic outcomes, meaning that the result is known and can be predicted with complete certainty based on the given conditions and rules. Other problems may involve uncertainty or randomness, leading to non-deterministic outcomes. For example, some optimization problems may have multiple potential solutions with different probabilities of being optimal.

- **Is a good solution absolute or relative?**

The nature of a good solution can be either absolute or relative. An absolute solution is one where finding a single correct path or outcome is sufficient to achieve the desired goal. In problems like the water jug puzzle, finding any valid path to the solution is considered good enough. On the other hand, a relative solution is one that requires evaluating multiple possible paths or outcomes to find the best or optimal solution. Problems like the traveling salesman problem seek the shortest route among all possible routes, making it a relative solution.



PROBLEM, PROBLEM SPACE, SEARCH

- **Is the solution a state or a path?**

The solution to a problem can be either a state or a path, depending on the nature of the problem. In some problems, the desired outcome is a specific state or configuration that satisfies the problem's requirements. For instance, in the 8-puzzle, the solution is a specific arrangement of tiles in the goal state. In other problems, the solution involves finding a path or sequence of steps to reach the desired goal state. For example, in maze solving, the solution is the path from the starting point to the exit.

- **What is the role of knowledge?**

The role of knowledge in problem-solving varies based on the complexity and nature of the problem. Knowledge plays a critical role in guiding the problem-solving process. In some problems, extensive domain specific knowledge is required to recognize patterns, constraints, and possible solutions. For example, chess requires deep knowledge of the game rules and strategic principles to make informed moves. In contrast, other problems may rely more on general problem-solving algorithms and heuristics, requiring less domain-specific knowledge.

- **Can a computer give the problem solution, or interaction with humans is required?**

The level of human interaction required in problem-solving depends on the problem's complexity and the capabilities of the problem-solving methods being used. In some cases, computers can autonomously find solutions to problems without any interaction with humans. For example, algorithms can efficiently solve mathematical equations or perform certain optimization tasks. However, in more complex and uncertain problems, human interaction may be necessary to provide additional information, preferences, or guidance. Conversational problem-solving, where the computer interacts with users to gather information or provide assistance, can be valuable in addressing such challenges.

STATE SPACE

A **state space** is a way to mathematically represent a problem by defining all the possible states in which the problem can be. This is used in search algorithms to represent the initial state, goal state, and current state of the problem. Each state in the state space is represented using a set of variables.

State space search has several features that make it an effective problem-solving technique in Artificial Intelligence.

These features include:

- **Exhaustiveness:** State space search explores all possible states of a problem to find a solution.
- **Completeness:** If a solution exists, state space search will find it.
- **Optimality:** Searching through a state space results in an optimal solution.
- **Uninformed and Informed Search:** State space search in artificial intelligence can be classified as uninformed if it provides additional information about the problem.



STATE SPACE REPRESENTATION

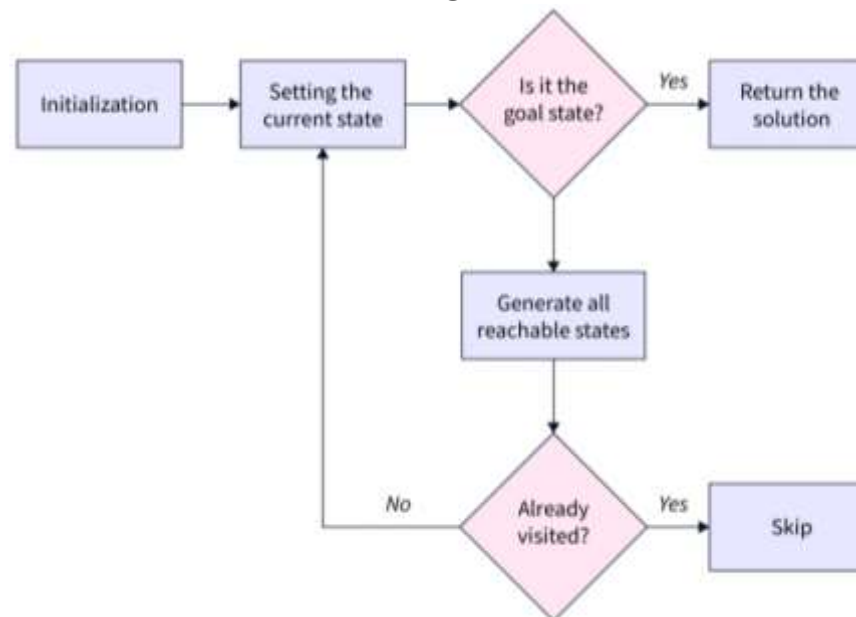
State space Representation involves defining an INITIAL STATE and a GOAL STATE and then determining a sequence of actions, called states, to follow.

- **State:** A state can be an Initial State, a Goal State, or any other possible state that can be generated by applying rules between them.
 - **Initial State:** The state of the issue as it first arises.
 - **Goal State:** The idealized final state that delineates a problem-solving strategy.
 - **Operators:** The collection of maneuvers or actions that can be used to change a state.
 - **Restrictions:** Guidelines or limitations must be adhered to to solve the problem.
- **Space:** In an AI problem, space refers to the exhaustive collection of all conceivable states.
- **Search:** This technique moves from the beginning state to the desired state by applying good rules while traversing the space of all possible states.
- **Search Tree:** To visualize the search issue, a search tree is used, which is a tree-like structure that represents the problem. The initial state is represented by the root node of the search tree, which is the starting point of the tree.
- **Transition Model:** This describes what each action does, while Path Cost assigns a cost value to each path, an activity sequence that connects the beginning node to the end node. The optimal option has the lowest cost among all alternatives.



STATE SPACE REPRESENTATION

- To begin the search process, we set the current state to the initial state.
- We then check if the current state is the goal state. If it is, we terminate the algorithm and return the result.
- If the current state is not the goal state, we generate the set of possible successor states that can be reached from the current state.
- For each successor state, we check if it has already been visited. If it has, we skip it, else we add it to the queue of states to be visited.
- Next, we set the next state in the queue as the current state and check if it's the goal state. If it is, we return the result. If not, we repeat the previous step until we find the goal state or explore all the states.
- If all possible states have been explored and the goal state still needs to be found, we return with no solution.



EXAMPLE OF STATE SPACE

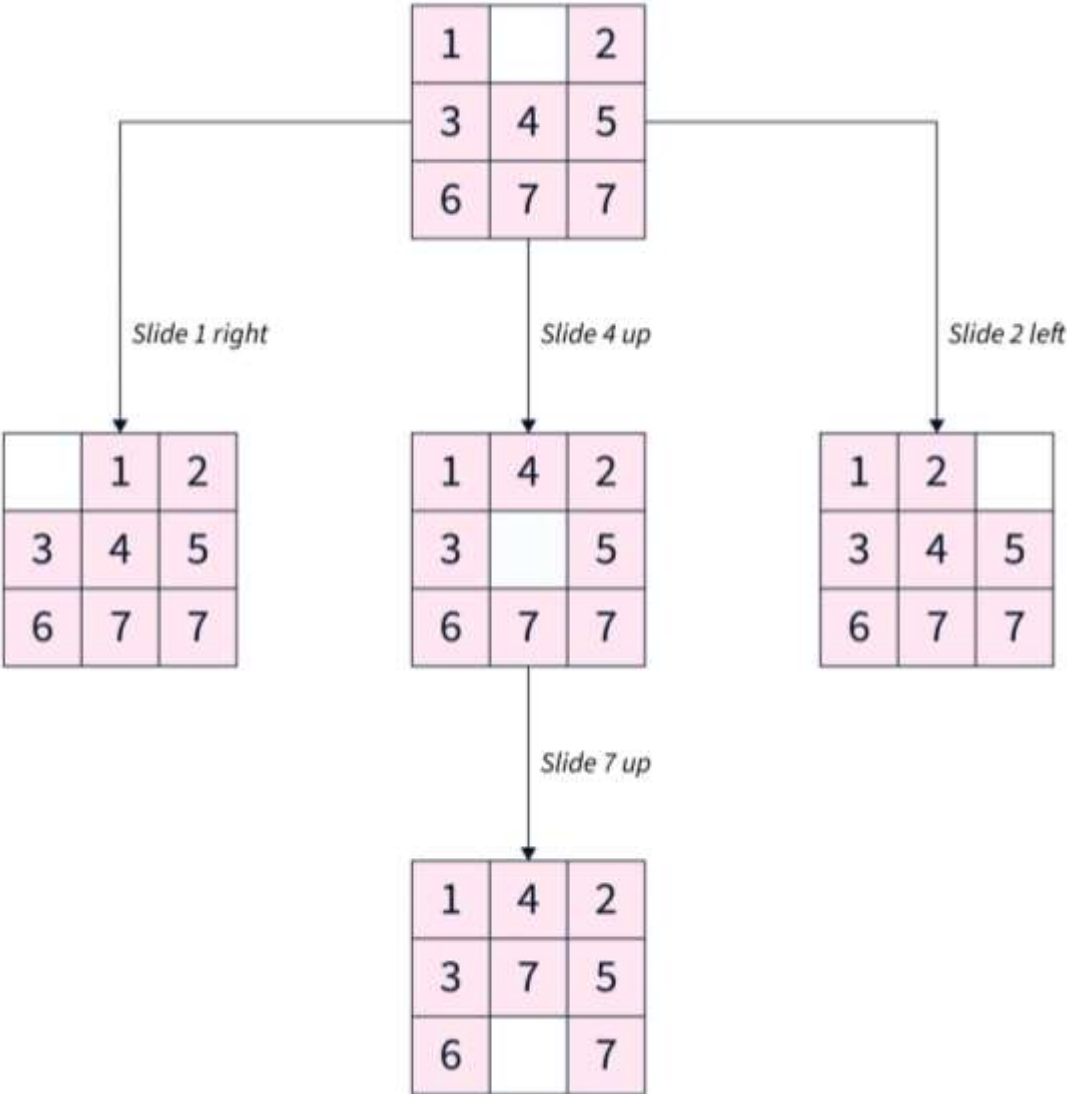
8-puzzle problem

1		2
3	4	5
6	7	7

Current State

1	4	2
3	7	5
6		7

Target State



STATE SPACE SEARCH

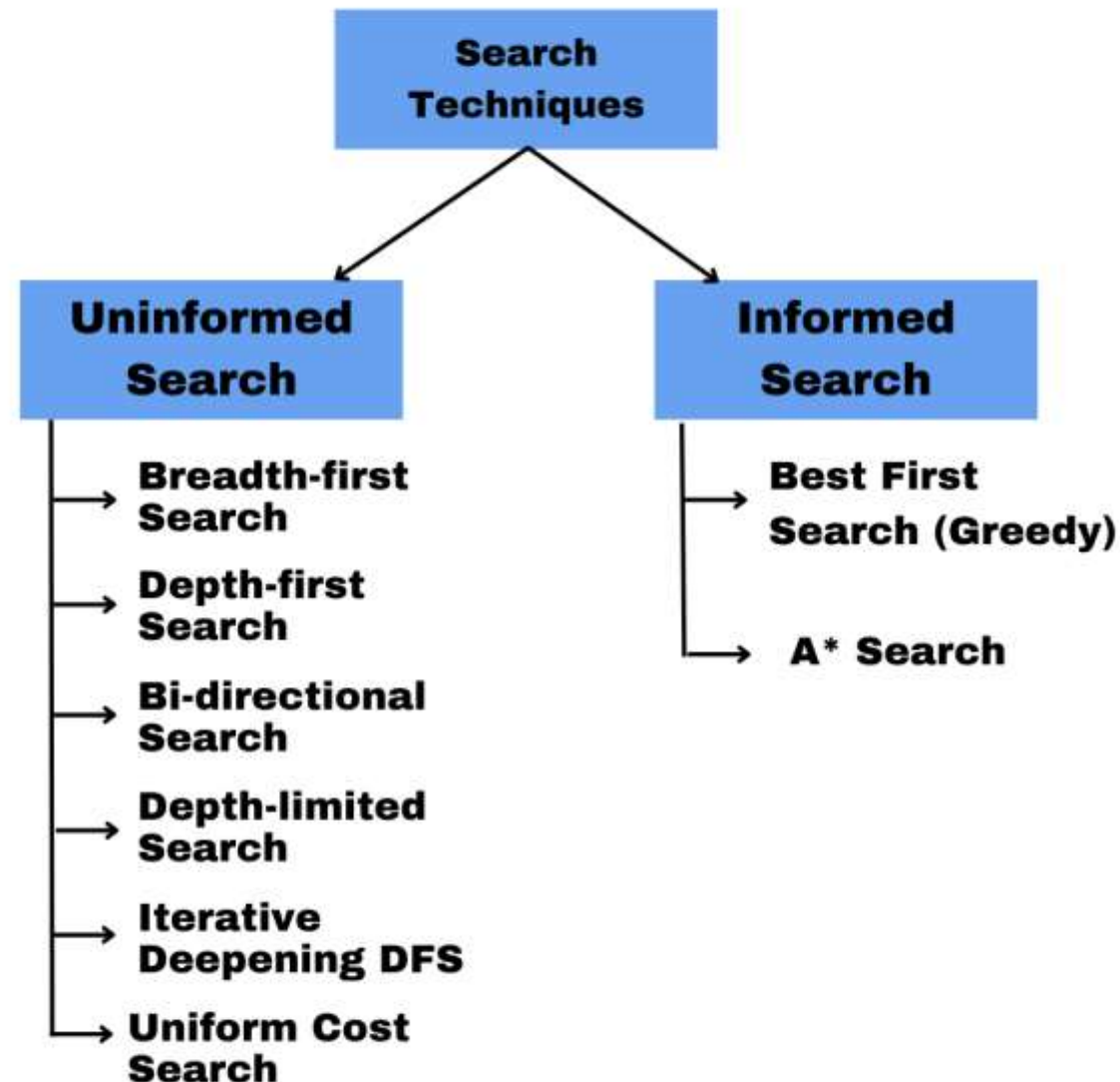
Search algorithms in AI provide search solutions by transforming the initial state to the desired state.

Properties of Search Algorithms: The four important properties of search algorithms in artificial intelligence for comparing their efficiency are as follows:

- **Completeness** - A search algorithm is said to be complete if it guarantees to yield a solution for any random input if at least one solution exists.
- **Optimality** - A solution discovered for an algorithm is considered optimal if it is assumed to be the best solution (lowest path cost) among all other solutions.
- **Time complexity** - It measures how long an algorithm takes to complete its job.
- **Space Complexity** - The maximum storage space required during the search, as determined by the problem's complexity.



STATE SPACE SEARCH



STATE SPACE SEARCH

Uninformed/Blind Search

The uninformed search needs domain information, such as proximity or goal location. It works by brute force because it only contains information on traversing the tree and identifying leaf and goal nodes.

Uninformed search is a method of searching a search tree without knowledge of the search space, such as initial state operators and tests for the objective, and is also known as blind search. It goes through each tree node until it reaches the target node. These algorithms are limited to producing successors and distinguishing between goal and non-goal states.

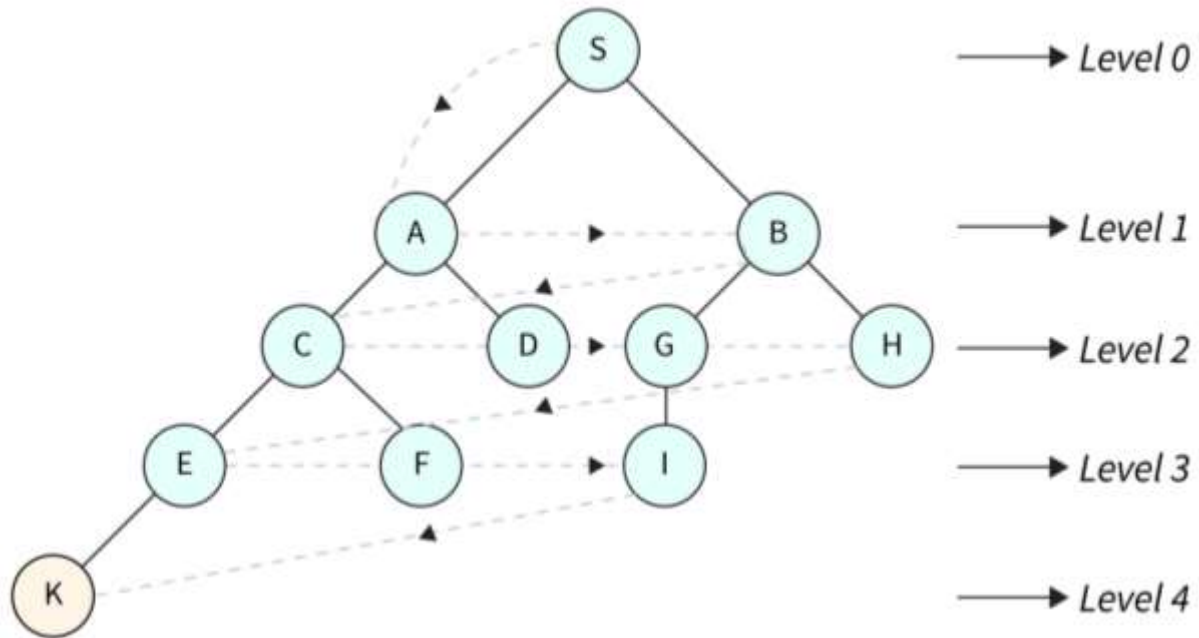
1.Breadth-first search - This is a search method for a graph or tree data structure. It starts at the tree root or searches key and goes through the adjacent nodes in the current depth level before moving on to the nodes in the next depth level. It uses the queue data structure that works on the first in, first out (FIFO) concept. It is a complete algorithm as it returns a solution if a solution exists.

2.Depth-first search - It is also an algorithm used to explore graph or tree data structures. It starts at the root node, as opposed to the breadth-first search. It goes through the branch nodes and then returns. It is implemented using a stack data structure that works on the concept of last in, first out (LIFO).

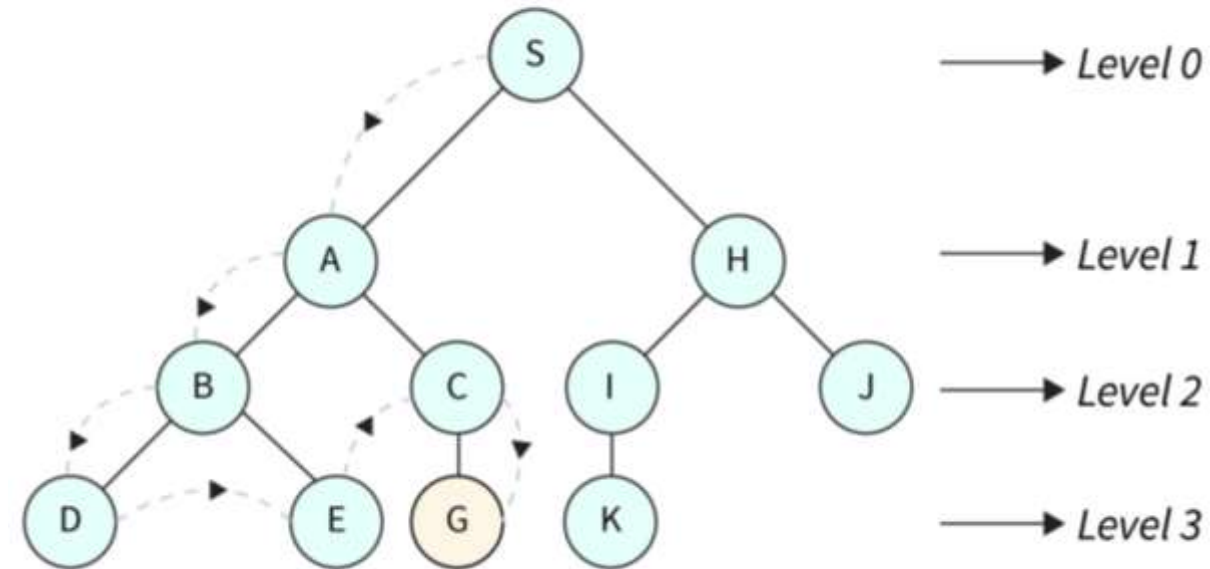


STATE SPACE SEARCH

Breadth First Search



Depth First Search



STATE SPACE SEARCH

Breadth First Search Algorithm

- **Initialization:**
 1. Start with a designated "**SOURCE**" node and a "**TARGET**" node.
 2. Initialize a queue and add the source node to it.
 3. Maintain a "visited" set or array to keep track of nodes already explored, preventing cycles and redundant processing.
 4. Optionally, maintain a "**PARENT**" dictionary or array to reconstruct the path later, storing the node from which each visited node was reached.
- **Exploration:**
 1. While the queue is not empty:
 - a. Dequeue a node (the "**CURRENT**" node).
 - b. If the current node is the **TARGET** node, the path is found. Reconstruct the path by backtracking through the **PARENT** pointers from the target to the source.
 - c. For each unvisited neighbor of the **CURRENT** node:
 - i. Mark the neighbor as visited.
 - ii. Set the **CURRENT** node as the neighbor's parent.
 - iii. Enqueue the neighbor.
- **Termination:** The algorithm terminates when the **TARGET** node is found or when the queue becomes empty (meaning the target is unreachable from the source).



STATE SPACE SEARCH

Depth First Search Algorithm

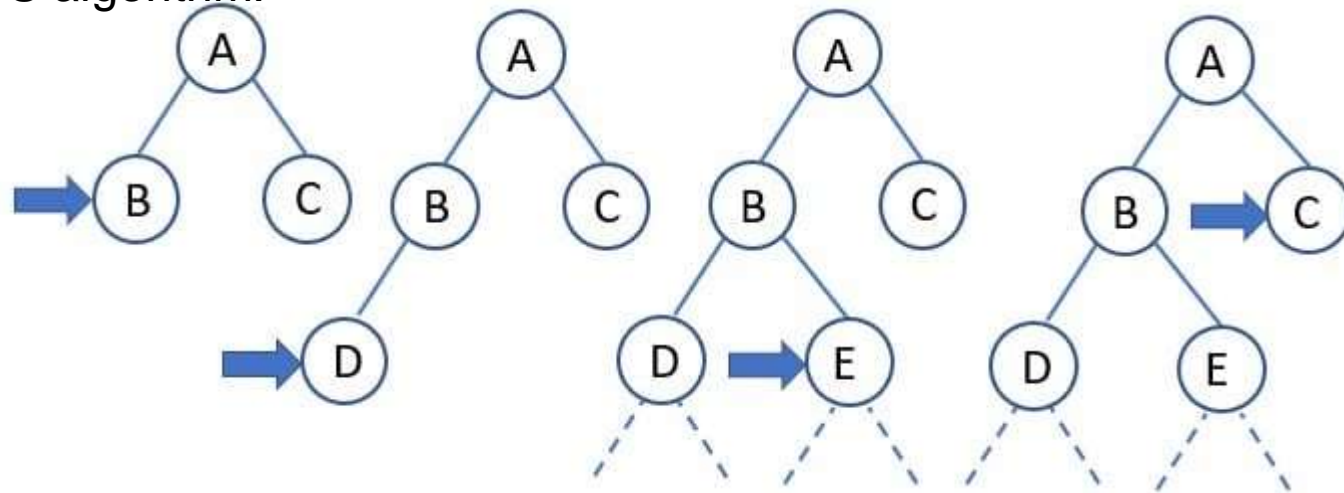
- **Initialization:**
 1. Start with a designated *source* node and a *target* node.
 2. Initialize a **stack** and push the source node onto it.
 3. Maintain a **visited** set or array to keep track of nodes that have already been explored, preventing infinite loops in cyclic graphs.
 4. Optionally, maintain a **parent** dictionary or array to reconstruct the path later, storing the node from which each visited node was reached.
- **Exploration:**
 1. While the stack is not empty:
 - a. Pop a node (the **current** node) from the top of the stack.
 - b. If the current node is the target node, the path is found.
 - i. Reconstruct the path by backtracking through the parent pointers from the target to the source.
 - c. For each unvisited neighbor of the current node:
 - i. Mark the neighbor as visited.
 - ii. Set the current node as the neighbor's parent.
 - iii. Push the neighbor onto the stack.
- **Termination:** The algorithm terminates when the target node is found or when the stack becomes empty (meaning the target is unreachable from the source).



STATE SPACE SEARCH

Depth Limited Depth First Search Algorithm

Depth limited search is the new search algorithm for uninformed search. The unbounded tree problem happens to appear in the depth-first search algorithm, and it can be fixed by imposing a boundary or a limit to the depth of the search domain. We will say that this limit as the depth limit, making the DFS search strategy more refined and organized into a finite loop. We denote this limit by l , and thus this provides the solution to the infinite path problem that originated earlier in the DFS algorithm.



STATE SPACE SEARCH

Depth Limited Depth First Search Algorithm

Initialization:

1. Start with a designated *source* node and a *target* node.
2. Define a maximum **depth limit** up to which the search will explore.
3. Maintain a **visited** set or array to avoid revisiting nodes in the current path.
4. Optionally, maintain a **parent** dictionary or array to reconstruct the path later.

Exploration (Recursive or Stack-Based):

1. Begin at the source node with the current depth = 0.
2. If the current node is the target node, return success and reconstruct the path using parent pointers.
3. If the current depth equals the depth limit, terminate this branch (do not expand further).
4. For each unvisited neighbor of the current node:
 - a. Mark the neighbor as visited.
 - b. Set the current node as the neighbor's parent.
 - c. Recursively call Depth-Limited Search on the neighbor with depth + 1.

Termination:

1. The search ends when:
 - a. The target node is found within the depth limit (success), or
 - b. All paths are explored up to the depth limit without finding the target (failure).



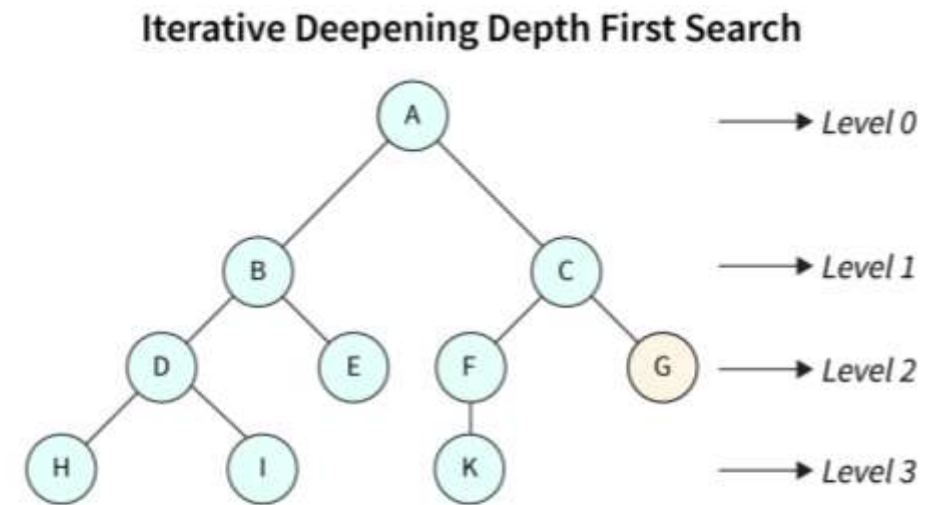
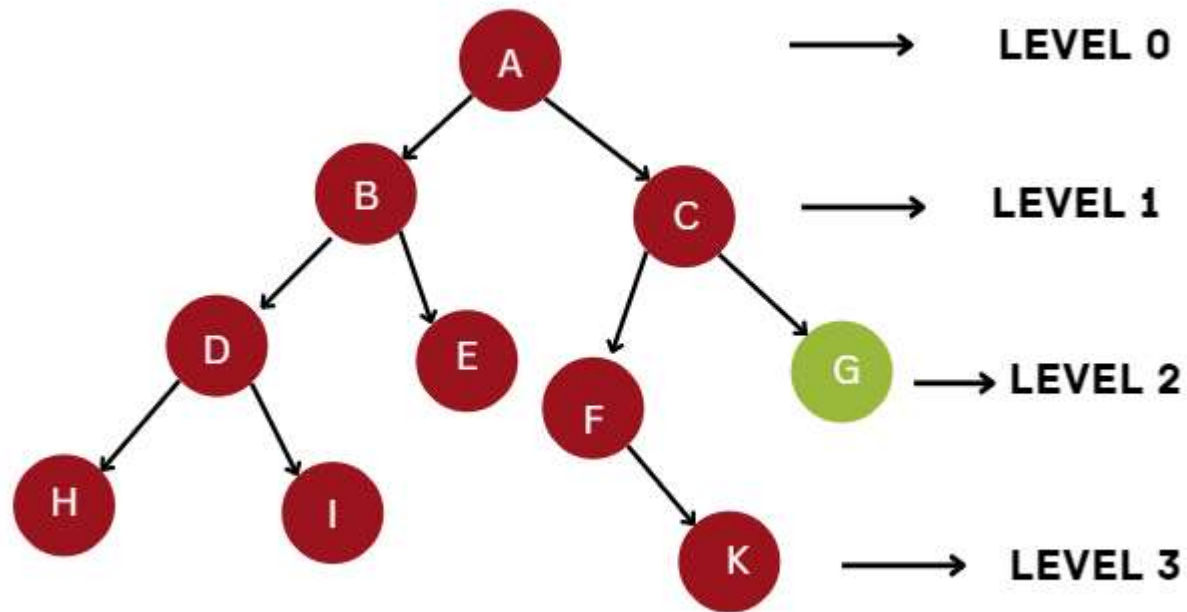
STATE SPACE SEARCH

Iterative Deepening Depth-First Search (IDDFS) Algorithm

IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically, we do DFS in a BFS fashion.



1st Iteration -----> A

2nd Iteration -----> A, B, C

3rd Iteration -----> A, B, D, E, C, F, G

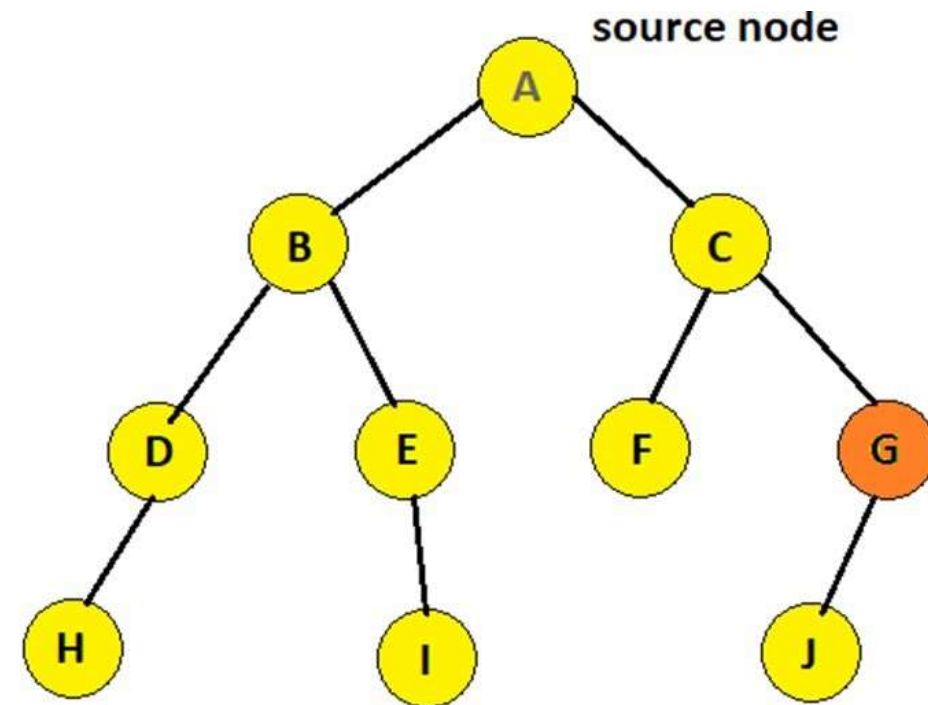
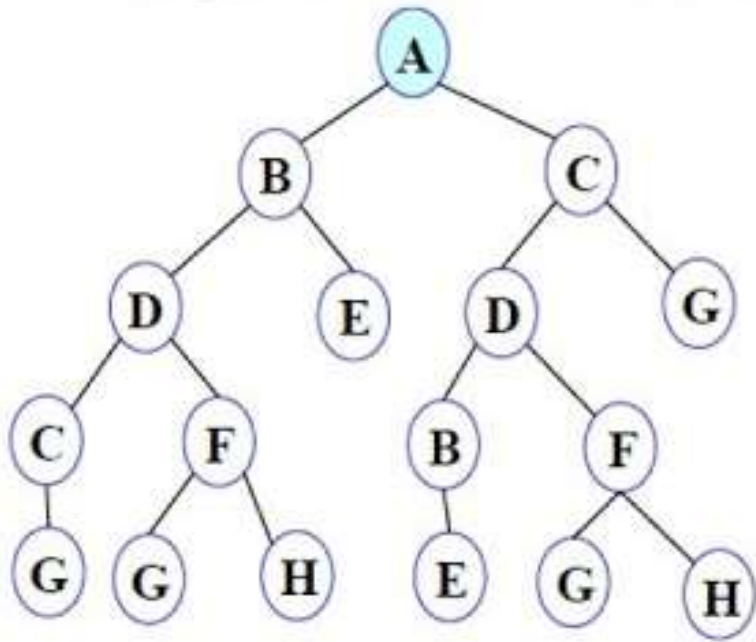
4th Iteration -----> A, B, D, H, I, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

www.veerpreps.com

STATE SPACE SEARCH

Exercise



STATE SPACE SEARCH

Iterative Deepening Depth-First Search (IDDFS) Algorithm

- **Initialization:**
 1. Start with a designated *source* node and a *target* node.
 2. Set an initial **depth limit** to 0.
 3. Incrementally increase the depth limit by 1 in each iteration until:
 - a. The target node is found (success), or
 - b. The maximum allowed depth is reached (failure).
- **Exploration:**
 1. For each depth limit d :
 - a. Perform a **Depth-Limited Search (DLS)** starting from the source node with the current depth = 0 and limit = d .
 - b. If DLS finds the target node, terminate and reconstruct the path.
 - c. If DLS reaches the depth limit without success, increase the limit and repeat.
- **Termination:**
 1. The algorithm terminates when:
 - a. The target node is found at some depth limit (success), or
 - b. All nodes up to the maximum depth have been explored without finding the target (failure).



WATER JUG PROBLEM

The Water Jug Problem is a classic puzzle in artificial intelligence. In this version, there are two jugs: one with a capacity of 4 liters and another with a capacity of 3 liters. Neither jug has any measurement markings. A pump is available to completely fill either jug with water.

The objective is to measure exactly 2 liters of water in the 4-liter jug. Starting with both jugs empty, the task is to determine a sequence of steps that results in exactly 2 liters being poured into the 4-liter jug.

In production rules for the water jug problem, let x denote a 4-litre jug, and y denote a 3-litre jug, i.e. $x=0,1,2,3,4$ or $y=0,1,2,3$

➤ **Start state/Initial State** = $(0,0)$, **Goal state** = $(2,n)$ from any n

Production rules for the water jug problem in AI are as follows:

1.	(x,y) is $X<4 \rightarrow (4, Y)$	Fill the 4-litre jug
2.	(x, y) if $Y<3 \rightarrow (x, 3)$	Fill the 3-litre jug
3.	(x, y) if $x>0 \rightarrow (x-d, d)$	Pour some water from a 4-litre jug
4.	(x, y) if $Y>0 \rightarrow (d, y-d)$	Pour some water from a 3-litre jug
5.	(x, y) if $x>0 \rightarrow (0, y)$	Empty 4-litre jug on the ground
6.	(x, y) if $y>0 \rightarrow (x,0)$	Empty 3-litre jug on the ground

7.	(x, y) if $X+Y \geq 4$ and $y>0 \rightarrow (4, y-(4-x))$	Pour water from a 3-litre jug into a 4-litre jug until it is full
8.	(x, y) if $X+Y \geq 3$ and $x>0 \rightarrow (x-(3-y), 3)$	Pour water from a 3-litre jug into a 4-litre jug until it is full
9.	(x, y) if $X+Y \leq 4$ and $y>0 \rightarrow (x+y, 0)$	Pour all the water from a 3-litre jug into a 4-litre jug
10.	(x, y) if $X+Y \leq 3$ and $x>0 \rightarrow (0, x+y)$	Pour all the water from a 4-litre jug into a 3-litre jug
11.	$(0, 2) \rightarrow (2, 0)$	Pour 2-litre water from 3-litre jug into 4-litre jug
12.	$(2, Y) \rightarrow (0, y)$	Empty 2-litre in the 4-litre jug on the ground.



WATER JUG PROBLEM

Step	State (x, y)	Applied Rule	Description
0	(0, 0)	—	Initial state
1	(0, 3)	Rule 2	Fill 3L jug
2	(3, 0)	Rule 9	Pour 3L into 4L
3	(3, 3)	Rule 2	Fill 3L jug
4	(4, 2)	Rule 7	Pour from 3L → 4L until full
5	(0, 2)	Rule 5	Empty 4L jug
6	(2, 0)	Rule 11	Pour 2L into 4L jug (Goal)

BFS State-Space Tree

- **Level 0 (Initial State)**
(0, 0)
- **Level 1 (Apply Rule 1 & 2)**
(4, 0) – Fill 4L jug
(0, 3) – Fill 3L jug
- **Level 2 (From (0, 3) using Rule 9)**
(3, 0) – Pour 3L → 4L
- **Level 3 (From (3, 0) using Rule 2)**
(3, 3) – Fill 3L jug
- **Level 4 (From (3, 3) using Rule 7)**
(4, 2) – Pour 3L → 4L until full
- **Level 5 (From (4, 2) using Rule 5)**
(0, 2) – Empty 4L jug
- **Level 6 (From (0, 2) using Rule 11)**
(2, 0) – Pour 2L → 4L [Goal]



STATE SPACE SEARCH

Informed/Heuristic Search

Informed search algorithms are a class of algorithms in artificial intelligence that use heuristic functions to guide the search process. A heuristic is a function that estimates the cost of reaching the goal from a given node, providing additional information that helps the algorithm make smarter decisions.

Role of Heuristics

Heuristics play a crucial role in informed search algorithms. They help prioritize which nodes or paths the algorithm should explore first by estimating how close a node is to the goal. This dramatically reduces the number of states explored, making the search process more efficient.

Comparison with Uninformed Search

In contrast to informed search, **uninformed search algorithms** like breadth-first or depth-first search explore the search space without any additional information, often leading to longer search times and inefficient exploration. For example, breadth-first search explores all possible states level by level, which can be highly time-consuming in large search spaces.



STATE SPACE SEARCH

Key Characteristics of Informed Search Algorithms

- 1. Heuristic Function:** Informed search algorithms use a heuristic function $h(n)$ that provides an estimate of the minimal cost from node n to the goal. This function helps the algorithm to prioritize which nodes to explore first based on their potential to lead to an optimal solution.
- 2. Efficiency:** By focusing on more promising paths, informed search algorithms often find solutions more quickly than uninformed methods, especially in large or complex search spaces.
- 3. Optimality and Completeness:** Depending on the heuristic used, informed search algorithms can be both optimal and complete. An algorithm is complete if it is guaranteed to find a solution if one exists, and it is optimal if it always finds the best solution. For instance, the A* search algorithm is both complete and optimal when the heuristic function is admissible (i.e., it never overestimates the true cost).
- 4. Admissibility and Consistency of Heuristics:** An admissible heuristic guarantees that the algorithm finds the optimal solution. A heuristic is admissible if it never overestimates the true cost to reach the goal.

1. Hill climbing Search
2. Greedy Best-First Search
3. A* Algorithm
4. AO* Algorithm



HILL CLIMBING

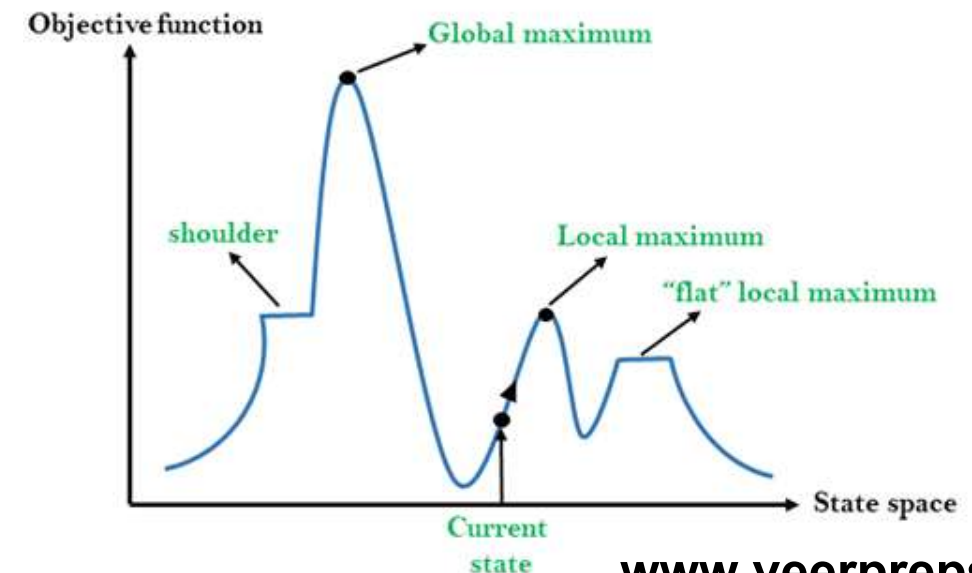
Hill climbing is a widely used **optimization algorithm** in **Artificial Intelligence (AI)** that helps find the best possible solution to a given problem. As part of the **local search algorithms** family, it is often applied to **optimization problems** where the goal is to identify the optimal solution from a set of potential candidates.

In the **Hill Climbing algorithm**, the **state-space diagram** is a visual representation of all possible states the search algorithm can reach, plotted against the values of the **objective function** (the function we aim to maximize).

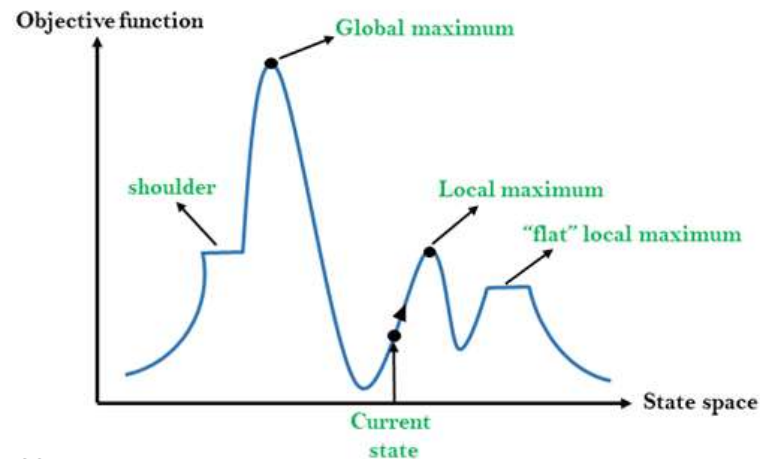
In the state-space diagram:

- **X-axis**: Represents the state space, which includes all the possible states or configurations that the algorithm can reach.
- **Y-axis**: Represents the values of the **objective function** corresponding to each state.

The optimal solution in the state-space diagram is represented by the state where the **objective function** reaches its maximum value, also known as the **global maximum**.



HILL CLIMBING



Key Regions in the State-Space Diagram

- 1. Local Maximum:** A local maximum is a state better than its neighbors but not the best overall. While its objective function value is higher than nearby states, a global maximum may still exist.
- 2. Global Maximum:** The global maximum is the best state in the state-space diagram, where the objective function achieves its highest value. This is the optimal solution the algorithm seeks.
- 3. Plateau/Flat Local Maximum:** A plateau is a flat region where neighboring states have the same objective function value, making it difficult for the algorithm to decide on the best direction to move.
- 4. Ridge:** A ridge is a higher region with a slope, which can look like a peak. This may cause the algorithm to stop prematurely, missing better solutions nearby.
- 5. Current State:** The current state refers to the algorithm's position in the state-space diagram during its search for the optimal solution.
- 6. Shoulder:** A shoulder is a plateau with an uphill edge, allowing the algorithm to move toward better solutions if it continues searching beyond the plateau.

HILL CLIMBING - TYPES

Types of Hill Climbing

1. **Simple Hill Climbing:** This is the most basic form of hill climbing. The algorithm evaluates each neighboring state in sequence and moves to the first neighbor that improves the objective function.
 - **Pros:** Simple and easy to implement.
 - **Cons:** Can get stuck in local optima easily and may require many iterations to find a better solution.

Steps:

1. **Start with an Initial Solution:** Choose an initial state (solution) randomly or based on some heuristic.
2. **Evaluate the Initial Solution:** Compute the value of the objective function for the current state.
3. **Generate Neighbors:** Generate the neighboring states of the current state.
4. **Evaluate Neighbors:** Evaluate each neighbor to see if it improves the objective function.
5. **Select the First Better Neighbor:** Move to the first neighbor that has a better (higher or lower, depending on the problem) objective function value.
6. **Repeat:** Repeat steps 3-5 until no improvement is found (i.e., no neighbor is better than the current state).



HILL CLIMBING - TYPES

2. Steepest-Ascent Hill Climbing (Gradient Ascent/Descent): Evaluates all neighbors and moves to the neighbor with the highest improvement (or lowest cost).

- **Pros:** More likely to find a better solution than simple hill climbing since it considers all possible moves.
- **Cons:** More computationally expensive as it evaluates all neighbors before making a move.

Steps:

1. **Start with an Initial Solution:** Choose an initial state (solution) randomly or based on some heuristic.
2. **Evaluate the Initial Solution:** Compute the value of the objective function for the current state.
3. **Generate Neighbors:** Generate the neighboring states of the current state.
4. **Evaluate All Neighbors:** Evaluate all neighbors to determine the one with the best (highest or lowest) objective function value.
5. **Select the Best Neighbor:** Move to the neighbor with the best objective function value.
6. **Repeat:** Repeat steps 3-5 until no neighbor improves the objective function.



HILL CLIMBING - TYPES

3. Stochastic Hill Climbing: Selects a random neighbor and moves to it if it improves the objective function. The randomness helps in exploring the search space more broadly.

- **Pros:** Can escape local optima more effectively than simple hill climbing.
- **Cons:** Less predictable and may require more iterations to converge.

Steps:

1. **Start with an Initial Solution:** Choose an initial state (solution) randomly or based on some heuristic.
2. **Evaluate the Initial Solution:** Compute the value of the objective function for the current state.
3. **Generate a Random Neighbor:** Generate a random neighboring state of the current state.
4. **Evaluate the Neighbor:** Compute the value of the objective function for the random neighbor.
5. **Accept or Reject the Neighbor:** Move to the neighbor if it has a better (higher or lower) objective function value.
6. **Repeat:** Repeat steps 3-5 until no further improvement is found.



HILL CLIMBING – LIMITATION & CHALLENGES

1. Local Optima:

- Hill Climbing is a local search algorithm, which means it tends to get stuck in local optima solutions that are better than their immediate neighbors but not necessarily the global optimum.
- This challenge arises because Hill Climbing always moves to a better neighboring solution, even if there's a much better solution further away. It lacks the ability to explore beyond the current local neighborhood.

2. Sensitivity to Initial Conditions: The effectiveness of Hill Climbing is highly dependent on the choice of the initial solution. Different initial solutions can lead to entirely different local optima or even failed convergence.

3. Lack of Global Exploration: Hill Climbing is inherently focused on improving the current solution and may miss out on better solutions in distant parts of the search space. It lacks a mechanism for global exploration, which is crucial for finding the global optimum.

4. Plateau Problem: A **plateau** is a flat region in the search space where all neighboring states have the same value. This makes it difficult for the algorithm to choose the best direction to move forward.

5. Ridge Problem: A **ridge** is a region where movement in all possible directions seems to lead downward, resembling a peak. As a result, the Hill Climbing algorithm may stop prematurely, believing it has reached the optimal solution when, in fact, better solutions exist.



BEST FIRST SEARCH

Best First Search is a heuristic search algorithm that selects the most promising node for expansion based on an evaluation function. It prioritizes nodes in the search space using a heuristic to estimate their potential. By iteratively choosing the most promising node, it aims to efficiently navigate towards the goal state, making it particularly effective for optimization problems.

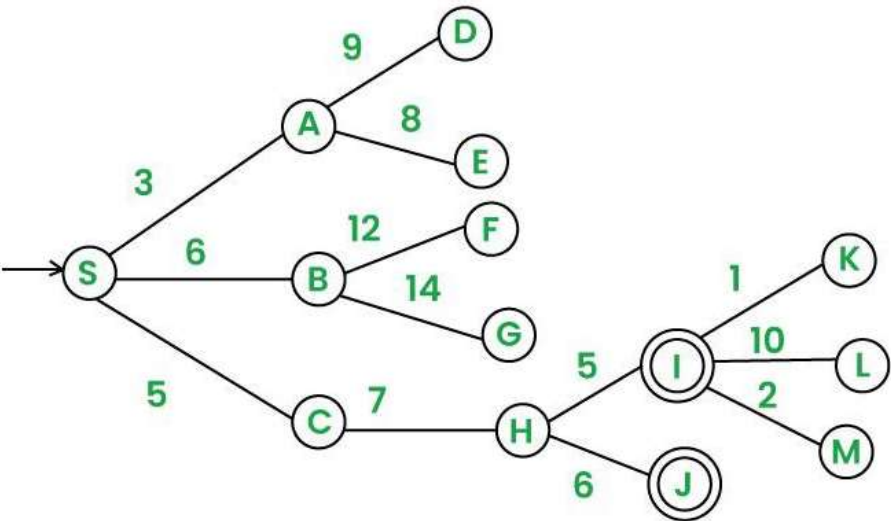
$f(n)$ = Evaluating Function (Path Cost from Node A to Node B)

Best First Search (BFS) follows a graph by using a priority queue and heuristics. It keeps an 'Open' list for nodes that need exploring and a 'Closed' list for those already checked. Here's how it operates:

1. Create 2 empty lists: **OPEN** and **CLOSED**
2. Start from the initial node (say **N**) and put it in the 'ordered' OPEN list
3. Repeat the next steps until the GOAL node is reached
 - a. If the **OPEN** list is empty, then **EXIT** the loop returning 'False'
 - b. Select the first/top node (say **N**) in the **OPEN** list and move it to the **CLOSED** list. Also, capture the information of the parent node
 - c. If **N** is a **GOAL** node, then move the node to the **CLOSED** list and exit the loop returning '**True**'. The solution can be found by backtracking the path
 - d. If **N** is not the **GOAL** node, expand node **N** to generate the '**immediate**' next nodes linked to node **N** and add all those to the **OPEN** list
 - e. Reorder the nodes in the OPEN list in ascending order according to an evaluation function $f(n)$



BEST FIRST SEARCH

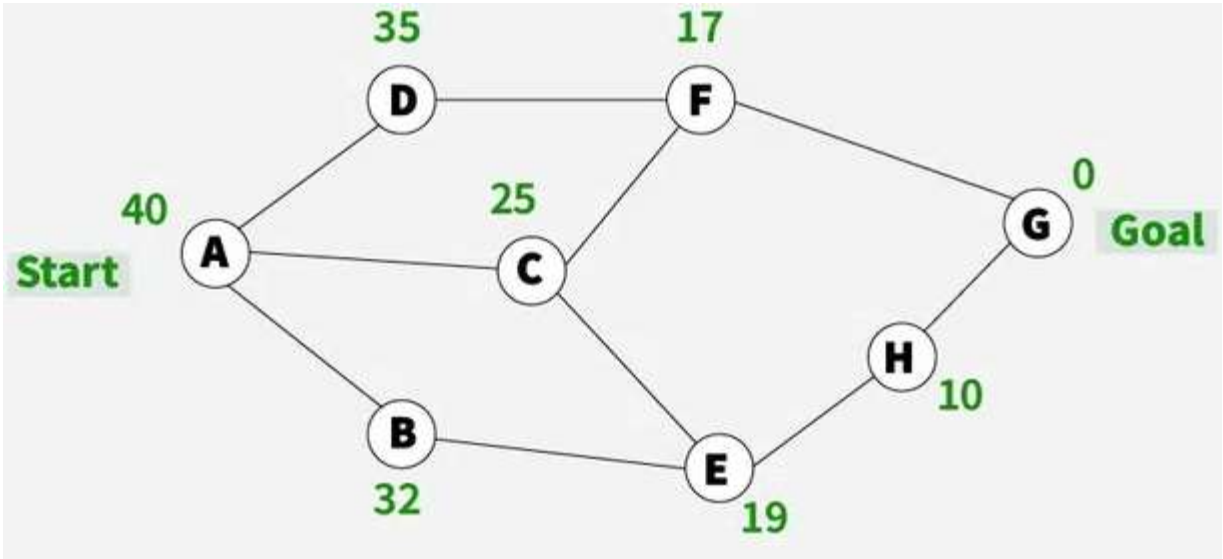


Greedy Best First Search

Evaluating Function

$$f(n) = h(n)$$

$h(n)$ = Heuristic Function



A* ALGORITHM

A* (A-Star) is a popular choice for pathfinding because it combines the strengths of Dijkstra's algorithm and Greedy Best-First Search, making it efficient and optimal. It uses a heuristic function to evaluate paths by balancing the cost to reach a node (**g-cost** [$g(n)$]) and the estimated cost to the goal (**h-cost** [$h(n)$]).

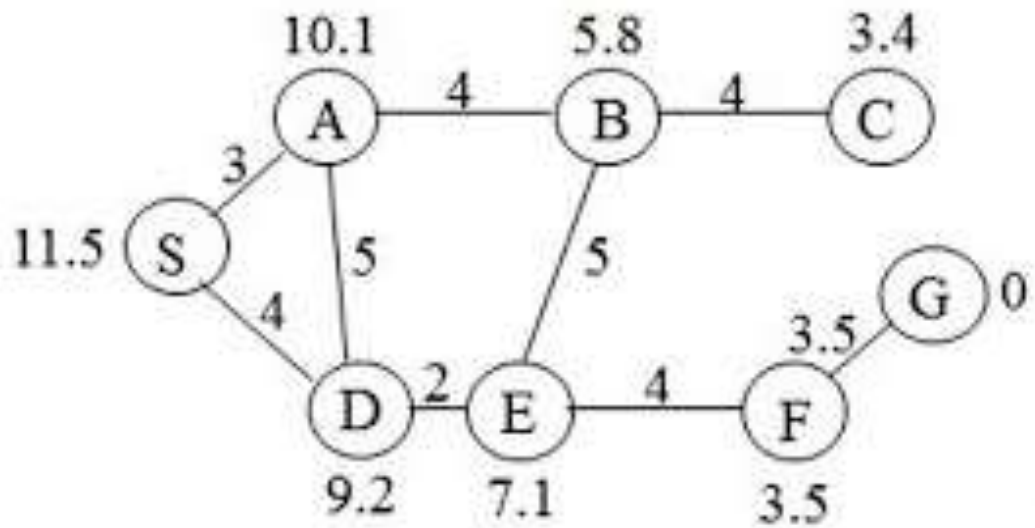
A* uses **two important** parameters to find the cost of a path:

1. **$g(n)$** : Actual cost of reaching node n from the start node. This is the accumulated cost of the path from the start node to node n .
2. **$h(n)$** : The heuristic finds of the cost to reach the goal from node n . This is a weighted guess about how much further it will take to reach the goal.

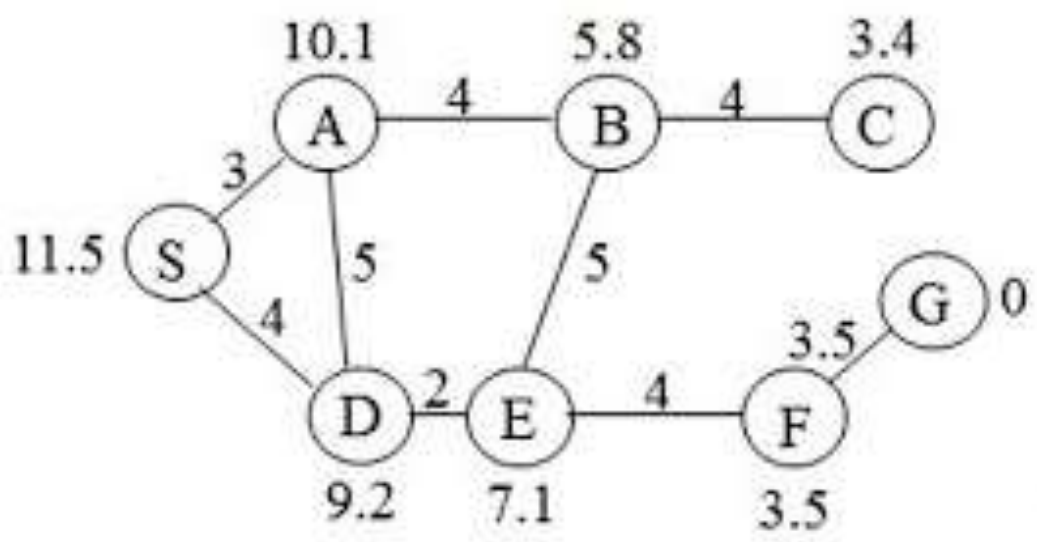
The function, **$f(n)=g(n)+h(n)$** is the total estimated cost of the cheapest solution through node n . This function combines the path cost so far and the heuristic cost to estimate the total cost guiding the search more efficiently.



A* ALGORITHM



A* ALGORITHM



Find the most cost-effective path to reach the final state from initial state using A* Algorithm. Consider $g(n)$ = Depth of node and $h(n)$ = Number of misplaced tiles.

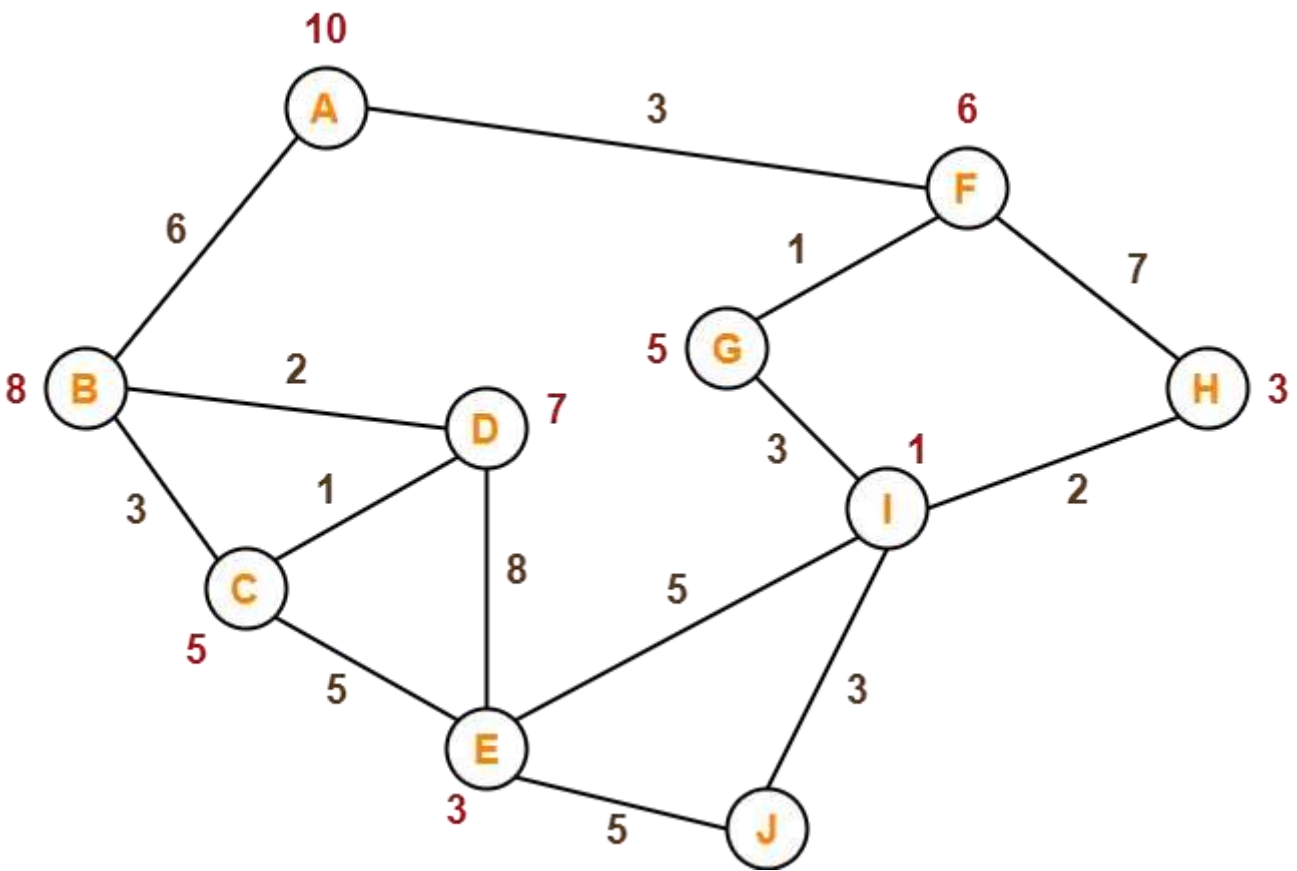
2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.



ADMISSIBILITY OF A* ALGORITHM

The cost of reaching the goal state is assessed using an admissible heuristic in an informed search algorithm, however, if we need to discover a solution to the problem, the estimated cost must be lower than or equal to the true cost of reaching the goal state. The algorithm employs the allowable heuristic to determine the best-estimated route from the current node to the objective state.

The evaluation function in A* looks like this:

$$f(n) = g(n) + h(n)$$

$$f(n) = \text{Actual cost} + \text{Estimated cost}$$

here,

n = current node.

$f(n)$ = evaluation function.

$g(n)$ = the cost from the initial node to the current node.

$h(n)$ = estimated cost from the current node to the goal state.

Conditions:

$h(n) \leq h^*(n) \therefore$ Underestimation

$h(n) \geq h^*(n) \therefore$ Overestimation

Key Points:

- The heuristic function $h(n)$ is admissible if $h(n)$ is **never larger** than $h^*(n)$ or if $h(n)$ is always less or equal to the true value.
- If A* employs an admissible heuristic and $h(\text{goal})=0$, then we can argue that **A* is admissible**.
- If the heuristic function is constantly tuned to be low with respect to the true cost, i.e. $h(n) \leq h^*(n)$, then you are going to get an optimal solution of 100%

ADMISSIBILITY OF A* ALGORITHM

Case 1:

Let's suppose, you are going to purchase shoes and shoes have a price of **\$1000**. Before making the purchase, you estimated that the shoes will be worth **\$1200**, When you went to the store, the shopkeeper informed you that the shoe's actual price was **\$1000**, which is less than your estimated value, indicating that you had overestimated their value by **\$200**. so this is the case of **Overestimation**.

$$1200 > 1000$$

$$\text{i.e. } h(n) \geq h^*(n) \therefore \text{Overestimation}$$

Case 2:

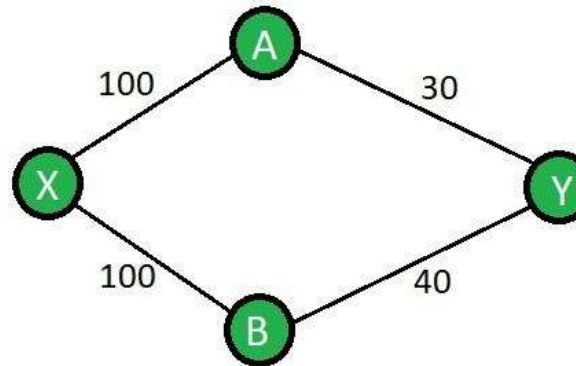
Similar to the last situation, you are planning to buy a pair of shoes. This time, you estimate the shoe value to be **\$800**. However, when you arrive at the store, the shopkeeper informs you that the shoes' true price is **\$1000**, which is higher than your estimate. Indicating that you had underestimated their value by **\$200**. In this situation, **Underestimation** has occurred.

$$800 < 1000$$

$$\text{i.e. } h(n) \leq h^*(n) \therefore \text{Underestimation}$$



ADMISSIBILITY OF A* ALGORITHM



Case 1: Overestimation

H(A)= 60, Estimated values i.e. $h(n)$

H(B)= 50

So, using A* equation, **$f(n) = G(n) + h(n)$**

by putting values

$$f(A) = 100 + 60 = 160$$

$$f(B) = 100 + 50 = 150$$

by comparing $f(A)$ & $f(B)$, $f(A) > f(B)$ so choose path of B node and apply A* equation again

$$\begin{aligned} f(Y) &= g(Y) + h(Y) \text{ [here } h(Y) \text{ is 0, Goal state]} \\ &= 140 + 0 \text{ [} g(Y)=100+40=140 \text{ this is actual cost i.e. } h^*(B) \text{]} \\ &= 140 \end{aligned}$$

Case 2: Underestimation

H(A) = 20 [This are estimated values i.e. $h(n)$]

H(B) = 10

So, using A* equation, **$f(n) = G(n) + h(n)$**

by putting values

$$f(A) = 100 + 20 = 120$$

$f(B) = 100 + 10 = 110$, by comparing $f(A)$ & $f(B)$, $f(A) > f(B)$, so choose path of B node and apply A* equation again

$$\begin{aligned} f(Y) &= g(Y) + h(Y) \text{ [here } h(Y) \text{ is 0, because it is goal state]} \\ &= 140 + 0 \text{ [} g(Y) = 100 + 40 = 140 \text{ this is actual cost i.e. } h^*(B) \text{]} \\ &= 140 \end{aligned}$$

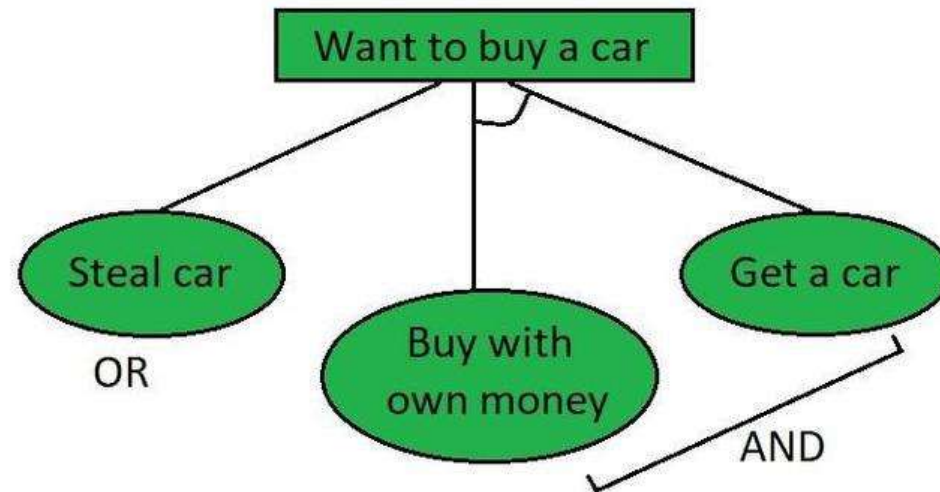
$$\begin{aligned} f(Y) &= g(Y) + h(Y) \\ &= 130 + 0 = 130 \end{aligned}$$



AO* ALGORITHM

The AO* algorithm is an advanced search algorithm utilized in artificial intelligence, particularly in problem-solving and decision-making contexts. It is an extension of the A* algorithm, designed to handle more complex problems that require handling multiple paths and making decisions at each node.

The **AO* algorithm** (short for "And-Or Star") is a powerful **best-first search** method used to solve problems that can be represented as a **directed acyclic graph (DAG)**. Unlike traditional algorithms like A*, which explore a single path, the AO* algorithm evaluates multiple paths simultaneously.



AO* ALGORITHM - WORKING PRINCIPLES

The AO* algorithm works by utilizing a tree structure where each node represents a state in the problem space. The key components of the algorithm are:

Node Types

- **AND Nodes:** Represent states where all child nodes must be satisfied to achieve a goal. If a task requires multiple conditions to be met, it would be represented as an AND node.
- **OR Nodes:** Represent states where at least one child node must be satisfied to achieve a goal. This type is useful in scenarios where multiple paths can lead to a solution.

Heuristic Function

The algorithm employs a heuristic function, similar to A*, to estimate the cost to reach the goal from any given node. This function helps in determining the most promising paths to explore. The heuristic function $h(n)$ estimates the cost to reach the goal from node n : $h(n)$ =estimated cost to reach the goal from node

- **For OR Nodes:** The algorithm considers the lowest cost among the child nodes. The cost for an OR node can be expressed as:
$$C(n)=\min\{C(c1),C(c2),\dots,C(ck)\}$$
- **For AND Nodes:** The algorithm computes the cost of all child nodes and selects the maximum cost, as all conditions must be met. The cost for an AND node can be expressed as:
$$C(n)=\max\{C(c1),C(c2),\dots,C(ck)\}$$

Total Estimated Cost: $f(n)=C(n)+h(n)$

- $C(n)$ is the actual cost to reach node n from the start node.
- $h(n)$ is the estimated cost from node n to the goal.

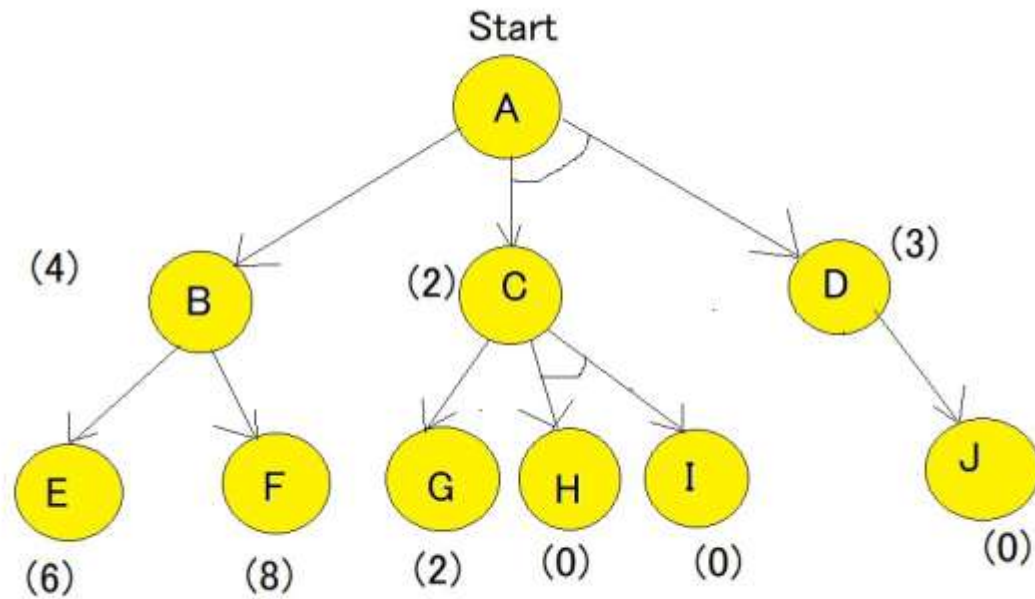


AO* ALGORITHM

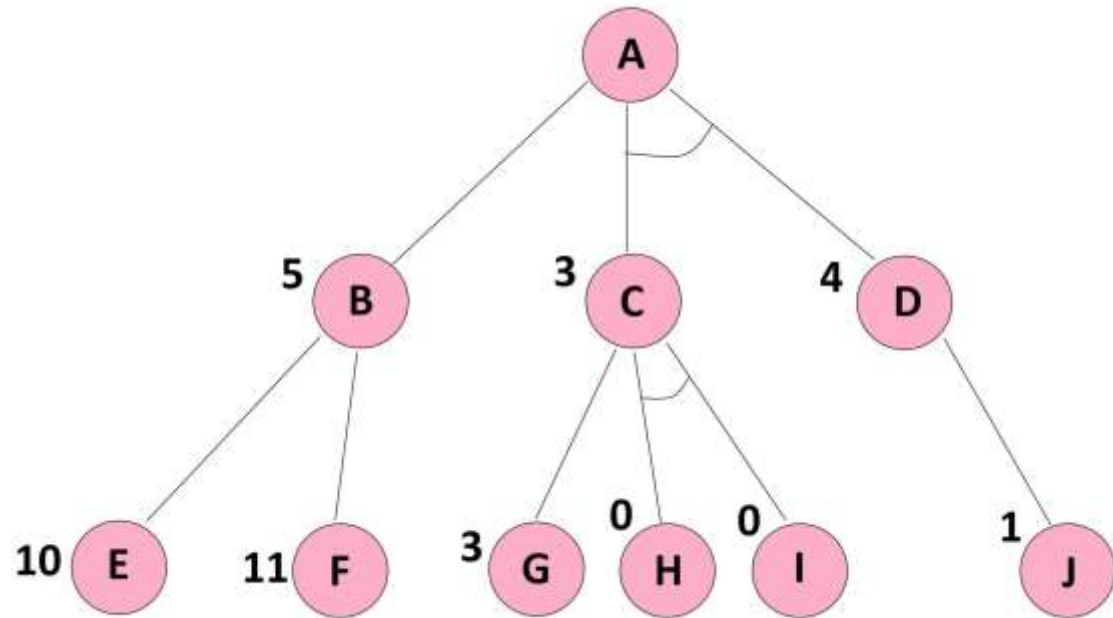
1. Initialise the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value FUTILITY otherwise calculate only f' for each of the successors.
4. If f' is 0 then mark the node as SOLVED
5. Change the value of f' for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as SOLVED then mark the parent node as SOLVED.
7. If starting node is SOLVED or value greater than FUTILITY, stop, else repeat from 2.



AO* ALGORITHM - WORKING PRINCIPLES



AO* ALGORITHM - WORKING PRINCIPLES



MIN-MAX ALGORITHM

Min-Max algorithm is a decision-making algorithm used in artificial intelligence, particularly in game theory and computer games. It is designed to minimize the possible loss in a worst-case scenario (hence "min") and maximize the potential gain (therefore "max").

Working of Min-Max Process in AI

Min-Max algorithm involves two players: the maximizer and the minimizer, each aiming to optimize their own outcomes.

Players Involved

Maximizing Player (Max):

- Aims to maximize their score or utility value.
- Chooses the move that leads to the highest possible utility value, assuming the opponent will play optimally.

Minimizing Player (Min):

- Aims to minimize the maximizer's score or utility value.
- Selects the move that results in the lowest possible utility value for the maximizer, assuming the opponent will play optimally.



MIN-MAX ALGORITHM

The Min-Max algorithm involves several key steps, executed recursively until the optimal move is determined. Here is a step-by-step breakdown:

Step 1: Generate the Game Tree

- **Objective:** Create a tree structure representing all possible moves from the current game state.
- **Details:** Each node represents a game state, and each edge represents a possible move.

Step 2: Evaluate Terminal States

- **Objective:** Assign utility values to the terminal nodes of the game tree.
- **Details:** These values represent the outcome of the game (win, lose, or draw).

Step 3: Propagate Utility Values Upwards

- **Objective:** Starting from the terminal nodes, propagate the utility values upwards through the tree.
- **Details:** For each non-terminal node:
 - If it's the maximizing player's turn, select the maximum value from the child nodes.
 - If it's the minimizing player's turn, select the minimum value from the child nodes.

Step 4: Select Optimal Move

- **Objective:** At the root of the game tree, the maximizing player selects the move that leads to the highest utility value.



MIN-MAX ALGORITHM

Min-Max Formula

The Min-Max value of a node in the game tree is calculated using the following recursive formulas:

1. Maximizing Player's Turn:

$$Max(s) = \max_{a \in A(s)} Max(Result(s, a))$$

Here:

- $Max(s)$ is the maximum value the maximizing player can achieve from state s .
- $A(s)$ is the set of all possible actions from state s .
- $Result(s, a)$ is the resulting state from taking action a in state s .
- $Min(Result(s, a))$ is the value for the minimizing player from the resulting state.

Minimizing Player's Turn:

$$Min(s) = \min_{a \in A(s)} Max(Result(s, a))$$

Here:

- $Min(s)$ is the minimum value the minimizing player can achieve from state s .
- The other terms are similar to those defined above.

Terminal States

For terminal states, the utility value is directly assigned:

$$Utility(s) = \begin{cases} 1 & \text{if the maximizing player wins from state } s \\ 0 & \text{if the game is draw from state } s \\ -1 & \text{if the minimizing player wins from state } s \end{cases}$$



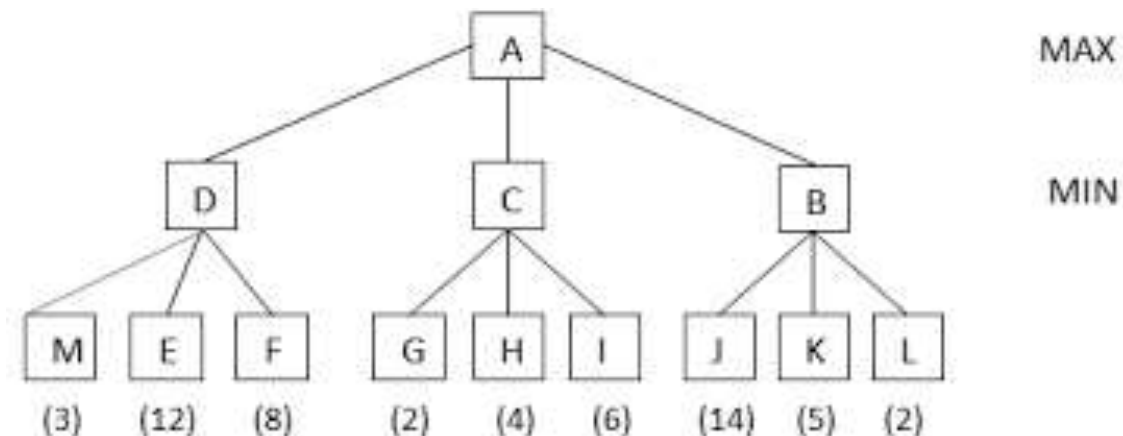
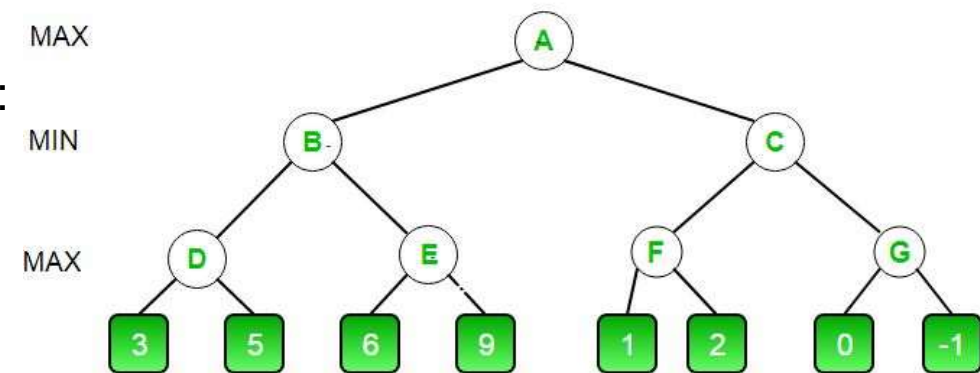
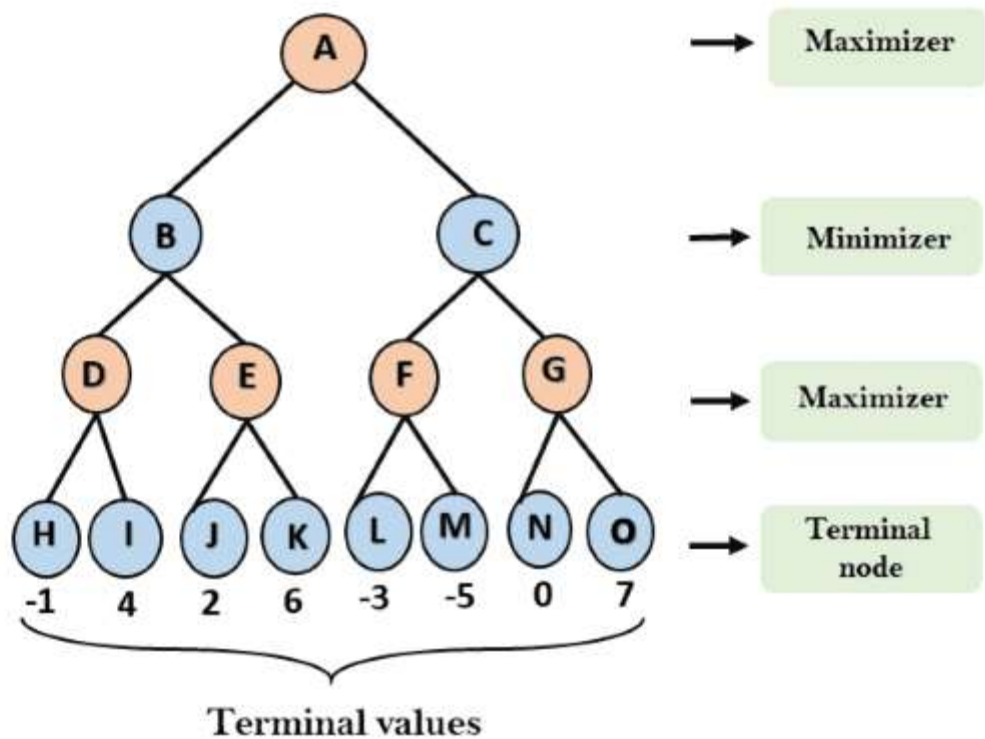
MIN-MAX ALGORITHM

Consider a simple game where the utility values of terminal states are given. To illustrate the Min-Max calculations:

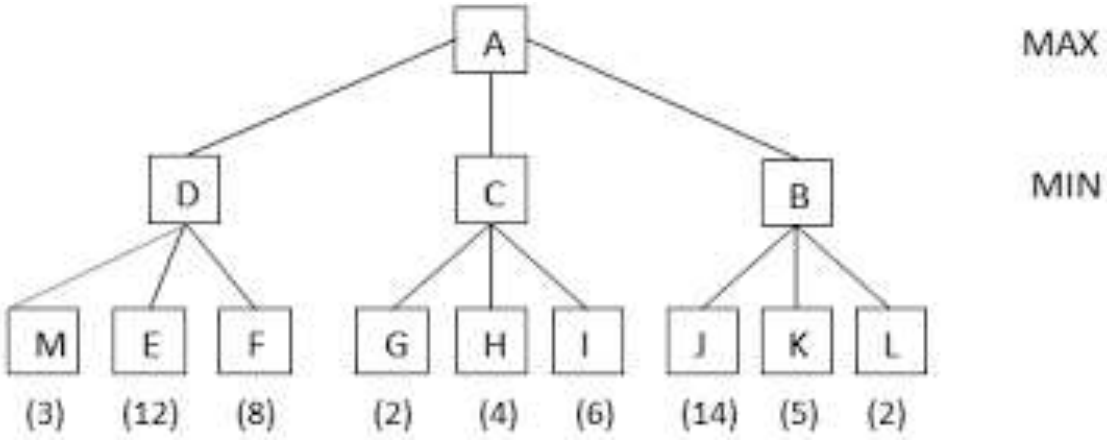
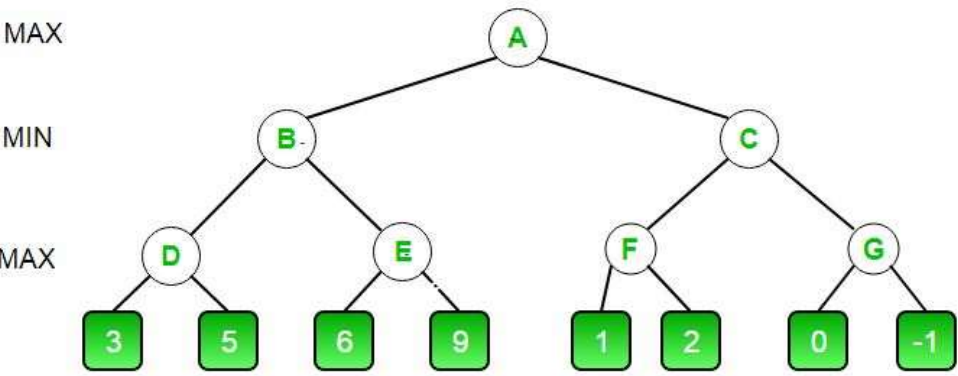
1. Start from the terminal states and calculate the utility values.
2. Propagate these values up the tree using the Min-Max formulas.

For example, if the terminal states have utility values U_1, U_2, \dots, U_n then:

- For the maximizing player's node: $Max(s) = \max(U_1, U_2, \dots, U_n)$
- For the minimizing player's node: $Min(s) = \min(U_1, U_2, \dots, U_n)$



MIN-MAX ALGORITHM



ALPHA BETA PRUNING

The key idea behind Alpha Beta Pruning is to avoid evaluating branches of the game tree that cannot influence the final decision based on the values already discovered during the search. It achieves this using two values: **Alpha** and **Beta**.

The critical points of Alpha Beta Pruning in AI are as follows.

The initialization of the parameters

- $Alpha(\alpha)$ is initialized with $-\infty$.
- $Beta(\beta)$ is initialized with $+\infty$.

Updating the parameters

- $Alpha(\alpha)$ is updated only by the maximizer at its turn.
- $Beta(\beta)$ is updated only by the minimizer at its turn.

Passing the parameters

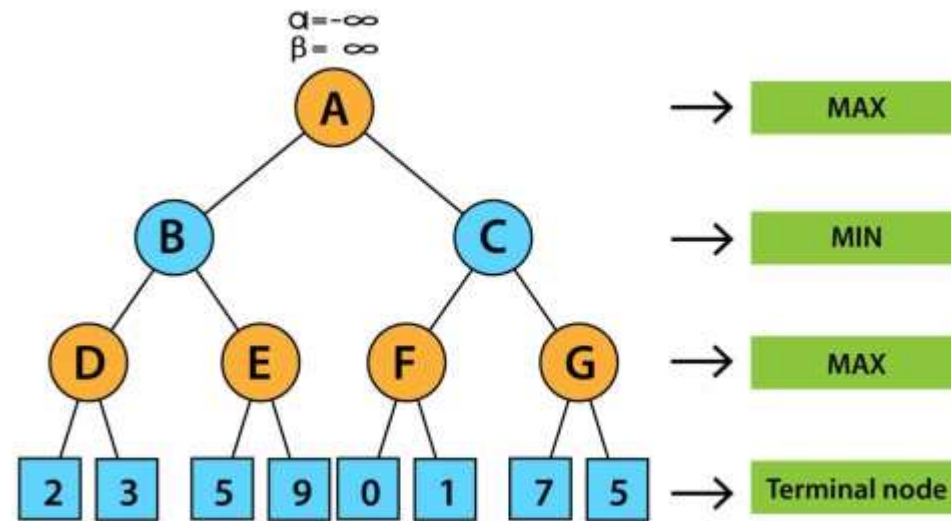
- $Alpha(\alpha)$ and $Beta(\beta)$ are passed on to only child nodes.
- While backtracking the game tree, the node values are passed to parent nodes.

Pruning Condition

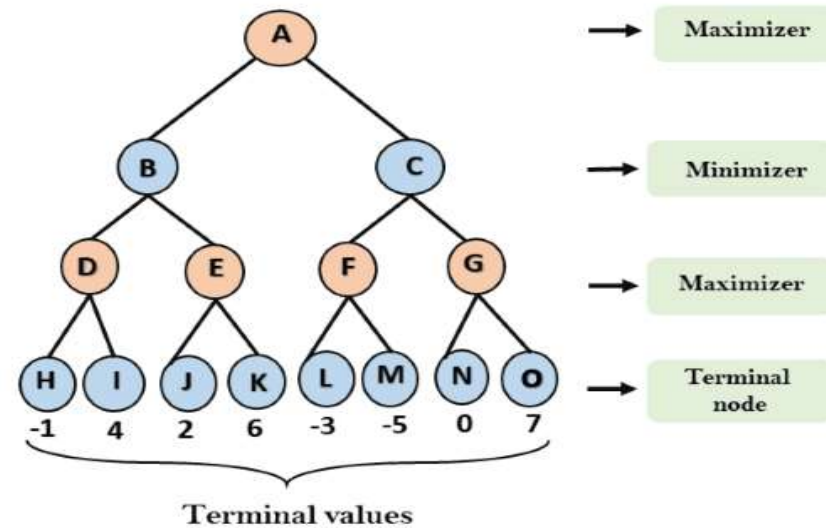
- The child sub-trees, which are not yet traversed, are pruned if the condition $\alpha \geq \beta$ holds.



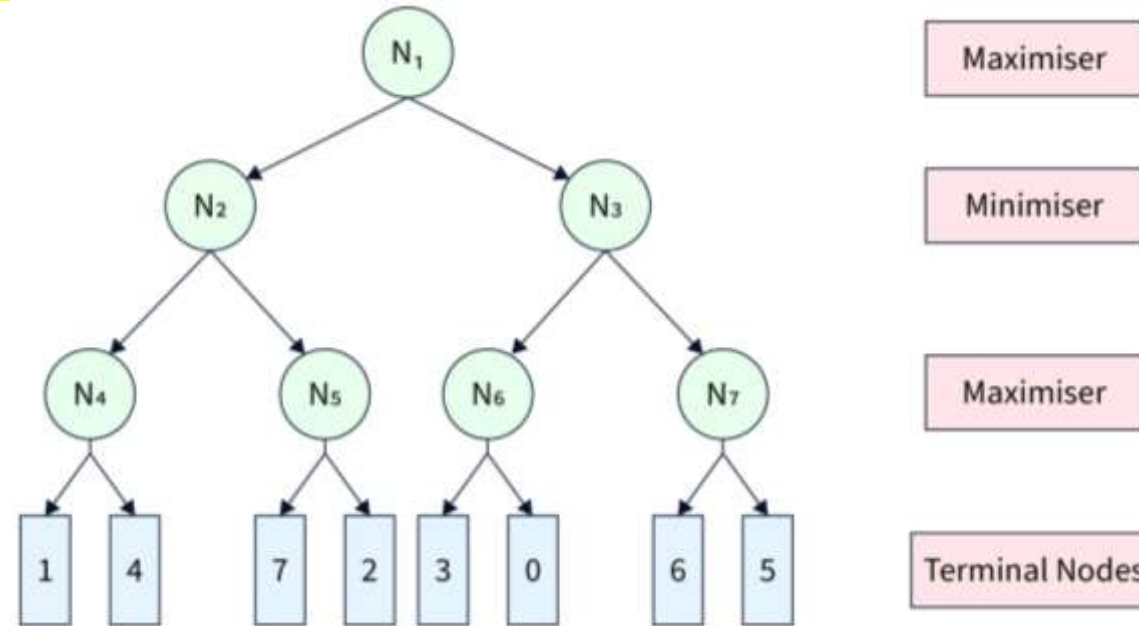
ALPHA BETA PRUNING



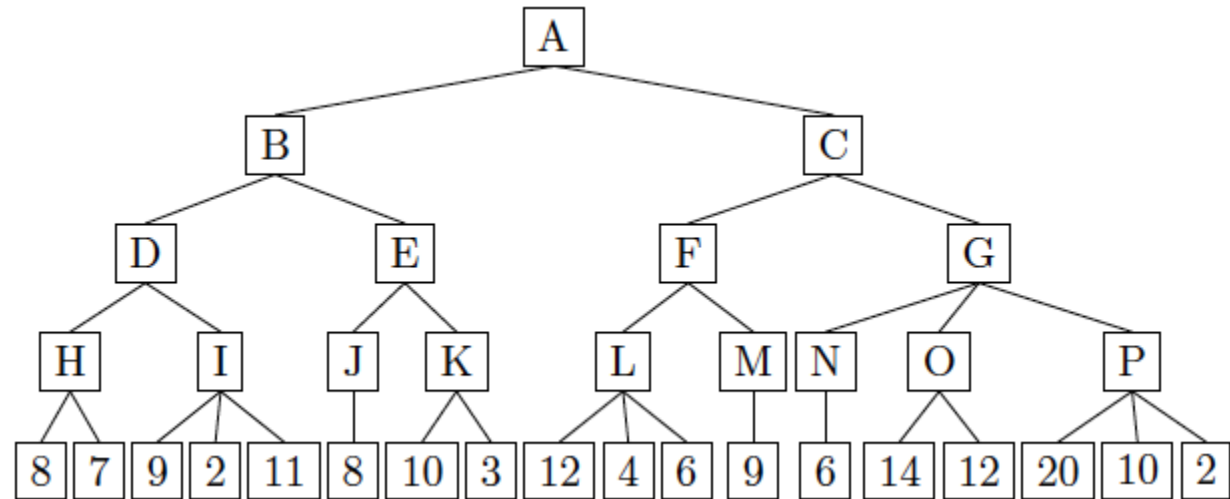
ALPHA BETA PRUNING



ALPHA BETA PRUNING



ALPHA BETA PRUNING



SYMBOLIC LOGIC

PROPOSITIONAL LOGIC IN ARTIFICIAL INTELLIGENCE

Logic is concerned with reasoning and the validity of arguments. In general, in logic, we are not concerned with the truth of statements, but rather with their validity. That is to say, although the following argument is clearly logical, it is not something that we would consider to be true:

- All lemons are blue
- Mary is a lemon
- Therefore, Mary is blue

This set of statements is considered to be valid because the conclusion (Mary is blue) follows logically from the other two statements, which we often call the premises.

1. Propositional Logic
2. Predicate Logic/ First order Logic

PROPOSITIONAL LOGIC IN ARTIFICIAL INTELLIGENCE

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

- a) It is Sunday.
- b) The Sun rises from West (False proposition)
- c) $3+3=7$ (False proposition)
- d) 5 is a prime number.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

1. Atomic Propositions

2. Compound propositions

- **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

- a) $2+2$ is 4, it is an atomic proposition as it is a **true** fact.
- b) "The Sun is cold" is also a proposition as it is a **false** fact.
- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

- a) "It is raining today, and street is wet."
- b) "Ankit is a doctor, and his clinic is in Mumbai."

LOGICAL CONNECTIVES

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

- 1. Negation:** A sentence such as $\neg P$ is called negation of P. A literal can be either Positive literal or negative literal.
- 2. Conjunction:** A sentence which has \wedge connective such as, $P \wedge Q$ is called a conjunction.
Example: Rohan is intelligent and hardworking. It can be written as,
P= Rohan is intelligent, Q= Rohan is hardworking. $\rightarrow P \wedge Q$.
- 3. Disjunction:** A sentence which has \vee connective, such as $P \vee Q$. is called disjunction, where P and Q are the propositions.
Example: "Ritika is a doctor or Engineer"
Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as $P \vee Q$.
- 4. Implication:** A sentence such as $P \rightarrow Q$, is called an implication. Implications are also known as if-then rules. It can be represented as
If it is raining, then the street is wet.
Let P= It is raining, and Q= Street is wet, so it is represented as $P \rightarrow Q$
- 5. Biconditional:** A sentence such as $P \Leftrightarrow Q$ is a Biconditional sentence, example If I am breathing, then I am alive
P= I am breathing, Q= I am alive, it can be represented as $P \Leftrightarrow Q$.

LOGICAL CONNECTIVES

Connective symbols	Word	Technical term	Example
\wedge	AND	Conjunction	$A \wedge B$
\vee	OR	Disjunction	$A \vee B$
\rightarrow	Implies	Implication	$A \rightarrow B$
\Leftrightarrow	If and only if	Biconditional	$A \Leftrightarrow B$
\neg or \sim	Not	Negation	$\neg A$ or $\neg B$

For Negation:

P	$\neg P$
True	False
False	True

For Conjunction:

P	Q	$P \wedge Q$
True	True	True
True	False	False
False	True	False
False	False	False

For disjunction:

P	Q	$P \vee Q$
True	True	True
False	True	True
True	False	True
False	False	False

For Implication:

P	Q	$P \rightarrow Q$
True	True	True
True	False	False
False	True	True
False	False	True

For Biconditional:

P	Q	$P \Leftrightarrow Q$
True	True	True
True	False	False
False	True	False
False	False	True

PRECEDENCE OF CONNECTIVES

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

PROPERTIES OF OPERATORS

- **Commutativity:**
 - $P \wedge Q = Q \wedge P$, or
 - $P \vee Q = Q \vee P$.
- **Associativity:**
 - $(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$,
 - $(P \vee Q) \vee R = P \vee (Q \vee R)$
- **Identity element:**
 - $P \wedge \text{True} = P$,
 - $P \vee \text{True} = \text{True}$.
- **Distributive:**
 - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$.
 - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$.
- **DE Morgan's Law:**
 - $\neg (P \wedge Q) = (\neg P) \vee (\neg Q)$
 - $\neg (P \vee Q) = (\neg P) \wedge (\neg Q)$.
- **Double-negation elimination:**
 - $\neg (\neg P) = P$.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic. Example:
 - **All the girls are intelligent.**
 - **Some apples are sweet.**
- Propositional logic has limited expressive power.
- In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

RULES OF INFERENCE

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, so **generating the conclusions from evidence and facts is termed as Inference.**

Inference rules

Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:

- **Implication:** It is one of the logical connectives which can be represented as $P \rightarrow Q$. It is a Boolean expression.
- **Converse:** The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as $Q \rightarrow P$.
- **Contrapositive:** The negation of converse is termed as contrapositive, and it can be represented as $\neg Q \rightarrow \neg P$.
- **Inverse:** The negation of implication is called inverse. It can be represented as $\neg P \rightarrow \neg Q$.

RULES OF INFERENCE

From the above term some of the compound statements are equivalent to each other, which we can prove using truth table:

P	Q	$P \rightarrow Q$	$Q \rightarrow P$	$\neg Q \rightarrow \neg P$	$\neg P \rightarrow \neg Q$
T	T	T	T	T	T
T	F	F	T	F	T
F	T	T	F	T	F
F	F	T	T	T	T

RULES OF INFERENCE

Types of Inference rules:

1. Modus Ponens:

The Modus Ponens rule is one of the most important rules of inference, and it states that if P and $P \rightarrow Q$ is true, then we can infer that Q will be true. It can be represented as:

$$\text{Notation for Modus ponens: } \frac{P \rightarrow Q, P}{\therefore Q}$$

Example:

Statement-1: "If I am sleepy then I go to bed" $\implies P \rightarrow Q$

Statement-2: "I am sleepy" $\implies P$

Conclusion: "I go to bed." $\implies Q$.

Hence, we can say that, if $P \rightarrow Q$ is true and P is true then Q will be true.

2. Modus Tollens:

The Modus Tollens rule state that if $P \rightarrow Q$ is true and $\neg Q$ is true, then $\neg P$ will also true. It can be represented as:

$$\text{Notation for Modus Tollens: } \frac{P \rightarrow Q, \neg Q}{\neg P}$$

Statement-1: "If I am sleepy then I go to bed" $\implies P \rightarrow Q$

Statement-2: "I do not go to the bed." $\implies \neg Q$

Statement-3: Which infers that **"I am not sleepy"** $\implies \neg P$

RULES OF INFERENCE

3. Hypothetical Syllogism:

The Hypothetical Syllogism rule state that if $P \rightarrow R$ is true whenever $P \rightarrow Q$ is true, and $Q \rightarrow R$ is true. It can be represented as the following notation:

Example:

Statement-1: If you have my home key then you can unlock my home. $P \rightarrow Q$

Statement-2: If you can unlock my home then you can take my money. $Q \rightarrow R$

Conclusion: If you have my home key then you can take my money. $P \rightarrow R$

4. Disjunctive Syllogism:

The Disjunctive syllogism rule state that if $P \vee Q$ is true, and $\neg P$ is true, then Q will be true. It can be represented as:

$$\text{Notation of Disjunctive syllogism: } \frac{P \vee Q, \neg P}{Q}$$

Example:

Statement-1: Today is Sunday or Monday. $\Rightarrow P \vee Q$

Statement-2: Today is not Sunday. $\Rightarrow \neg P$

Conclusion: Today is Monday. $\Rightarrow Q$

RULES OF INFERENCE

5. Addition:

The Addition rule is one the common inference rule, and it states that If P is true, then $P \vee Q$ will be true.

$$\text{Notation of Addition: } \frac{P}{P \vee Q}$$

Example:

Statement: I have a vanilla ice-cream. $\implies P$

Statement-2: I have Chocolate ice-cream.

Conclusion: I have vanilla or chocolate ice-cream. $\implies (P \vee Q)$


6. Simplification:

The simplification rule state that if $P \wedge Q$ is true, then **Q or P** will also be true. It can be represented as:

$$\text{Notation of Simplification rule: } \frac{P \wedge Q}{Q} \text{ Or } \frac{P \wedge Q}{P}$$

Proof by Truth-Table:

P	Q	$P \wedge Q$
0	0	0
1	0	0
0	1	0
1	1	1



7. Resolution:

The Resolution rule state that if $P \vee Q$ and $\neg P \wedge R$ is true, then $Q \vee R$ will also be true. **It can be represented as**

$$\text{Notation of Resolution } \frac{P \vee Q, \neg P \wedge R}{Q \vee R}$$

TRANSLATING BETWEEN ENGLISH AND LOGIC NOTATION

1. “It is raining and it is Tuesday.”

might be expressed as: $R \wedge T$,

Where R means “it is raining” and T means “it is Tuesday.”

2. “it is raining in New York” or “it is raining heavily” or even “it rained for 30 minutes on Thursday” , then R will probably not suffice.

To express more complex concepts like these, we usually use **predicates**.

Hence, for example, we might translate

“it is raining in New York” as: $N(R)$ We might equally well choose to write it as: $R(N)$

- When we write $N(R)$, we are saying that a property of the rain is that it is in New York, whereas with $R(N)$ we are saying that a property of New York is that it is raining. Which we use depends on the problem we are solving.
- It is likely that if we are solving a problem about New York, we would use $R(N)$, whereas if we are solving a problem about the location of various types of weather, we might use $N(R)$.

TRANSLATING BETWEEN ENGLISH AND LOGIC NOTATION

Jeffery is happy.

Solomon and Kevin are both dogs.

Carlos is happier than Sue, but sadder than Fred.

James is a troublemaker when Kevin dislikes him.

Whenever he eats sandwiches that have pickles in them, he ends up either asleep at his desk or singing loud songs

FIRST-ORDER LOGIC (PREDICATE LOGIC)

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
 - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, Wumpus,
 - **Relations:** It can be **unary relation such as:** red, round, is adjacent, or **n-any relation such as:** the sister of, brother of, has color, comes between
 - **Function:** Father of, best friend, third inning of, end of,
- As a natural language, first-order logic also has two main parts:
 - **Syntax**
 - **Semantics**

FIRST-ORDER LOGIC (PREDICATE LOGIC)

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf,
Connectives	\wedge , \vee , \neg , \Rightarrow , \Leftrightarrow
Equality	$=$
Quantifier	\forall , \exists

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2,, term n)**.

Example: Ravi and Ajay are brothers: \Rightarrow Brothers(Ravi, Ajay).

Chinky is a cat: \Rightarrow cat (Chinky).

Complex Sentences:

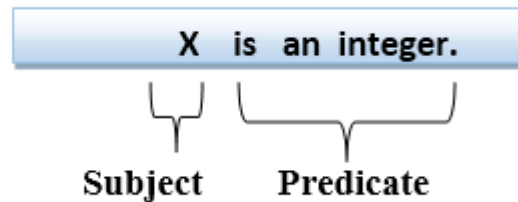
- Complex sentences are made by combining atomic sentences using connectives.

FIRST-ORDER LOGIC (PREDICATE LOGIC)

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
 - **Universal Quantifier, (for all, everyone, everything)**
 - **Existential quantifier, (for some, at least one).**

FIRST-ORDER LOGIC (PREDICATE LOGIC)

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol \forall , which resembles an inverted A.

Note: In universal quantifier we use implication " \rightarrow ".

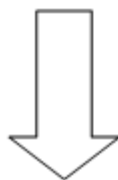
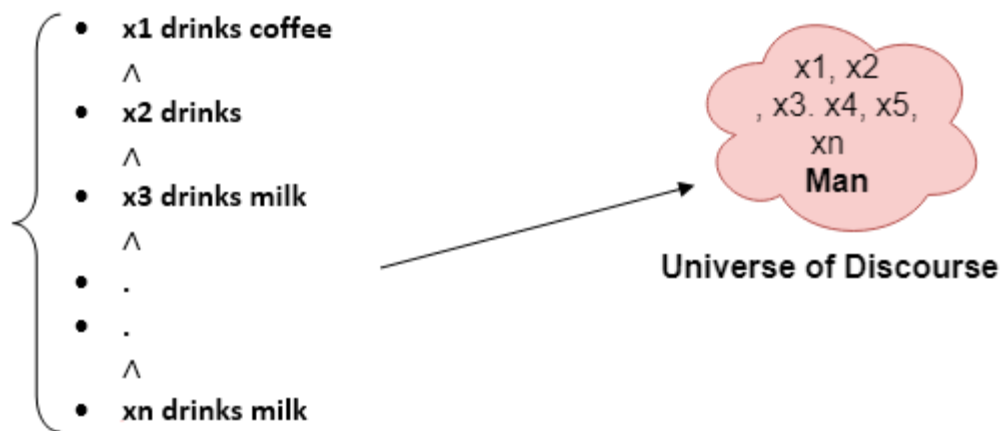
If x is a variable, then $\forall x$ is read as:

- For all x
- For each x
- For every x .

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as seen:



So in shorthand notation, we can write it as :

$\forall x \text{ man}(x) \rightarrow \text{drink}(x, \text{coffee}).$

FIRST-ORDER LOGIC (PREDICATE LOGIC)

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator \exists , which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

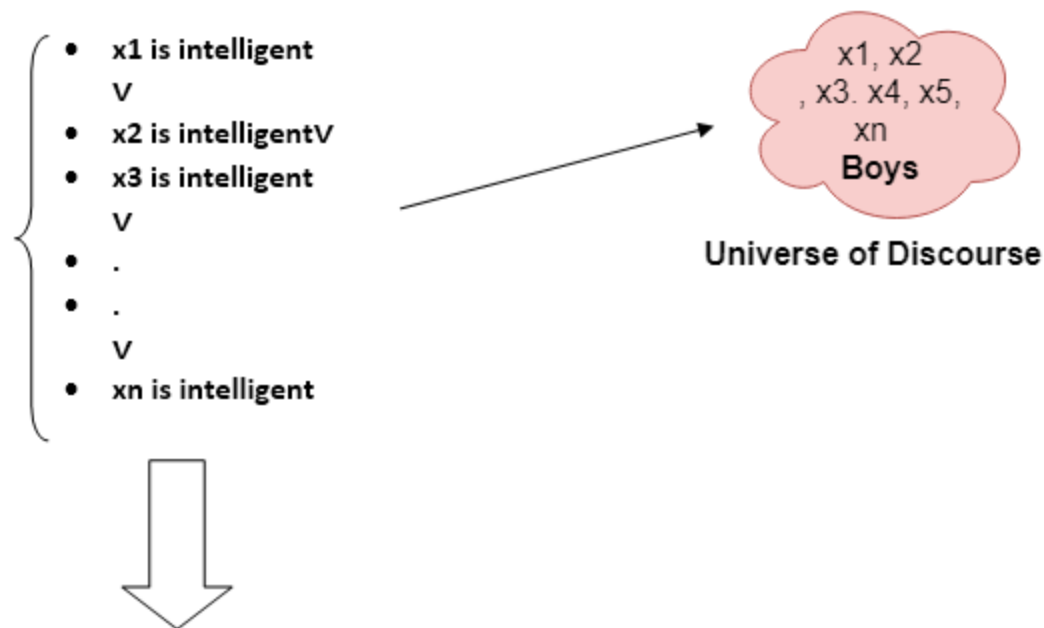
Note: In Existential quantifier we always use AND or Conjunction symbol (\wedge).

If x is a variable, then existential quantifier will be $\exists x$ or $\exists(x)$. And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some boys are intelligent.



So in short-hand notation, we can write it as:

$\exists x: \text{boys}(x) \wedge \text{intelligent}(x)$

It will be read as:

There are some x where x is a boy who is intelligent.

FIRST-ORDER LOGIC (PREDICATE LOGIC)

1. All birds fly.

In this question the predicate is "**fly(bird).**"

And since there are all birds who fly so it will be represented as follows.

$$\forall x \text{ bird}(x) \rightarrow \text{fly}(x).$$

2. Every man respects his parent.

In this question, the predicate is "**respect(x, y),**" where **x=man, and y= parent.**

Since there is every man so will use \forall , and it will be represented as follows:

$$\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent}).$$

3. Some boys play cricket.

In this question, the predicate is "**play(x, y),**" where **x= boys, and y= game.** Since there are some boys so we will use \exists , and it will be represented as:

$$\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket}).$$

FIRST-ORDER LOGIC (PREDICATE LOGIC)

4. Not all students like both Mathematics and Science.

In this question, the predicate is "**like(x, y),**" where **x= student, and y= subject.**

Since there are not all students, so we will use **\forall with negation**, so following representation for this:

$$\neg \forall (x) [\text{student}(x) \rightarrow \text{like}(x, \text{Mathematics}) \wedge \text{like}(x, \text{Science})].$$

5. Only one student failed in Mathematics.

In this question, the predicate is "**failed(x, y),**" where **x= student, and y= subject.**

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$$\exists (x) [\text{student}(x) \rightarrow \text{failed}(x, \text{Mathematics}) \wedge \forall (y) [\neg (x=y) \wedge \text{student}(y) \rightarrow \neg \text{failed}(y, \text{Mathematics})]].$$

INFERENCE IN FIRST-ORDER LOGIC

Substitution:

Substitution is a basic procedure that is applied to terms and formulations. It can be found in all first-order logic inference systems. When there are quantifiers in FOL, the substitution becomes more complicated. When we write $F[a/x]$, we are referring to the substitution of a constant "a" for the variable "x."

[Note: first-order logic can convey facts about some or all of the universe's objects.]

Equality:

In First-Order Logic, atomic sentences are formed not only via the use of predicate and words, but also through the application of equality. We can do this by using **equality symbols**, which indicate that the two terms relate to the same thing.

Example: Brother (John) = Smith.

In the above example, the object referred by the **Brother (John)** is close to the object referred by **Smith**. The equality symbol can be used with negation to portray that two terms are not the same objects.

Example: $\neg(x=y)$ which is equivalent to $x \neq y$.

INFERENCE IN FIRST-ORDER LOGIC

FOL inference rules for quantifier:

First-order logic has inference rules similar to propositional logic, therefore here are some basic inference rules in FOL:

1. Universal Generalization:

- Universal generalization is a valid inference rule that states that if premise $P(c)$ is true for any arbitrary element c in the universe of discourse, we can arrive at the conclusion $\forall x P(x)$.
- It can be represented as:

$$\frac{P(c)}{\forall x P(x)}$$

- If we want to prove that every element has a similar property, we can apply this rule.
- x must not be used as a free variable in this rule.

Example: Let's represent, $P(c)$: "A byte contains 8 bits", so "All bytes contain 8 bits." for $\forall x P(x)$, it will also be true.

INFERENCE IN FIRST-ORDER LOGIC

FOL inference rules for quantifier:

First-order logic has inference rules similar to propositional logic, therefore here are some basic inference rules in FOL:

2. Universal Instantiation:

- A valid inference rule is universal instantiation, often known as universal elimination or UI. It can be used to add additional sentences many times.
- The new knowledge base is logically equal to the existing knowledge base.
- **We can infer any phrase by replacing a ground word for the variable**, according to UI
- The UI rule say that we can infer any sentence $P(c)$ by substituting a ground term c (a constant within domain x) from $\forall x P(x)$ **for any object in the universe of discourse**.
- It can be represented as

$$\frac{P(c)}{\forall x P(x)}$$

Example: 1 IF "Every person like ice-cream" $\Rightarrow \forall x P(x)$ so we can infer that "John likes ice-cream" $\Rightarrow P(c)$

INFERENCE IN FIRST-ORDER LOGIC

Example: 2 Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL: $\forall x \text{ king}(x) \wedge \text{greedy}(x) \rightarrow \text{Evil}(x)$,

We can infer any of the following statements using Universal Instantiation from this information:

- $\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \rightarrow \text{Evil}(\text{John})$,
- $\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \rightarrow \text{Evil}(\text{Richard})$,
- We can infer any phrase by replacing a ground word for the variable, according to UI
- $\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \rightarrow \text{Evil}(\text{Father}(\text{John}))$,

INFERENCE IN FIRST-ORDER LOGIC

FOL inference rules for quantifier:

First-order logic has inference rules similar to propositional logic, therefore here are some basic inference rules in FOL:

3. Existential Instantiation:

- Existential instantiation is also known as Existential Elimination, and it is a legitimate first-order logic inference rule.
- It can only be used to replace the existential sentence once.
- Although the new KB is not conceptually identical to the old KB, it will be satisfiable if the old KB was.
- This rule states that for a new constant symbol c , one can deduce $P(c)$ from the formula given in the form of $\exists x P(x)$.
- The only constraint with this rule is that c must be a new word for which $P(c)$ is true.
- It's written like this:

$$\frac{\exists x P(x)}{P(c)}$$

INFERENCE IN FIRST-ORDER LOGIC

Example: 1

From the given sentence: $\exists x \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$,

So we can infer: $\text{Crown}(K) \wedge \text{OnHead}(K, \text{John})$, as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is known as **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

1. An existential generalization is a valid inference rule in first-order logic that is also known as an existential introduction.

2. This rule argues that if some element c in the universe of discourse has the property P , we can infer that something in the universe has the attribute P .

3. It's written like this:

$$\frac{P(c)}{\exists x P(x)}$$

Example: Let's say that,

"Priyanka got good marks in English."

"Therefore, someone got good marks in English."

FIRST-ORDER LOGIC (PREDICATE LOGIC)

Free and Bound Variables

There are two types of variables based upon their interaction with the quantifiers in a First Order Logic in AI, namely free and bound variables.

1. Free Variables:

Free variables are those variables that do not come under the scope of the quantifier. For instance, in an expression $\forall x \exists y P(x, y, z)$, z is a free variable because it doesn't come under the scope of any quantifier.

2. Bound Variables:

Bound variables are those variables that occur inside the scope of the quantifier. For instance, in an expression $\forall x \exists y P(x, y, z)$, x and y are bound variables because they occur inside the scope of the quantifiers.

FIRST-ORDER LOGIC (PREDICATE LOGIC)

Convert the following sentences to Predicate Logic

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Pompeians were either loyal to Caesar or hated him.
6. Every one is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.
9. All men are people.

Using Predicate Logic to prove that,
Was Marcus loyal to Caesar?

FIRST-ORDER LOGIC (PREDICATE LOGIC)

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Pompeians were either loyal to Caesar or hated him.
6. Every one is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.
9. All men are people.

FIRST-ORDER LOGIC (PREDICATE LOGIC)

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\forall x: Pompeian(x) \rightarrow Roman(x)$
4. $ruler(Caesar)$
5. $\forall x: Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. $\forall x: \exists y: loyalto(x, y)$
7. $\forall x: \forall y: person(x) \wedge ruler(y) \wedge tryassassinate(x, y) \rightarrow \neg loyalto(x, Caesar)$
8. $tryassassinate(Marcus, Caesar)$
9. $\forall x: man(x) \rightarrow person(x)$

FIRST-ORDER LOGIC (PREDICATE LOGIC)

$$\begin{array}{c} \text{nil} \\ \downarrow (1) \\ \text{man}(\text{Marcus}) \\ \downarrow (9) \\ \text{person}(\text{Marcus}) \\ \downarrow (8) \\ \text{person}(\text{Marcus}) \wedge \text{tryassassinate}(\text{Marcus}, \text{Caesar}) \\ \downarrow (4) \\ \text{person}(\text{Marcus}) \wedge \text{tryassassinate}(\text{Marcus}, \text{Caesar}) \wedge \text{ruler}(\text{Caesar}) \\ \downarrow (7, \text{Substitution}) \\ \neg \text{loyalto}(\text{Marcus}, \text{Caesar}) \end{array}$$

FIRST-ORDER LOGIC (PREDICATE LOGIC)

2. Was Marcus hates Caesar?

$\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$

$\downarrow (2)$

$\text{Pompeian}(\text{Marcus}) \neg \text{loyalto}(\text{Marcus}, \text{Caesar})$

$\downarrow (3)$

$\text{Roman}(\text{Marcus}) \neg \text{loyalto}(\text{Marcus}, \text{Caesar})$

$\downarrow (53)$

$\text{hate}(\text{Marcus}, \text{Caesar})$

1. $\text{man}(\text{Marcus})$

2. $\text{Pompeian}(\text{Marcus})$

3. $\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x)$

4. $\text{ruler}(\text{Caesar})$

5. $\forall x: \text{Roman}(x) \rightarrow$
 $\text{loyalto}(x, \text{Caesar}) \vee$
 $\text{hate}(x, \text{Caesar})$

6. $\forall x: \exists y: \text{loyalto}(x, y)$

7. $\forall x: \forall y: \text{person}(x) \wedge \text{ruler}(y) \wedge$
 $\text{tryassassinate}(x, y) \rightarrow$
 $\neg \text{loyalto}(x, \text{Caesar})$

8. $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$

9. $\forall x: \text{man}(x) \rightarrow \text{person}(x)$

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- Specific attributes instance and isa play important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they are used to express, namely class membership and class inclusion.
- Fig. 4.2 shows the first five sentences of the last section represented in logic in three different ways.
- The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as Roman), each of which corresponds to a class.
- Asserting that $P(x)$ is true is equivalent to asserting that x is an instance (or element) of P .
- The second part of the figure contains representations that use the instance predicate explicitly.

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

1. **Man(Marcus).**
2. **Pompeian(Marcus).**
3. **$\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x).$**
4. **ruler(Caesar).**
5. **$\forall x: \text{Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar}).$**

1. **instance(Marcus, man).**
2. **instance(Marcus, Pompeian).**
3. **$\forall x: \text{instance}(x, \text{Pompeian}) \rightarrow \text{instance}(x, \text{Roman}).$**
4. **instance(Caesar, ruler).**
5. **$\forall x: \text{instance}(x, \text{Roman}). \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar}).$**

1. **instance(Marcus, man).**
2. **instance(Marcus, Pompeian).**
3. **isa(Pompeian, Roman)**
4. **instance(Caesar, ruler).**
5. **$\forall x: \text{instance}(x, \text{Roman}). \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar}).$**
6. **$\forall x: \forall y: \forall z: \text{instance}(x, y) \wedge \text{isa}(y, z) \rightarrow \text{instance}(x, z).$**

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- The predicate **instance** is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs.
- But these representations do not use an explicit **isa** predicate.
- Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3.
- The implication rule states that if an object is an instance of the subclass Pompeian then it is an instance of the superclass Roman.
- Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship.
- The third part contains representations that use both the **instance** and **isa** predicates explicitly.
- The use of the **isa** predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

COMPUTABLE FUNCTIONS AND PREDICATES

It may be necessary to compute functions as part of a fact. In these cases a computable predicate is used. A computable predicate may include computable functions such as $+$, $-$, $*$, etc. For example, $gt(x-y,10) \rightarrow bigger(x)$ contains the computable predicate gt which performs the greater than function. Note that this computable predicate uses the computable function subtraction.

Example: Consider the following statements:

1. Marcus was a man.
2. Marcus was Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 2024.
8. Alive means not dead.
9. If someone dies, he is dead at all later times.

COMPUTABLE FUNCTIONS AND PREDICATES

1. Marcus was a man.
 $\text{man}(\text{Marcus})$
2. Marcus was Pompeian.
 $\text{Pompeian}(\text{Marcus})$
3. Marcus was born in 40 A.D.
 $\text{born}(\text{Marcus}, 40)$
4. All men are mortal.
 $\forall x: \text{man}(x) \rightarrow \text{mortal}(x)$
5. All Pompeians died when the volcano erupted in 79 A.D.
 $\text{erupted}(\text{volcano}, 79) \wedge \forall x: \text{Pompeian}(x) \rightarrow \text{died}(x, 79)$
6. No mortal lives longer than 150 years.
 $\forall x \forall t1 \forall t2: \text{mortal}(x) \wedge \text{born}(x, t1) \wedge \text{gt}(t2 - t1, 150) \rightarrow \text{dead}(x, t2)$
7. It is now 2024.
 $\text{now} = 2024$
8. Alive means not dead.
 $\forall x \forall t: [\text{alive}(x, t) \rightarrow \sim \text{dead}(x, t)] \wedge [\sim \text{dead}(x, t) \rightarrow \text{alive}(x, t)]$
9. If someone dies, he is dead at all later times.
 $\forall x \forall t1 \forall t2: \text{died}(x, t1) \wedge \text{gt}(t2, t1) \rightarrow \text{dead}(x, t2)$

COMPUTABLE FUNCTIONS AND PREDICATES

Suppose we want to answer the question “**Is Marcus alive now?**”. We can do this by either proving **alive(Marcus, now)** or **~alive(Marcus, now)**.

$\sim\text{alive}(\text{Marcus}, \text{now})$
 $\downarrow 8$
 $\sim[\sim\text{dead}(\text{Marcus}, \text{now})]$
 \downarrow negation operation
 $\text{dead}(\text{Marcus}, \text{now})$
 $\downarrow 9$
 $\text{died}(\text{Marcus}, t1) \wedge \text{gt}(\text{now}, t1)$
 $\downarrow 5$
 $\text{erupted}(\text{volcano}, 79) \wedge \text{Pompeian}(\text{Marcus}) \wedge \text{gt}(\text{now}, 79)$
 \downarrow fact, 2
 $\text{gt}(\text{now}, 79)$
 \downarrow
 $\text{gt}(1991, 79)$
 \downarrow compute gt
nil

FIRST-ORDER LOGIC (PREDICATE LOGIC)

In logic, a **Well-Formed Formula (WFF)** is a string of symbols that is syntactically correct and can be interpreted as a meaningful statement. Essentially, it's a formula that follows the rules of a specific logical language, making it a valid expression within that system.

- **Symbols:**

WFFs are built from symbols like propositional variables (e.g., P , Q), logical connectives (e.g., \wedge for AND, \vee for OR, \neg for NOT), and parentheses for grouping.

- **Formation Rules:**

Specific rules dictate how these symbols can be combined to create a WFF. These rules vary depending on the type of logic (propositional, predicate, etc.).

- **Meaningful Statement:**

A WFF represents a proposition (a statement that can be true or false) and can be assigned a truth value (true or false) based on the meaning of its components and the rules of the logic system.

RESOLUTION

In the previous sections facts were proved or queries were answered using backward chaining. In this section we will examine the use of resolution for this purpose. Resolution proves facts and answers queries by **refutation**. This involves assuming the fact/query is untrue and reaching a contradiction which indicates that the opposite must be true.

Algorithm: Converting WFFS to Clause Form

1. Remove all implies, i.e. \rightarrow by applying the following: $a \rightarrow b$ is equivalent to $\sim a \vee b$.
2. Use the following rules to reduce the scope of each negation operator to a single term:
 - $\sim(\sim a) = a$
 - $\sim(a \wedge b) = \sim a \vee \sim b$
 - $\sim(a \vee b) = \sim a \wedge \sim b$
 - $\sim \forall x: p(x) = \exists x: \sim p(x)$
 - $\sim \exists x: p(x) = \forall x: \sim p(x)$
3. Each quantifier must be linked to a unique variable. For example, consider $\forall x: p(x) \vee \forall x: q(x)$. In this both quantifiers are using the same variable and one must be changed to another variable: $\forall x: p(x) \vee \forall y: q(y)$.

RESOLUTION

3. Move all quantifiers, in order, to the left of each wff.
4. Remove existential quantifiers by using **Skolem constants or functions**. For example, $\exists x: p(x)$ becomes $p(s1)$ and $\forall x \exists y: q(x,y)$ is replaced with $\forall x: q(s2(x), x)$.
5. Drop the quantifier prefix.
6. Apply the associative property of disjunctions: $a \vee (b \vee c) = (a \vee b) \vee c$ and remove brackets giving $a \vee b \vee c$.
7. Remove all disjunctions of conjunctions from predicates, i.e. create conjunctions of disjunctions instead, by applying the following rule iteratively: $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$.
8. Create a separate clause for each conjunction.
9. Rename variables in the clauses resulting from step 8 to ensure that no two clauses refer to the same variable.

RESOLUTION

Algorithm: Resolution

1. Convert the WFFS to clause form.
 2. Add the fact (or query) P to be proved to the set of clauses:
 - a) Negate P .
 - b) Convert negated P to clause form.
 - c) Add the result of (b) to the set of clauses.
 3. Repeat
 - a) Select two clauses.
 - b) Resolve the clauses using unification.
 - c) If the resolvent clause is the empty clause, then a contradiction has been reached. If not add the resolvent to the set of clauses.
- Until (a contradiction is found OR no progress can be made)

RESOLUTION - UNIFICATION

- Unification in AI is the process of making two logical expressions identical by determining a suitable substitution of variables.
- In AI, unification plays an essential role in theorem proving, where it helps match hypotheses with conclusions in logical deductions.
- In logic programming languages like Prolog, unification enables the system to match rules and facts to queries, allowing efficient pattern matching and rule evaluation.

Consider two logical expressions in predicate logic:

1. $\text{Parent}(X, \text{Mary})$
2. $\text{Parent}(\text{John}, \text{Mary})$

To unify these expressions, we find a substitution that makes them identical. Here, substituting $X = \text{John}$ results in:

$$\text{Parent}(\text{John}, \text{Mary}) = \text{Parent}(\text{John}, \text{Mary})$$

Since the expressions are now identical, unification is successful. This process allows AI systems to **infer new knowledge and establish logical relationships**, making it fundamental in knowledge-based reasoning and automated decision-making.

RESOLUTION

Consider the following wffs:

1. $\text{man}(\text{Marcus})$
2. $\text{Pompeian}(\text{Marcus})$
3. $\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x)$
4. $\text{ruler}(\text{Caesar})$
5. $\forall x: \text{Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$
6. $\forall x \exists y: \text{loyalto}(x, y)$
7. $\forall x \forall y: \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassassinate}(x, y) \rightarrow \sim \text{loyalto}(x, y)$
8. $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$
9. $\forall x: \text{man}(x) \rightarrow \text{person}(x)$

Converting these to clause form gives:

1. $\text{man}(\text{Marcus})$
2. $\text{Pompeian}(\text{Marcus})$
3. $\sim \text{Pompeian}(x) \vee \text{Roman}(x)$
4. $\text{ruler}(\text{Caesar})$
5. $\sim \text{Roman}(x_1) \vee \text{loyalto}(x_1, \text{Caesar}) \vee \text{hate}(x_1, \text{Caesar})$
6. $\text{loyalto}(x_2, s_1(x_2))$
7. $\sim \text{person}(x_3) \vee \sim \text{ruler}(y) \vee \sim \text{tryassassinate}(x_3, y) \vee \sim \text{loyalto}(x_3, y)$
8. $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$
9. $\sim \text{man}(x_4) \vee \text{person}(x_4)$

RESOLUTION

Suppose that we want to prove that **Marcus hates Caesar**. We firstly convert this to a wff: **hate(Marcus,Caesar)**. The wff is then negated and converted to clause form: **~hate(Marcus,Caesar)**.

$$\begin{array}{c} \sim \text{hate}(\text{Marcus}, \text{Caesar}) \\ \downarrow 5 \\ \sim \text{Roman}(\text{Marcus}) \vee \text{loyalto}(\text{x1}, \text{Caesar}) \\ \downarrow 3 \\ \sim \text{Pompeian}(\text{Marcus}) \vee \text{loyalto}(\text{Marcus}, \text{Caesar}) \\ \downarrow 2 \\ \text{loyalto}(\text{Marcus}, \text{Caesar}) \\ \downarrow 7 \\ \sim \text{person}(\text{Marcus}) \vee \sim \text{ruler}(\text{Caesar}) \vee \sim \text{tryassassinate}(\text{Marcus}, \text{Caesar}) \\ \downarrow 4 \\ \sim \text{person}(\text{Marcus}) \vee \sim \text{tryassassinate}(\text{Marcus}, \text{Caesar}) \\ \downarrow 8 \\ \sim \text{person}(\text{Marcus}) \\ \downarrow 9 \\ \sim \text{man}(\text{Marcus}) \\ \downarrow 1 \\ \square \end{array}$$

RESOLUTION

Consider the wffs we created above:

1. $\text{man}(\text{Marcus})$
2. $\text{Pompeian}(\text{Marcus})$
3. $\text{born}(\text{Marcus}, 40)$
4. $\forall x: \text{man}(x) \rightarrow \text{mortal}(x)$
5. $\text{erupted}(\text{volcano}, 79) \wedge \forall x: \text{Pompeian}(x) \rightarrow \text{died}(x, 79)$
6. $\forall x \forall t_1 \forall t_2: \text{mortal}(x) \wedge \text{born}(x, t_1) \wedge \text{gt}(t_2 - t_1, 150) \rightarrow \text{dead}(x, t_2)$
7. $\text{now} = 1991$
8. $\forall x \forall t: [\text{alive}(x, t) \rightarrow \sim \text{dead}(x, t)] \wedge [\sim \text{dead}(x, t) \rightarrow \text{alive}(x, t)]$
9. $\forall x \forall t_1 \forall t_2: \text{died}(x, t_1) \wedge \text{gt}(t_2, t_1) \rightarrow \text{dead}(x, t_2)$

Suppose we now want to use resolution to prove that “**Marcus is not alive now**”. We firstly have to convert these statements to clause form:

- | | |
|---|---|
| 1. $\text{man}(\text{Marcus})$ | 8. $\text{now} = 1991$ |
| 2. $\text{Pompeian}(\text{Marcus})$ | 9. $\sim \text{alive}(x_3, t) \vee \sim \text{dead}(x_3, t)$ |
| 3. $\text{born}(\text{Marcus}, 40)$ | 10. $\text{dead}(x_4, t_3) \vee \text{alive}(x_4, t_3)$ |
| 4. $\sim \text{man}(x) \vee \text{mortal}(x)$ | 11. $\sim \text{died}(x_5, t_4) \vee \sim \text{gt}(t_5, t_4) \vee \text{dead}(x_5, t_5)$ |
| 5. $\text{erupted}(\text{volcano}, 79)$ | |
| 6. $\sim \text{Pompeian}(x_1) \vee \text{died}(x_1, 79)$ | |
| 7. $\sim \text{mortal}(x_2) \vee \sim \text{born}(x_2, t_1) \vee \sim \text{gt}(t_2 - t_1, 150) \vee \text{dead}(x_2, t_2)$ | |

RESOLUTION

We want to prove $\sim\text{alive}(\text{Marcus}, \text{now})$. We negate this and convert it clause form: $\text{alive}(\text{Marcus}, \text{now})$ and find a contradiction:

$$\begin{array}{c} \text{alive}(\text{Marcus}, \text{now}) \\ \downarrow 10 \\ \text{dead}(\text{Marcus}, \text{now}) \\ \downarrow 11 \\ \sim\text{died}(\text{Marcus}, t4) \vee \sim\text{gt}(\text{now}, t4) \\ \downarrow 6 \\ \sim\text{Pompeian}(\text{Marcus}) \vee \sim\text{gt}(\text{now}, 79) \\ \downarrow 2 \\ \sim\text{gt}(\text{now}, t4) \\ \downarrow 8 \\ \sim\text{gt}(1991, 79) \\ \downarrow \\ \square \end{array}$$

RESOLUTION

1. Consider the following facts:

1. John likes all kinds of food.
2. Apples are food.
3. Chicken is food
4. Anything anyone eats and is not killed by is food.
5. Bill eats peanuts and is still alive.
6. Sue eats everything Bill eats.

a) Convert the wffs for these facts to clause form.

b) Using resolution prove that “**John likes peanuts**”.

2. Consider the following facts:

1. Steve only likes easy courses.
2. Science courses are hard.
3. All the courses in the basketweaving department are easy.
4. BK301 is a basketweaving course.

a) Convert the wffs for these facts to clause form.

b) Use resolution to answer the question “**What course would Steve like?**”.

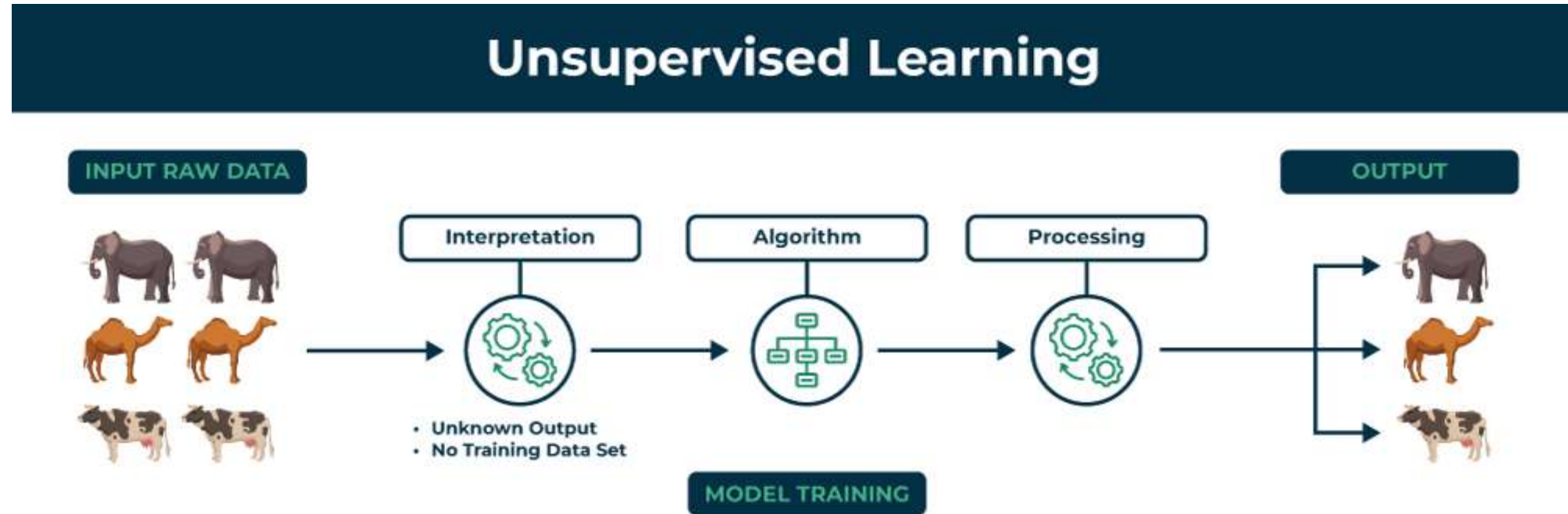
Artificial Intelligence and Machine Learning

MODULE III

UNSUPERVISED LEARNING

UNSUPERVISED LEARNING

- Unsupervised learning is a branch of machine learning that deals with unlabeled data.
- Unsupervised machine learning algorithms find hidden patterns and data without any human intervention, i.e., we don't give output to our model.
- The training model has only input parameter values and discovers the groups or patterns on its own.
- Unlike supervised learning, where the data is labeled with a specific category or outcome, unsupervised learning algorithms are tasked with finding patterns and relationships within the data without any prior knowledge of the data's meaning.



UNSUPERVISED LEARNING ALGORITHMS

There are mainly 3 types of Algorithms which are used for Unsupervised dataset.

1. Clustering
2. Association Rule Learning
3. Dimensionality Reduction

- **Association Rule Mining**

This type of unsupervised machine learning takes a rule-based approach to discovering interesting relationships between features in a given dataset. It works by using a measure of interest to identify strong rules found within a dataset.



UNSUPERVISED LEARNING ALGORITHMS

- **Dimensionality Reduction**

These algorithms seek to transform data from high-dimensional spaces to low-dimensional spaces without compromising meaningful properties in the original data. These techniques are typically deployed during exploratory data analysis (EDA) or data processing to prepare the data for modeling. Popular algorithms used for dimensionality reduction include principal component analysis (PCA) and Singular Value Decomposition (SVD) .

- **Clustering**

Clustering is a data mining technique which groups unlabeled data based on their similarities or differences. Clustering algorithms are used to process raw, unsimplified data objects into groups represented by structures or patterns in the



CLUSTERING

Cluster analysis is also known as clustering, which groups similar data points forming clusters. The goal is to ensure that data points within a cluster are more similar to each other than to those in other clusters. For example, in e-commerce retailers use clustering to group customers based on their purchasing habits. If one group frequently buys fitness gear while another prefers electronics. This helps companies to give personalized recommendations and improve customer experience. It is useful for:

- 1.Scalability:** It can efficiently handle large volumes of data.
- 2.High Dimensionality:** Can handle high-dimensional data.
- 3.Adaptability to Different Data Types:** It can work with numerical data like age, salary and categorical data like gender, occupation.
- 4.Handling Noisy and Missing Data:** Usually, datasets contain missing values or inconsistencies and clustering can manage them easily.
- 5.Interpretability:** Output of clustering is easy to understand and apply in real-world scenarios.



CLUSTERING - DISTANCE METRICS

Distance metrics are simple mathematical formulas to figure out how similar or different two data points are. Type of distance metrics we choose plays a big role in deciding clustering results. Some of the common metrics are:

- **Euclidean Distance:** It is the most widely used distance metric and finds the straight-line distance between two points.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Manhattan Distance:** It measures the distance between two points based on grid-like path. It adds the absolute differences between the values.

$$d = |x_1 - x_2| + |y_1 - y_2|$$

- **Cosine Similarity:** This method checks the angle between two points instead of looking at the distance. It's used in text data to see how similar two documents are.

$$S_c(x, y) = \frac{x \cdot y}{\|x\| \times \|y\|}$$

- **Jaccard Index:** A statistical tool used for comparing the similarity of sample sets. It's mostly used for yes/no type data or categories.

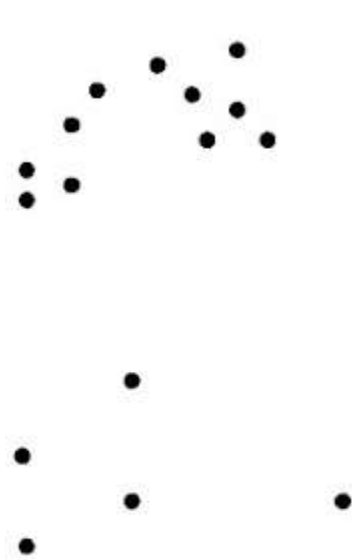
$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$



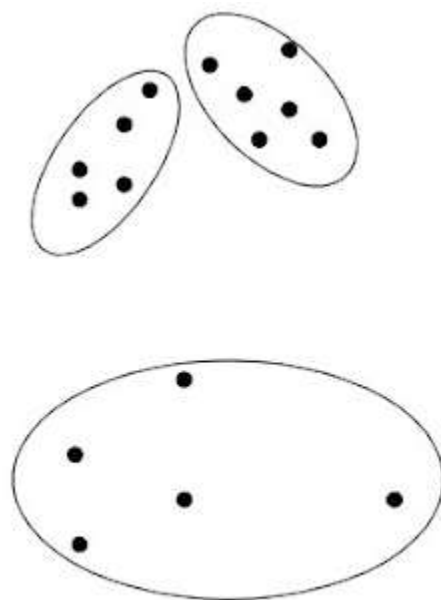
CLUSTERING - TYPES OF CLUSTERING

1. Partitioning Methods

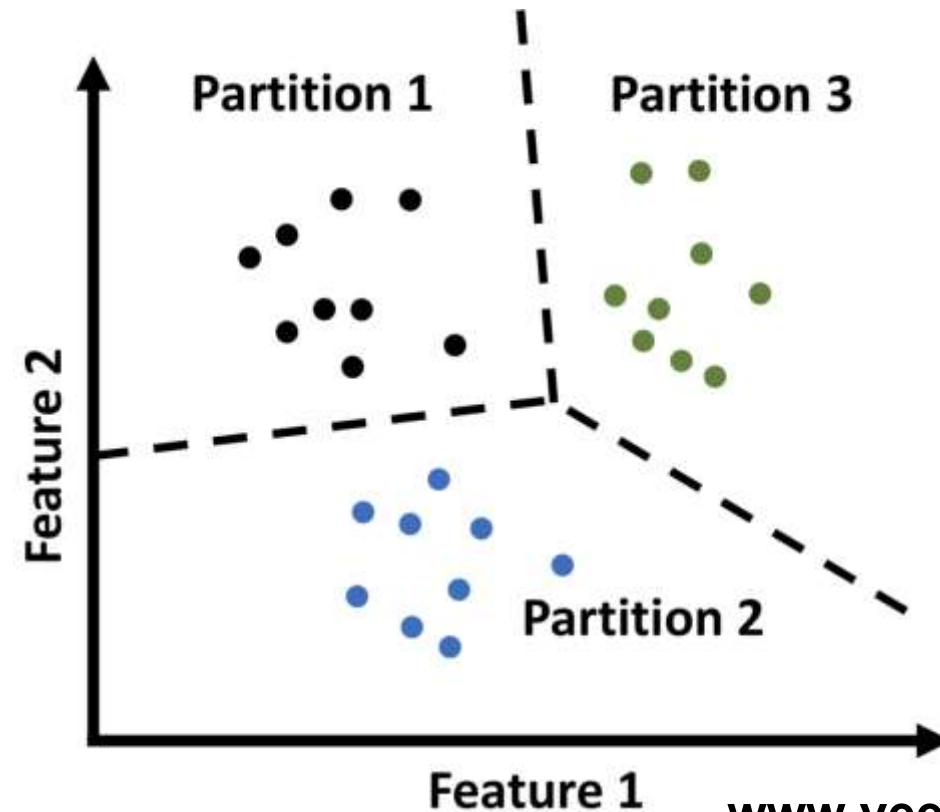
Partitioning Methods divide the data into k groups (clusters) where each data point belongs to only one group. These methods are used when you already know how many clusters you want to create. A common example is K-means clustering.



Original Points



A Partitional Clustering

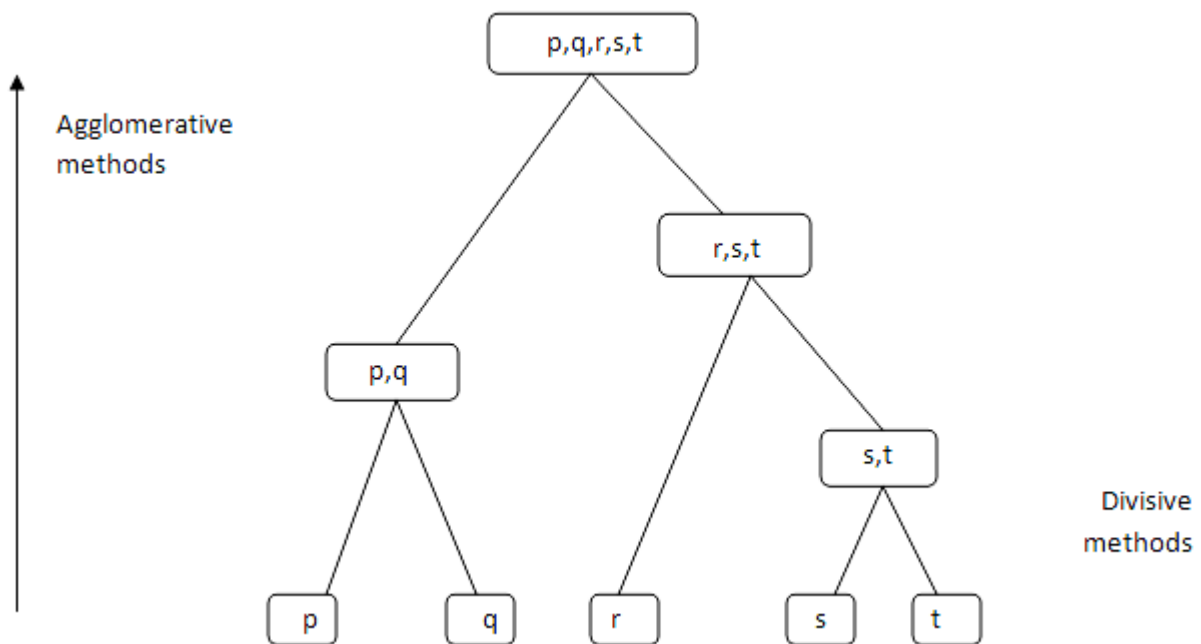


CLUSTERING - TYPES OF CLUSTERING

2. Hierarchical Methods

Hierarchical clustering builds a tree-like structure of clusters known as a **dendrogram** that represents the merging or splitting of clusters. It can be divided into:

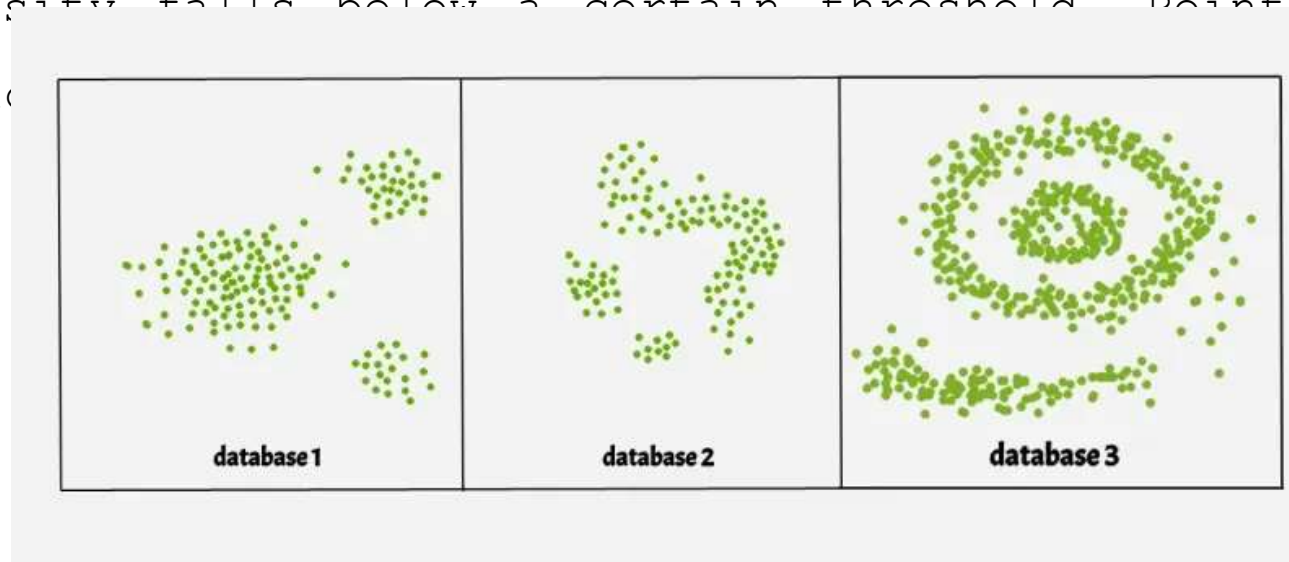
- **Agglomerative Approach (Bottom-up):** Agglomerative Approach starts with individual points and merges similar ones. Like a family tree where relatives are grouped step by step.
- **Divisive Approach (Top-down):** It starts with one big cluster and splits it repeatedly into smaller clusters. It is used to define broad categories like mammals, reptiles, etc.



CLUSTERING - TYPES OF CLUSTERING

3. Density-Based Methods

Density-based clustering group data points that are densely packed together and treat regions with fewer data points as noise or outliers. **This method is particularly useful when clusters are irregular in shape.** The algorithm identifies core points with a minimum number of neighboring points within a specified distance (known as the **epsilon radius**). It expands clusters by connecting these core points to their neighboring points until the density falls below a certain threshold. Points that do not include any cluster are considered noise.

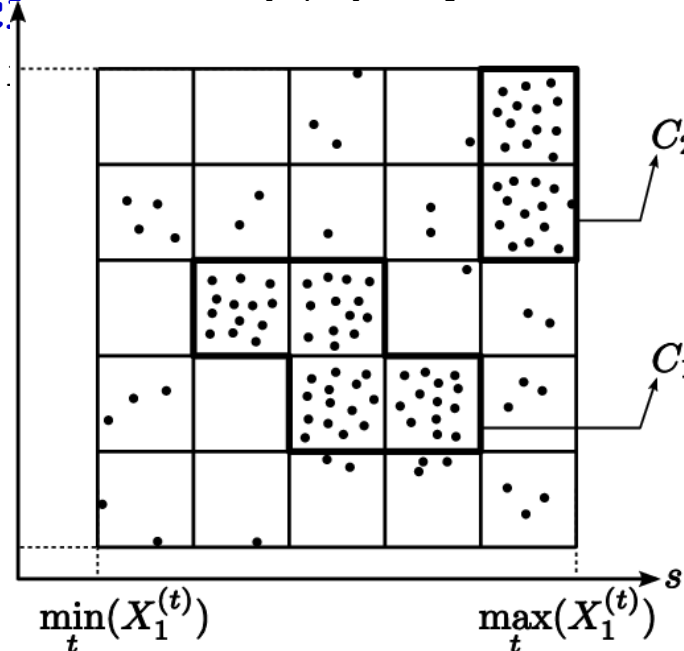


CLUSTERING - TYPES OF CLUSTERING

4. Grid-Based Methods

Grid-Based Methods divide data space into grids making clustering efficient. It quantizes the object areas into a finite number of cells that form a grid structure on which all of the operations for clustering are implemented. The benefit of the method is its quick processing time, which is generally independent of the number of data objects, still dependent on only the multiple cells in each dimension in the quantized space.

An instance of the grid-based approach involves **STING**, which explores statistical data stored in the grid cells, **WaveCl**, which explores statistical data objects using a wavelet transform approach, and **CLIQUE**, which defines clusters in high-dimensional data space.



CLUSTERING – TYPES OF CLUSTERING

5. Model-Based Methods

Model-based clustering is a **statistical approach** to data clustering. The observed (multivariate) data is considered to have been created from a finite combination of component models. Each component model is a probability distribution, generally a parametric multivariate distribution. Model-based clustering groups data by assuming it comes from a **mix of distributions**. **Gaussian Mixture Models (GMM)** are commonly used and assume the data is formed by several overlapping normal distributions. GMM is commonly used in **voice recognition systems** as it helps to distinguish different speakers by modeling each speaker's voice as a **Gaussian distribution**.

6. Constraint-Based Methods

It uses **User-defined constraints** to guide the clustering process. These constraints may specify certain relationships between **data points** such as which points should or should not be in the **same cluster**.



K-MEANS CLUSTERING

The k-means clustering algorithm operates by categorizing data points into clusters by using a mathematical distance measure, usually **Euclidean/Manhattan**, from the cluster center. The objective is to minimize the sum of distances between data points and their assigned clusters. The algorithm works as follows:

- K: The number of clusters in which the dataset has to be divided
- D: A dataset containing **N** number of objects

Output: A dataset of K clusters

1. First, we randomly initialize **k** points called means or cluster centroids.
 2. We categorize each item to its closest mean and we update the mean's coordinates, which are the averages of the items categorized in that cluster so far.
 3. We repeat the process for a given number of iterations and at the end, we have our clusters.
- Quality clusters contain at least two properties:
 1. All data points within a cluster should be similar.
 2. Clusters should be distinct from each other.



K-MEANS CLUSTERING

Cluster the following 2D points into **2 clusters (K=2)**

Point	X	Y
P1	2	10
P2	2	5
P3	8	4
P4	5	8

Step 1: Initialize Centroids

Choose any two points as initial centroids:

- $C1 = P1 (2, 10)$
- $C2 = P3 (8, 4)$



K-MEANS CLUSTERING

Step 2: Compute Distance (Iteration 1)

Use **Euclidean Distance**: $d =$

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Point	Dist. to C1 (2,10)	Dist. to C2 (8,4)	Cluster Assigned
P1 (2,10)	0.0	8.49	C1
P2 (2,5)	5.0	7.21	C1
P3 (8,4)	8.49	0.0	C2
P4 (5,8)	3.61	5.0	C1

Step 3: Update Centroids

- New C1 = Mean of (P1, P2, P4) = $((2 + 2 + 5)/3, (10 + 5 + 8)/3) = (3, 7.67)$
- New C2 = Mean of (P3) = (8,4)



K-MEANS CLUSTERING

Step 4: Recompute Distance (Iteration 2)

Point	Dist to C1 (3,7.67)	Dist to C2 (8,4)	Cluster Assigned
P1 (2,10)	2.40	8.49	C1
P2 (2,5)	2.77	7.21	C1
P3 (8,4)	6.40	0.0	C2
P4 (5,8)	2.10	5.0	C1

Step 5: Convergence

- Cluster assignments didn't change → Algorithm converged.

Final Clusters

- Cluster 1 (C1): P1, P2, P4
- Cluster 2 (C2): P3



TECHNIQUES TO FIND OPTIMAL VALUE OF K

- 1. Elbow Method:** Use this method when we have a general sense of data and want a quick visual way to estimate K.
- 2. Silhouette Method:** Use this method when we want to evaluate how well-clustered our points are and how well separated clusters are.
- 3. Gap Statistic:** Use this method when we want to compare our clustering to a random distribution to see if it is meaningful or not.
- 4. Davies-Bouldin Index:** Use this method when we want to minimize the similarity between clusters and find the optimal value of K that maximizes the separation.



ELBOW METHOD

- In K-Means clustering, we start by randomly initializing k clusters and iteratively adjusting these clusters until they stabilize at an equilibrium point. However, before we can do this, we need to decide how many clusters (k) we should use.

The Elbow Method helps us find this optimal k value. Here's how it works:

1. We iterate over a range of k values, typically from 1 to n (where n is a hyper-parameter you choose).

2. For each k, we calculate the Within-Cluster Sum of Squares (WCSS). WCSS measures how well the data points are clustered around their respective centroids.

It is defined as the sum of the squared distances between each point and its cluster centroid:

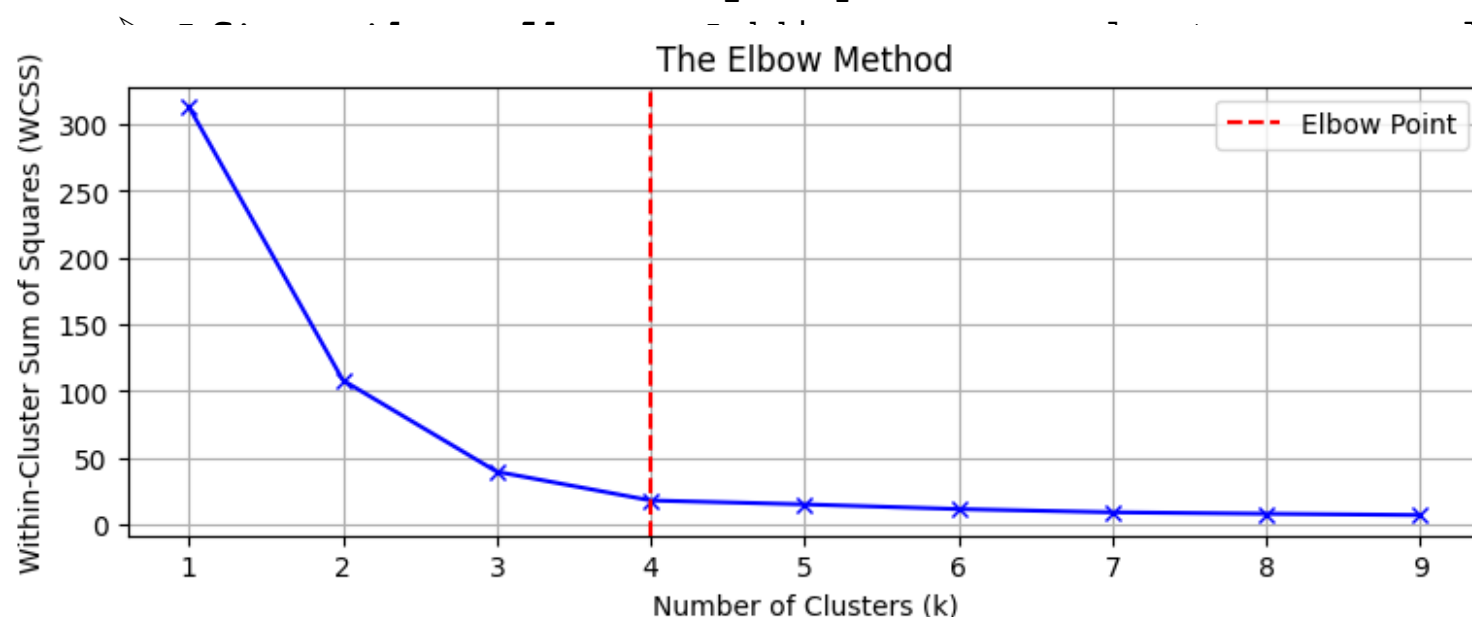
$$WCSS = \sum_{i=1}^k \sum_{j=1}^{n_i} \text{distance} \left(x_j^{(i)}, c_i \right)^2$$

where, $\text{distance} \left(x_j^{(i)}, c_i \right)$ represents the distance between j^{th} the $x_j^{(i)}$ in the cluster i and the centroid c_i of that cluster.



ELBOW METHOD

- Calculate a distance measure called WCSS (Within-Cluster Sum of Squares). This tells us how spread out the data points are within each cluster.
- Utilize different k values (number of clusters). For each k , we run KMeans and calculate the WCSS.
- Identifying the Elbow Point: As we increase k , the WCSS typically decreases because we're creating more clusters, which tend to capture more data variations. However, there comes a point where adding more clusters results in only a marginal decrease in WCSS. This is where we observe an "elbow" shape in the graph.
 - **Before the elbow:** Increasing k significantly reduces WCSS, indicating that new clusters effectively capture more of the data's variability.



It is in a minimal reduction in WCSS, it may be necessary and could lead to the **goal** is to identify the point where the rate of decrease in **WCSS sharply changes**, indicating that adding more clusters (beyond this point) yields diminishing returns.



SILHOUETTE METHOD

Silhouette Method measures how similar each data point is to its own cluster compared to other clusters. It calculates a score for each point which tells us if the point is well-clustered. A higher score means the point is close to its own cluster and far from others. The Silhouette Score ranges from -1 to $+1$:

- $+1$ means the point is well-placed in its cluster.
- 0 means the point is on the boundary between clusters.
- -1 means the point is in the wrong cluster.

We calculate the average Silhouette Score for different values of K . The best K is the one with the highest average score, indicating that the clusters are well-separated and each point is correctly grouped.

Where, $s(i) = \frac{b(i) - a(i)}{\max(b(i), a(i))}$ and $s(i) = 0$, if $|C_i| =$

- $s(i)$ is the silhouette score for point i .
- $a(i)$ is the average distance from point i to all other points in the same cluster (cohesion).



SILHOUETTE METHOD

$C(i)$ - The cluster assigned to the i^{th} data point

$|C(i)|$ - The number of data points in the cluster assigned to the i^{th} data point

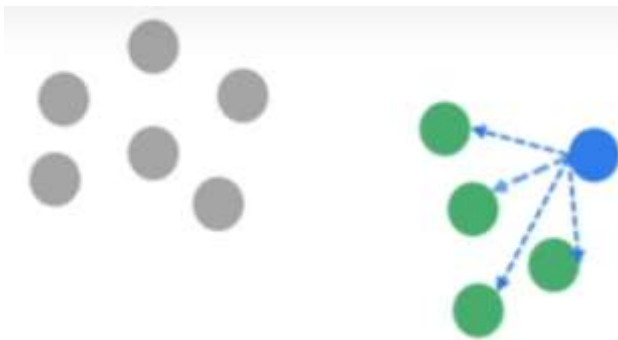
$a(i)$ is the measure of similarity of the point i to its own cluster. It is measured as the average distance of i from other points in the cluster.

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j)$$

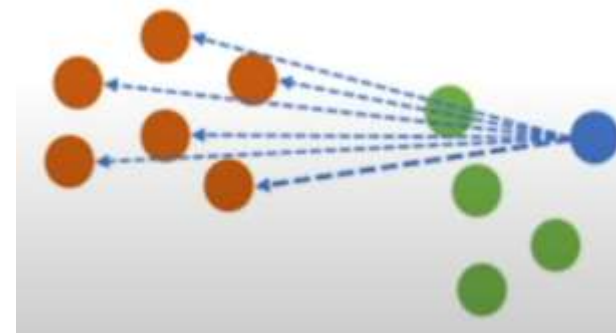
$b(i)$ is the measure of dissimilarity of i from points in other clusters.

$$b(i) = \min_{i \neq j} \frac{1}{|C_j|} \sum_{j \in C_j} d(i, j)$$

$d(i, j)$ is the distance between points i and j . Generally, **Euclidean Distance** is used as the distance metric.



$a(i)$

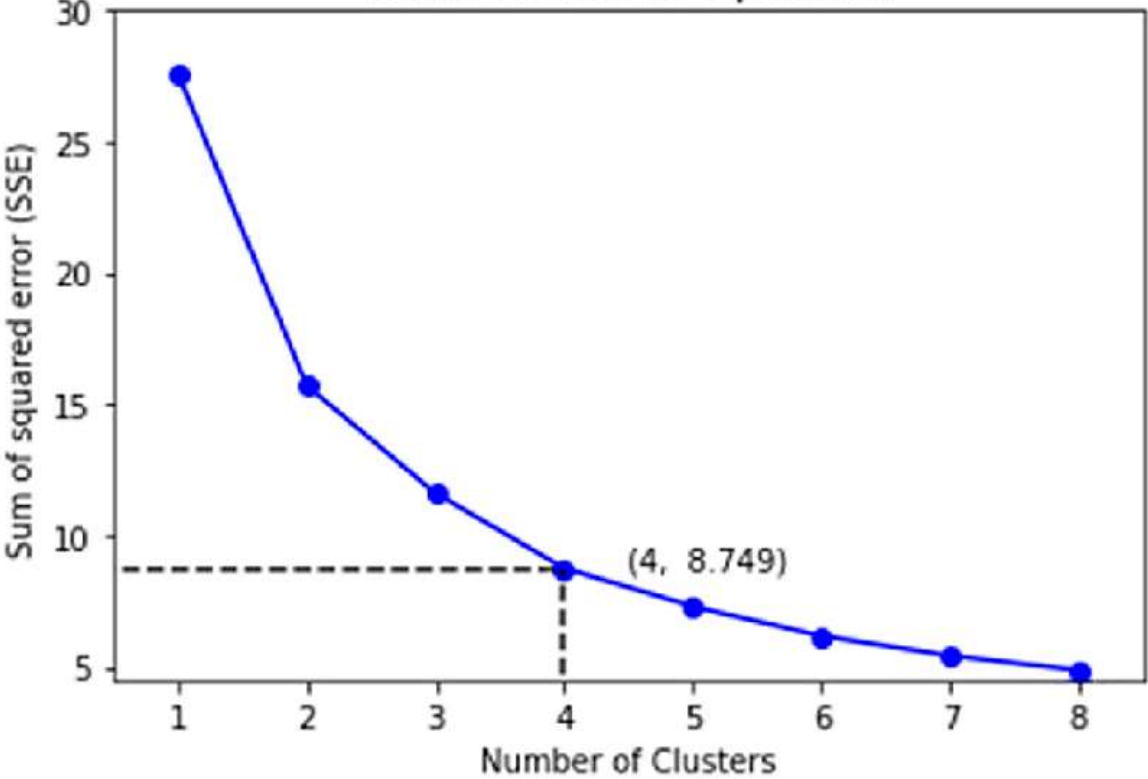


$b(i)$

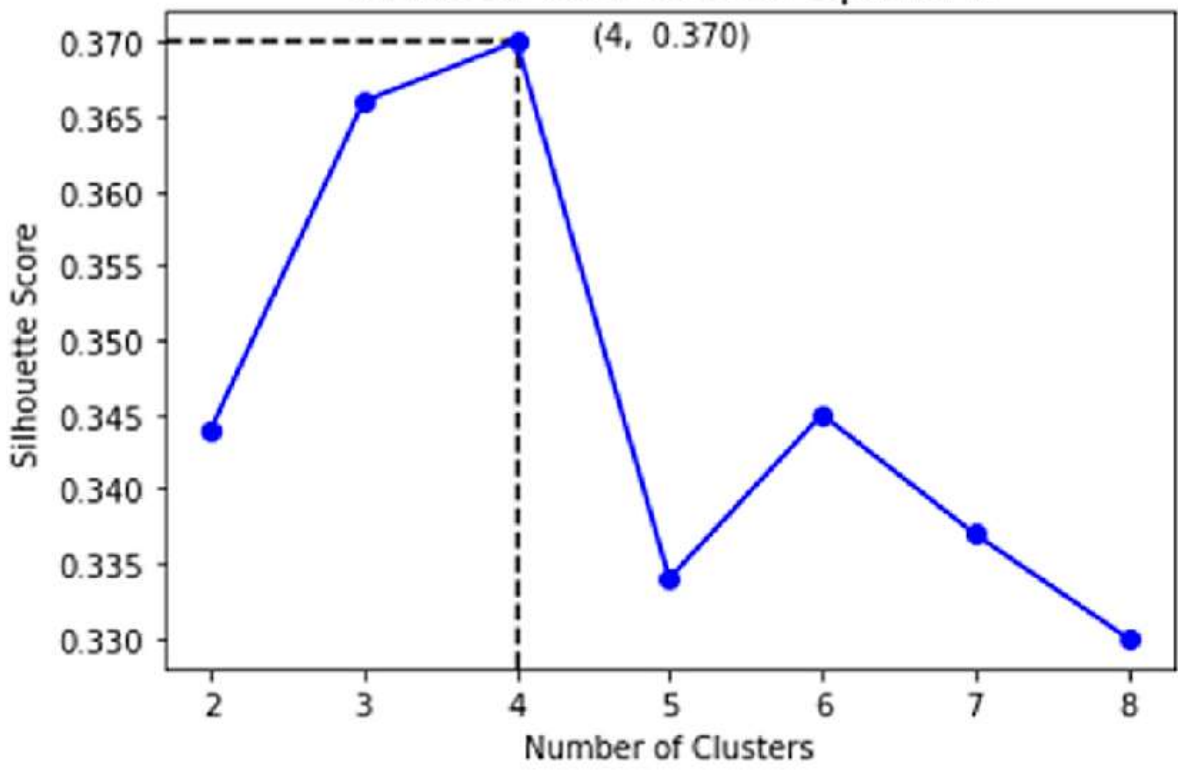


SILHOUETTE METHOD

Elbow Method for Optimal k



Silhouette Coefficient for Optimal k



K-MEDOIDS ALGORITHM

Steps of the K-Medoids Algorithm

1. Select K initial medoids (randomly chosen from data points).
2. Assign each point to the nearest medoid based on distance.
3. Swap Step: Try replacing a medoid with a non-medoid point to see if it reduces clustering cost.
4. Repeat Steps 2 & 3 until no further improvement.

Objective Function (Cost Function)

K-Medoids minimizes the sum of distances between each point and its medoid:

$$\sum_{i=1}^K \sum_{x \in C_i} d(x, m_i)$$



K-MEDOIDS ALGORITHM

Suppose we have the data given below, and we want to divide the data into two clusters, i.e., $k=2$.

S. No.	X	Y
1	9	6
2	10	4
3	4	4
4	5	8
5	3	8
6	2	5
7	8	5
8	4	6
9	8	4
10	9	3



K-MEDOIDS ALGORITHM

Pick up two random medoids(as our k=2). So, we pick M1 (8,4) and M2 (4,6) as our initial medoids.

Sl.No .	X	Y	Distance from M1 (8,4)	Distance from M2 (4,6)	Assigned Cluster
1	9	6	3	5	M1
2	10	4	2	8	M1
3	4	4	4	2	M2
4	5	8	7	3	M2
5	3	8	9	3	M2
6	2	5	7	3	M2
7	8	5	1	5	M1
8	4	6	-	-	M2
9	8	4	-	-	M1
10	9	3	2	8	M1

Cost = (3+2+1+2) + (2+3+3+3) =
(8) + (11) = 19



K-MEDOIDS ALGORITHM

Randomly select another medoid from the **Non medoid points** and swap it. Change M1 to (8,5). The new medoids are M1(8,5) and M2(4,6)

S1.No	X	Y	Distance from M1 (8,5)	Distance from M2 (4,6)	Assigned Cluster
1	9	6	2	5	M1
2	10	4	3	8	M1
3	4	4	5	2	M2
4	5	8	6	3	M2
5	3	8	9	3	M2
6	2	5	6	3	M2
7	8	5	–	5	M1
8	4	6	–	–	M2
9	8	4	1	–	M1
10	9	3	3	8	M1



K-MEDOIDS ALGORITHM

Points (1,2,7,10) are assigned to M1(8,5) and points (3,4,5,6) are assigned to M2(4,6).

• **New Total Cost**= (2+3+1+3)+(2+3+3+3) = 9+11 = 20

Calculate the Swap Cost

• **Swap Cost**= New Total Cost- Cost = 20-19 = 1

Since **Swap Cost**>0, we would undo the swap.

Our final medoids are **M1(8,4)** and **M2(4,6)**, and the **two clusters** are formed with these medoids.

Key Differences: K-Medoids vs. K-Means

Features	K-Means	K-Medoids
Centroid	Mean of Cluster Points	Actual Data Points (Medoids)
Distance Metric	Euclidean Distance	Any distance Metric
Outlier Sensitivity	Sensitive to Outliers	Robust to Outliers
Computational Cost	Faster	Slower (due to pairwise distance calculations)
Best for	Large Datasets, Spherical Clusters	Small Datasets, Non-spherical Clusters



HIERARCHICAL CLUSTERING

Hierarchical clustering is a method of cluster analysis that creates a hierarchical representation of the clusters in a dataset. The result of hierarchical clustering is a tree-like structure, called a dendrogram, which illustrates the hierarchical relationships among the clusters.

Types of Hierarchical Clustering

Basically, there are two types of hierarchical Clustering:

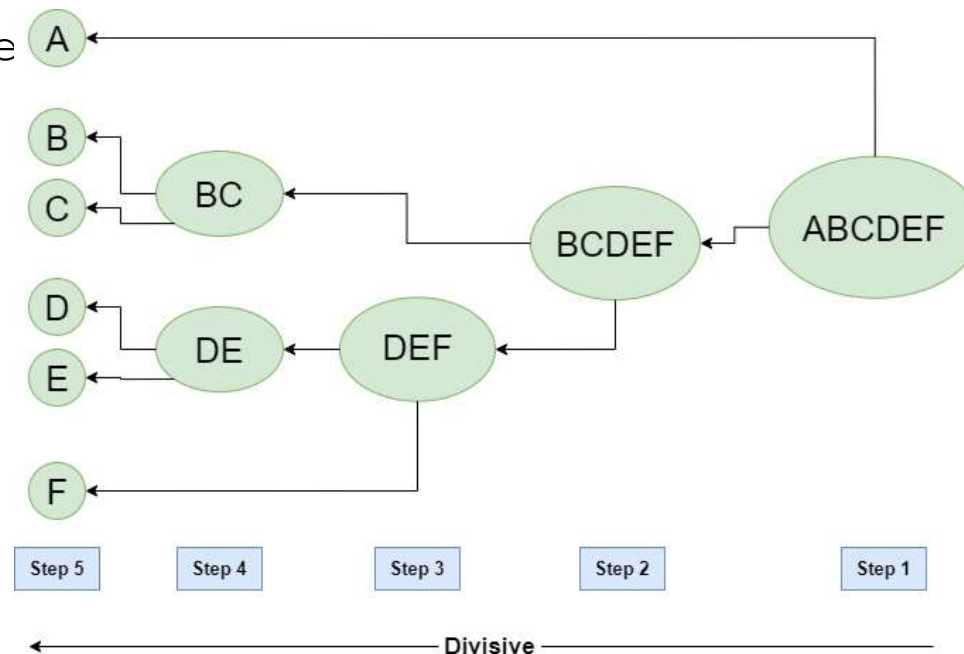
1. Divisive clustering
2. Agglomerative Clustering



DIVISIVE HIERARCHICAL CLUSTERING

A **divisive hierarchical clustering** method employs a top-down strategy.

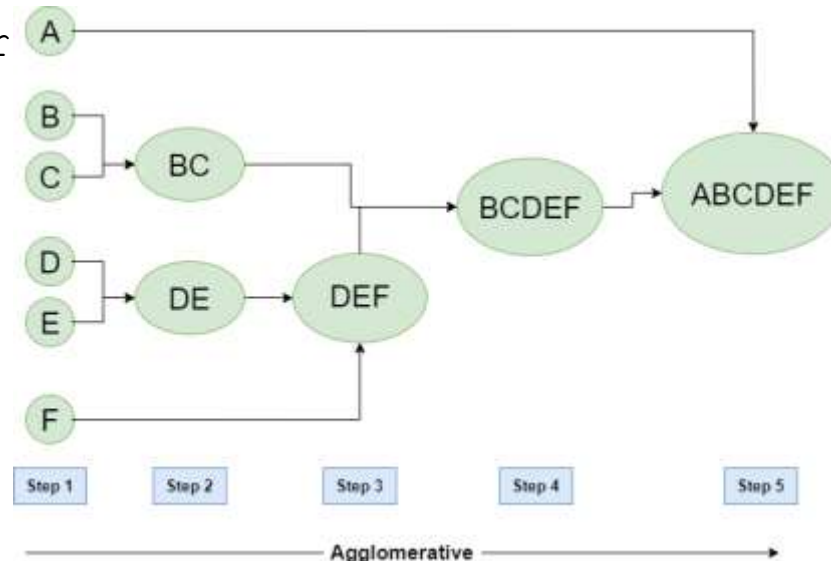
- It starts by placing all objects in one cluster, which is the hierarchy's root.
- It then divides the root cluster into several smaller subclusters, and recursively partitions those clusters into smaller ones.
- The partitioning process continues until each cluster at the lowest level is coherent enough—either containing only one object, or the objects within a cluster are sufficiently similar to each other.



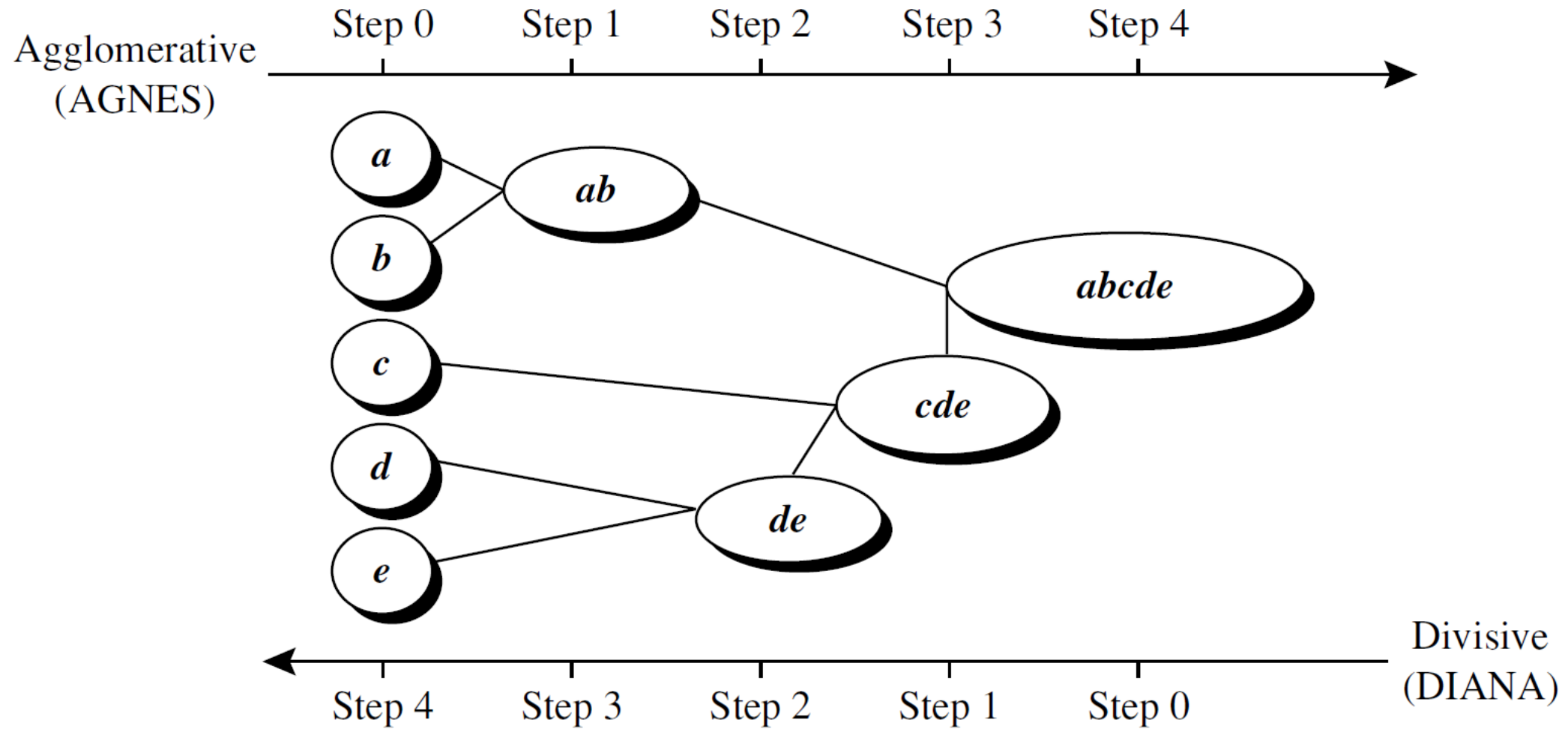
AGGLOMERATIVE HIERARCHICAL CLUSTERING

An **agglomerative hierarchical clustering** method uses a bottom-up strategy.

- It typically starts by letting each object form its own cluster and iteratively merges clusters into larger and larger clusters, until all the objects are in a single cluster or certain termination conditions are satisfied.
- The single cluster becomes the hierarchy's root.
- For the merging step, it finds the two clusters that are closest to each other (according to some similarity measure), and combines the two to form one cluster.
- Because two clusters are merged per one object, an agglomerative method



AGGLOMERATIVE HIERARCHICAL CLUSTERING



AGGLOMERATIVE HIERARCHICAL CLUSTERING

Types of Linkage Criteria

When deciding which clusters to merge, we compute **distance between clusters** in different ways:

1. Single Linkage (Min Distance)

- Distance between two clusters = **minimum distance** between any two points.
- Tends to form **long "chain-like" clusters**.

2. Complete Linkage (Max Distance)

- Distance between two clusters = **maximum distance** between any two points.
- Tends to form **compact, spherical clusters**.

3. Average Linkage

• Distance between two clusters = **average distance** between all points across



AGGLOMERATIVE HIERARCHICAL CLUSTERING

Points: $P_1=2, P_2=4, P_3=6, P_4=8$

Distance: Manhattan Distance

Rule = Single Linkage

	P1	P2	P3	P4
P1	0	2	4	6
P2	2	0	2	4
P3	4	2	0	2
P4	6	4	2	0

Smallest off-diagonal distance = 2 (ties exist).

Merge 1: $\{P_1\}$ and $\{P_2\} \rightarrow \text{Cluster } C_{12} = \{P_1, P_2\}$



AGGLOMERATIVE HIERARCHICAL CLUSTERING

	C12	P3	P4
C12	0	2	4
P3	2	0	2
P4	4	2	0

Single linkage updates:

- $d(C12, P3) = \min(d(P_1, P_3), d(P_2, P_3)) = \min(4, 2) = 2$
- $d(C12, P4) = \min(d(P_1, P_4), d(P_2, P_4)) = \min(6, 4) = 4$

Smallest distance = 2 (ties: $C_{12} - P_3$ and $P_3 - P_4$) .

Merge 2: C_{12} and $P_3 \rightarrow$ **Cluster** $C_{123} = \{P_1, P_2, P_3\}$

	C123	P4
C123	0	2
P4	2	0

Distance Matrix After Merge 2 (Clusters: C_{123} , P_4)

$$d(C_{123}, P_4) = \min(d(P_1, P_4), d(P_2, P_4), d(P_3, P_4)) = \min(6, 4, 2) = 2$$



AGGLOMERATIVE HIERARCHICAL CLUSTERING

	C123	P4
C123	0	2
P4	2	0

Smallest distance = 2

Merge 3: C_{123} and $P_4 \rightarrow$ Cluster $C_{1234} = \{P_1, P_2, P_3, P_4\}$

Merge #	Clusters Merged	Linkage Distance
1	$\{P_1, P_2\}$	2
2	$\{P_1, P_2\}, \{P_3\}$	2
3	$\{P_1, P_2, P_3\}, \{P_4\}$	2



DENSITY-BASED CLUSTERING

- Partitioning and hierarchical methods are designed to find spherical-shaped clusters. They have difficulty finding clusters of arbitrary shape such as the "S" shape and oval clusters.
 - To find clusters of arbitrary shape, alternatively, we can model clusters as dense regions in the data space, separated by sparse regions. This is the main strategy behind density-based clustering methods, which can discover clusters of nonspherical shape.
- **How can we find dense regions in density-based clustering?**
- The *density* of an object o can be measured by the number of objects close to o .
- DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) finds *core objects*, that is, objects that have dense neighborhoods. It connects core objects and their neighborhoods to form dense regions as clusters.
- **How does DBSCAN quantify the neighborhood of an object?**

A user-specified parameter $\epsilon > 0$ is used to specify the radius of a neighborhood we consider for every object. The **ϵ -neighborhood** of an object o is the space within a

DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

Input:

D : a data set containing n objects,
 ϵ : the radius parameter, and
 $MinPts$: the neighborhood density threshold.

Output: A set of density-based clusters.

Algorithm:

```
(1) mark all objects as unvisited;
(2) do
(3)     randomly select an unvisited object  $p$ ;
(4)     mark  $p$  as visited;
(5)     if the  $\epsilon$ -neighborhood of  $p$  has at least  $MinPts$  objects
(6)         create a new cluster  $C$ , and add  $p$  to  $C$ ;
(7)         let  $N$  be the set of objects in the  $\epsilon$ -neighborhood of  $p$ ;
(8)         for each point  $p'$  in  $N$ 
(9)             if  $p'$  is unvisited
(10)                 mark  $p'$  as visited;
(11)                 if the  $\epsilon$ -neighborhood of  $p'$  has at least  $MinPts$  points, add those
points to  $N$ ;
(12)                 if  $p'$  is not yet a member of any cluster, add  $p'$  to  $C$ ;
(13)         end for
(14)         output  $C$ ;
(15)     else mark  $p$  as Noise;
(16) until no object is unvisited;
```



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

Given the points A(3, 7), B(4, 6), C(5, 5), D(6, 4), E(7, 3), F(6, 2), G(7, 2) and H(8, 4), Find the core points and outliers using DBSCAN. Take Eps (ϵ) = 1.9 and MinPts = 4.

Data Points	X	Y
P1	3	7
P2	4	6
P3	5	5
P4	6	4
P5	7	3
P6	6	2
P7	7	2
P8	8	4
P9	3	3
P10	2	6
P11	3	5
P12	2	4

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1												
P2												
P3												
P4												
P5												
P6												
P7												
P8												
P9												
P10												
P11												
P12												

DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1	0											
P2		0										
P3			0									
P4				0								
P5					0							
P6						0						
P7							0					
P8								0				
P9									0			
P10										0		
P11											0	
P12												0

DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1	0											
P2	1.4 1	0										
P3			0									
P4				0								
P5					0							
P6						0						
P7							0					
P8								0				
P9									0			
P10										0		
P11											0	
P12												0



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1	0											
P2	1.4 1	0										
P3	2.8 3	1.4 1	0									
P4				0								
P5					0							
P6						0						
P7							0					
P8								0				
P9									0			
P10										0		
P11											0	
P12												0



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1	0											
P2	1.4 1	0										
P3	2.8 3	1.4 1	0									
P4	4.2 4	2.8 3	1.4 1	0								
P5					0							
P6						0						
P7							0					
P8								0				
P9									0			
P10										0		
P11											0	
P12												



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1	0											
P2	1.4 1	0										
P3	2.8 3	1.4 1	0									
P4	4.2 4	2.8 3	1.4 1	0								
P5	5.6 6	4.2 4	2.8 3	1.4 1	0							
P6						0						
P7							0					
P8								0				
P9									0			
P10										0		
P11											0	



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P1	0											
P2	1.4 1	0										
P3	2.8 3	1.4 1	0									
P4	4.2 4	2.8 3	1.4 1	0								
P5	5.6 6	4.2 4	2.8 3	1.4 1	0							
P6	5.8 3	4.4 7	3.1 6	2.0 0	1.4 1	0						
P7	6.4 0	5.0 0	3.6 1	2.2 4	1.0 0	1.0 0	0					
P8	5.8 3	4.4 7	3.1 6	2.0 0	1.4 1	2.8 3	2.2 4	0				
P9	4.0 0	3.1 6	2.8 3	3.1 6	4.0 0	3.1 6	4.1 2	5.1 0	0			

Eps (ϵ) = 1.9 and
MinPts = 4

- P1 : P2 , P10
- P2 : P1 , P3 , P11
- P3 : P2 , P4
- P4 : P3 , P5
- P5 : P4 , P6 , P7 , P8
- P6 : P5 , P7
- P7 : P5 , P6
- P8 : P5
- P9 : P12
- P10 : P1 , P11
- P11 : P2 , P10 , P12
- P12 : P9 , P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

Eps (ϵ) = 1.9 and

MinPts = 4

Data Points	Status	
P1		
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		
P10		
P11		
P12		

P1 : P2 , P10

P2 : P1 , P3 , P11

P3 : P2 , P4

P4 : P3 , P5

P5 : P4 , P6 , P7 , P8

P6 : P5 , P7

P7 : P5 , P6

P8 : P5

P9 : P12

P10 : P1 , P11

P11 : P2 , P10 , P12

P12 : P9 , P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

Eps (ϵ) = 1.9 and

MinPts = 4

Data Points	Status	
P1	Noise	
P2	Core	
P3	Noise	
P4	Noise	
P5	Core	
P6	Noise	
P7	Noise	
P8	Noise	
P9	Noise	
P10	Noise	
P11	Core	
P12	Noise	

P1 : P2 , P10

P2 : P1 , P3 , P11

P3 : P2 , P4

P4 : P3 , P5

P5 : P4 , P6 , P7 , P8

P6 : P5 , P7

P7 : P5 , P6

P8 : P5

P9 : P12

P10 : P1 , P11

P11 : P2 , P10 , P12

P12 : P9 , P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

Eps (ϵ) = 1.9 and

MinPts = 4

Data Points	Status	
P1	Noise	Border
P2	Core	
P3	Noise	Border
P4	Noise	Border
P5	Core	
P6	Noise	Border
P7	Noise	Border
P8	Noise	Border
P9	Noise	
P10	Noise	Border
P11	Core	
P12	Noise	Border

P1 : P2 , P10

P2 : P1 , P3 , P11

P3 : P2 , P4

P4 : P3 , P5

P5 : P4 , P6 , P7 , P8

P6 : P5 , P7

P7 : P5 , P6

P8 : P5

P9 : P12

P10 : P1 , P11

P11 : P2 , P10 , P12

P12 : P9 , P11

P1 : P2 , P10

P2 : P1 , P3 , P11

P3 : P2 , P4

P4 : P3 , P5

P5 : P4 , P6 , P7 , P8

P6 : P5 , P7

P7 : P5 , P6

P8 : P5

P9 : P12

P10 : P1 , P11

P11 : P2 , P10 , P12

P12 : P9 , P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

P1: P2, P10

P2: P1, P3, P11

P3: P2, P4

P4: P3, P5

P5: P4, P6, P7, P8

P6: P5, P7

P7: P5, P6

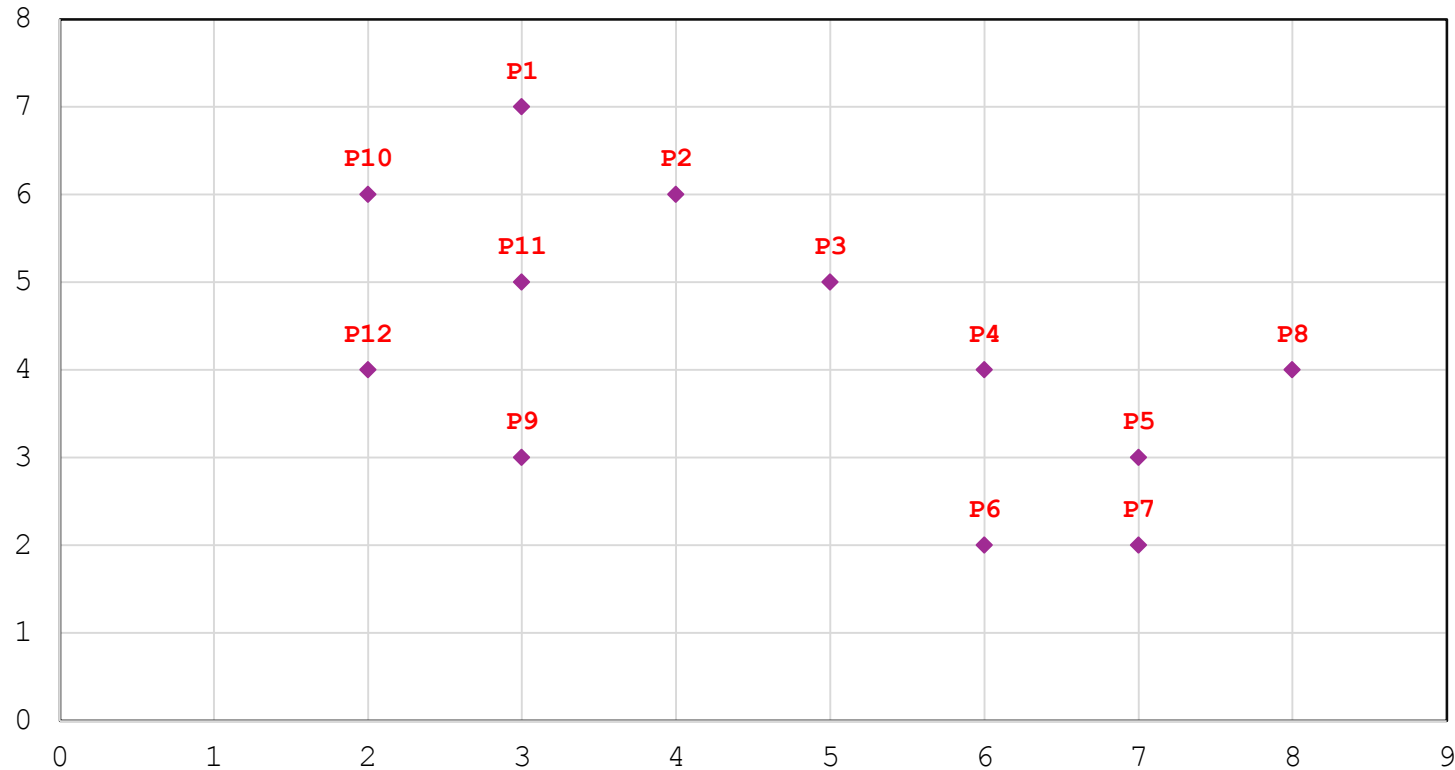
P8: P5

P9: P12

P10: P1, P11

P11: P2, P10, P12

P12: P9, P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

P1 : P2 , P10

P2 : P1 , P3 , P11

P3 : P2 , P4

P4 : P3 , P5

P5 : P4 , P6 , P7 , P8

P6 : P5 , P7

P7 : P5 , P6

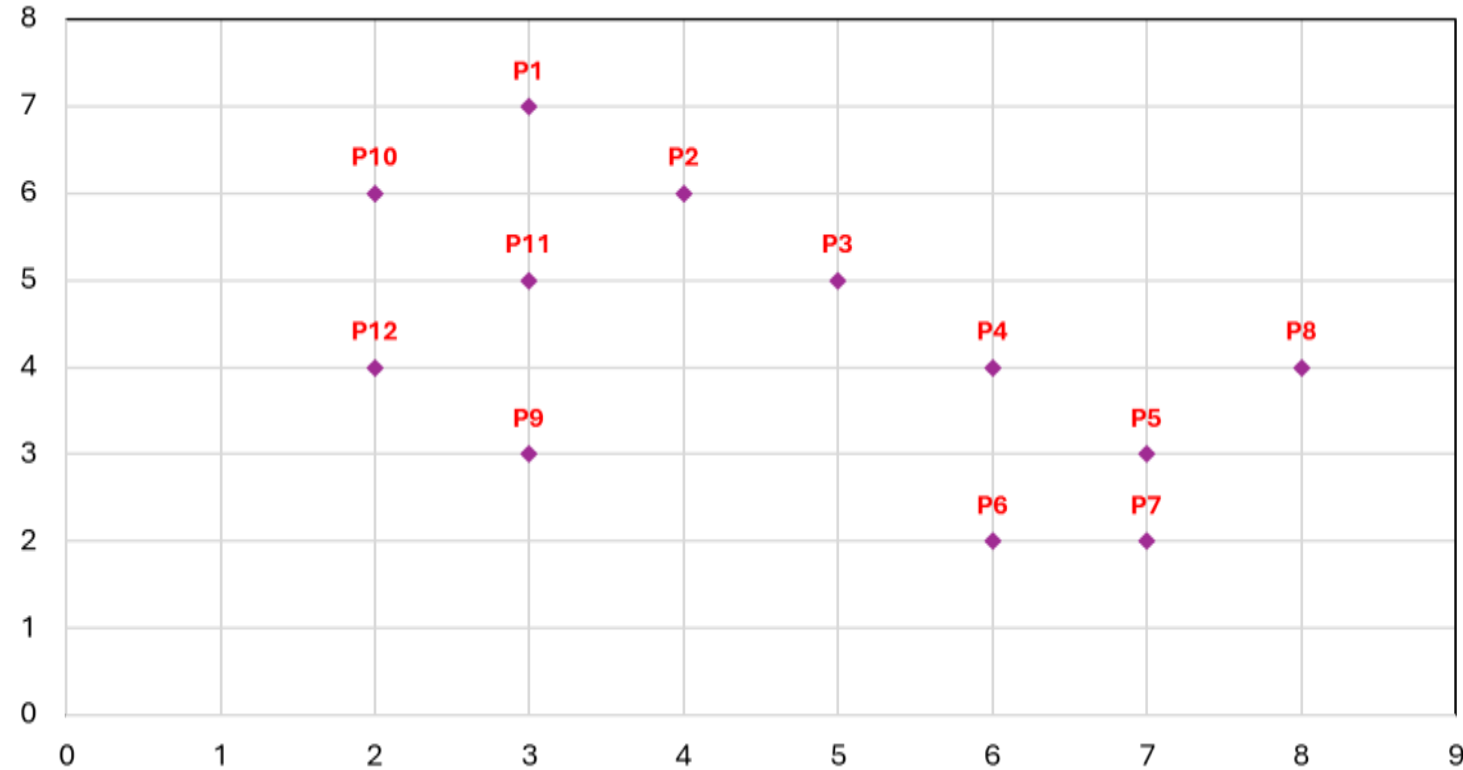
P8 : P5

P9 : P12

P10 : P1 , P11

P11 : P2 , P10 , P12

P12 : P9 , P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

P1: P2, P10

P2: P1, P3, P11

P3: P2, P4

P4: P3, P5

P5: P4, P6, P7, P8

P6: P5, P7

P7: P5, P6

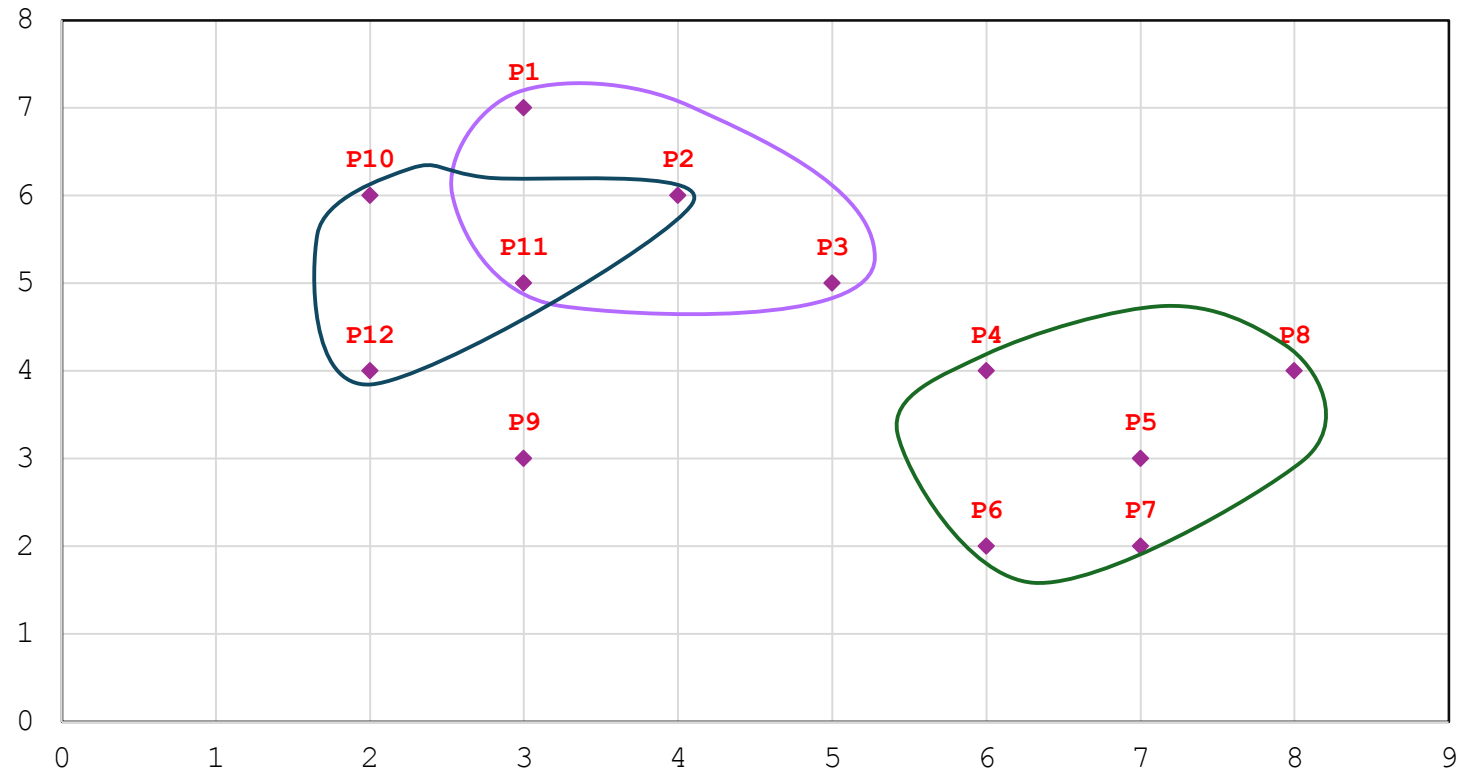
P8: P5

P9: P12

P10: P1, P11

P11: P2, P10, P12

P12: P9, P11



DBSCAN: A DENSITY-BASED CLUSTERING ALGORITHM

P1: P2, P10

P2: P1, P3, P11

P3: P2, P4

P4: P3, P5

P5: P4, P6, P7, P8

P6: P5, P7

P7: P5, P6

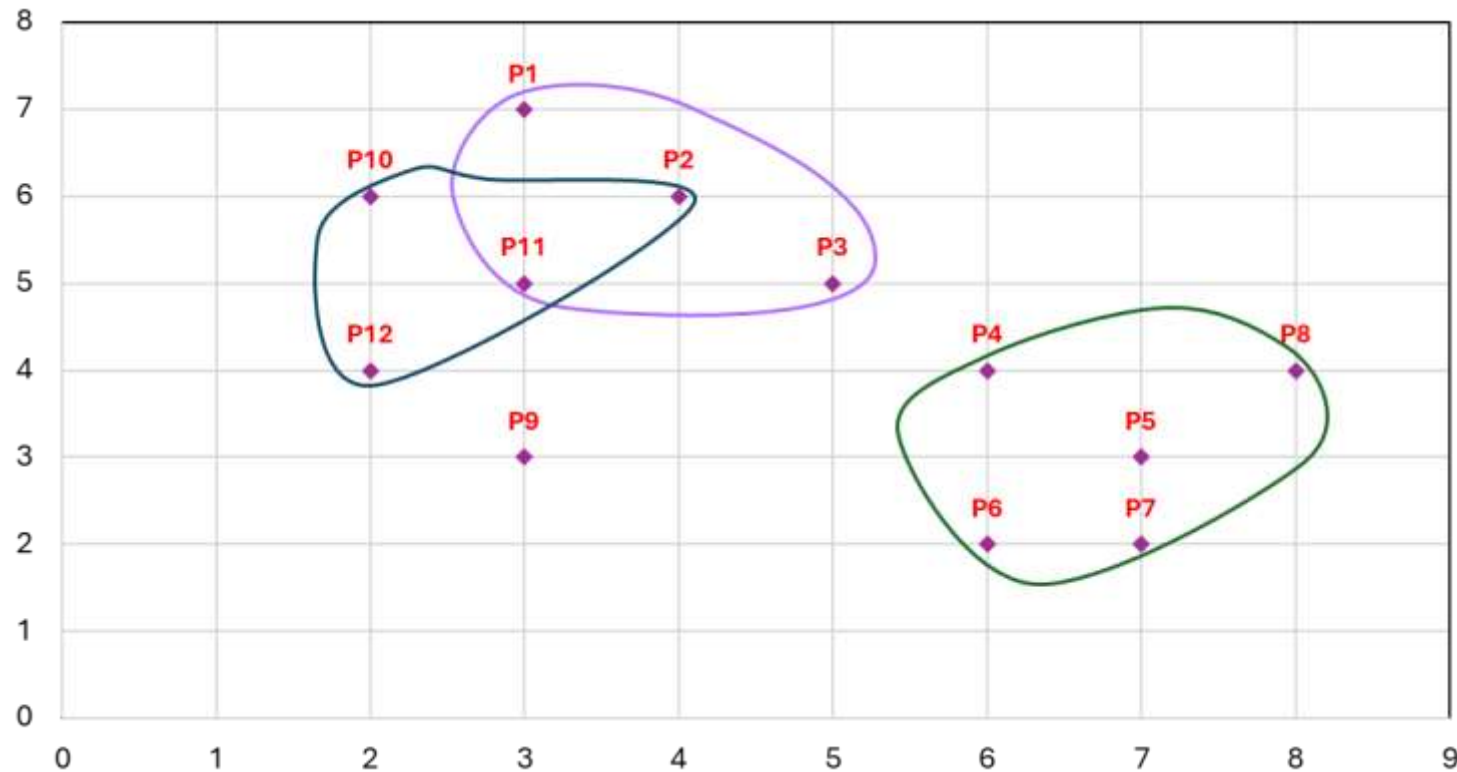
P8: P5

P9: P12

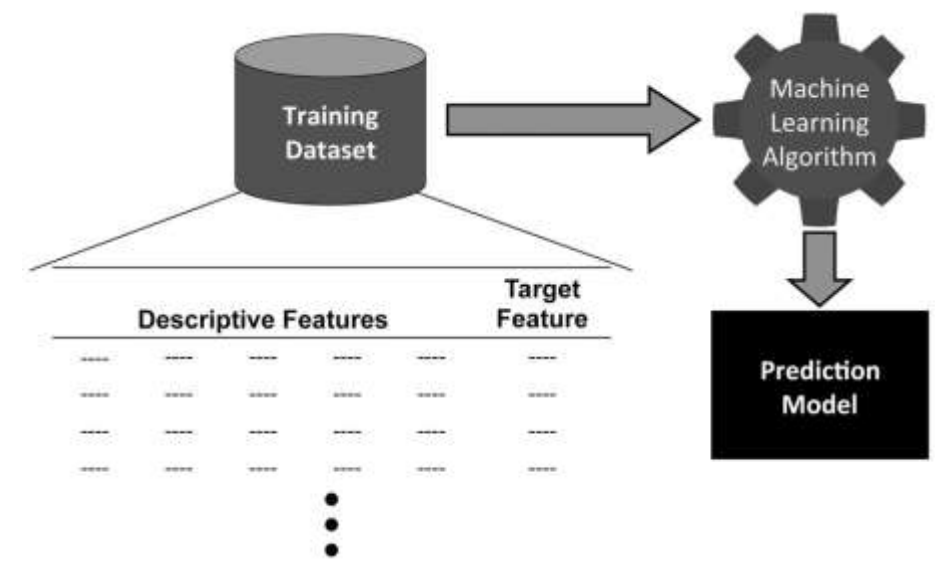
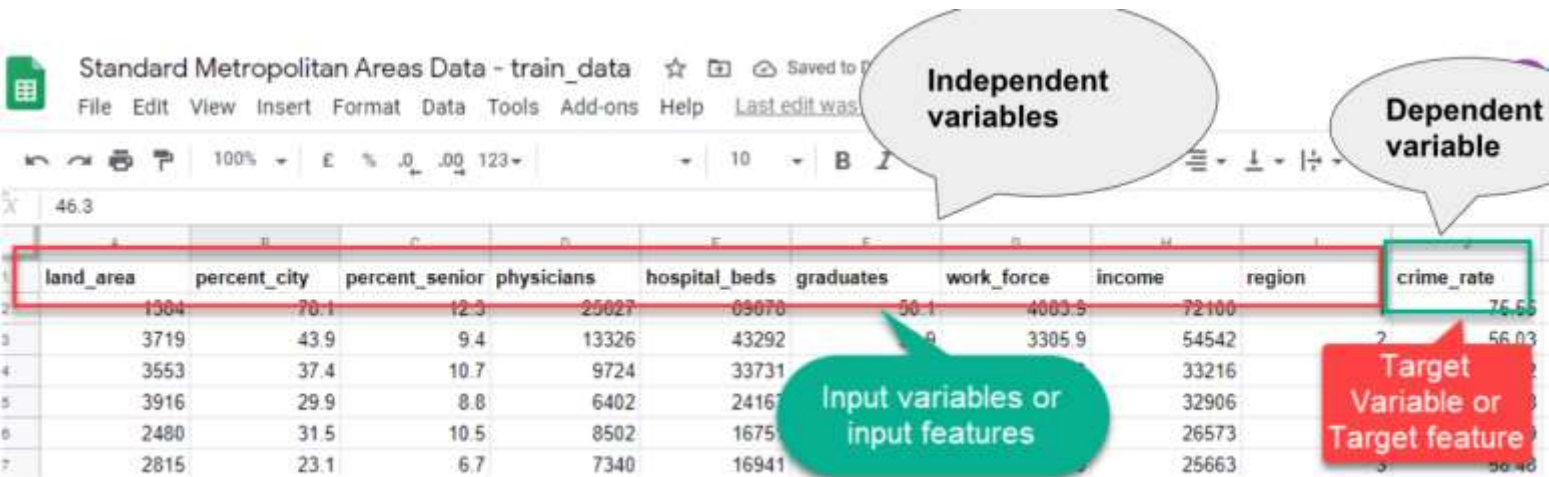
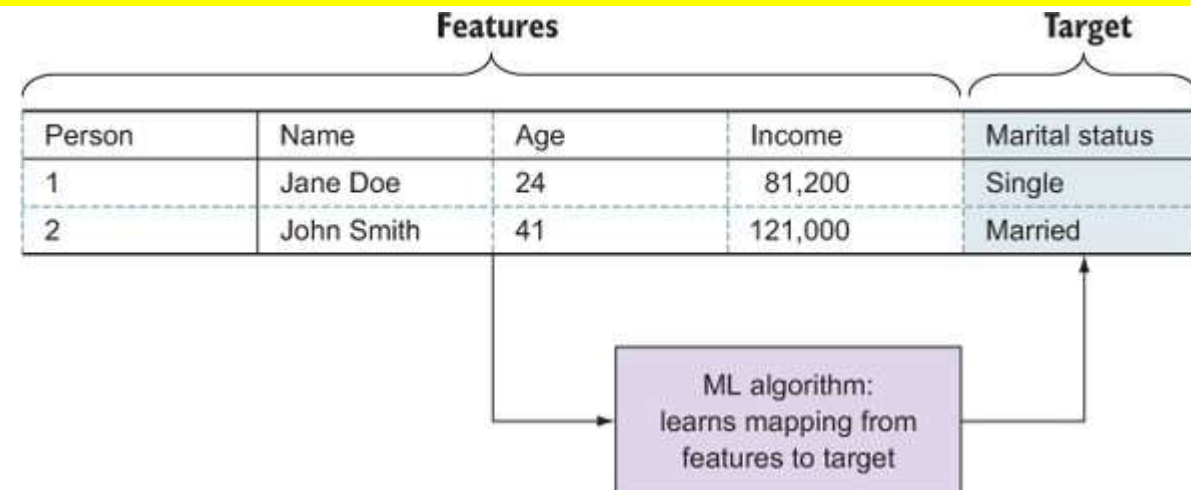
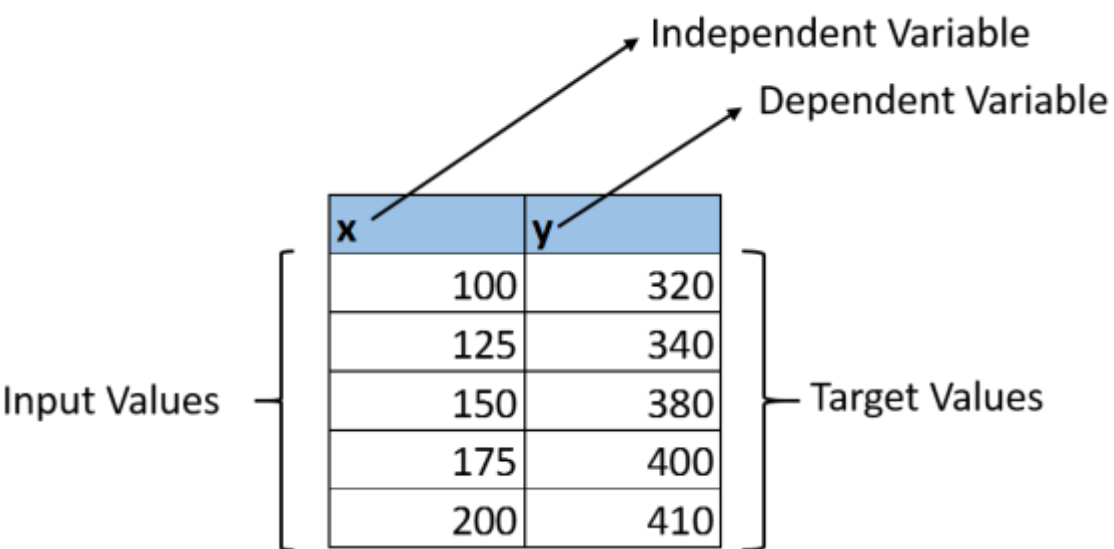
P10: P1, P11

P11: P2, P10, P12

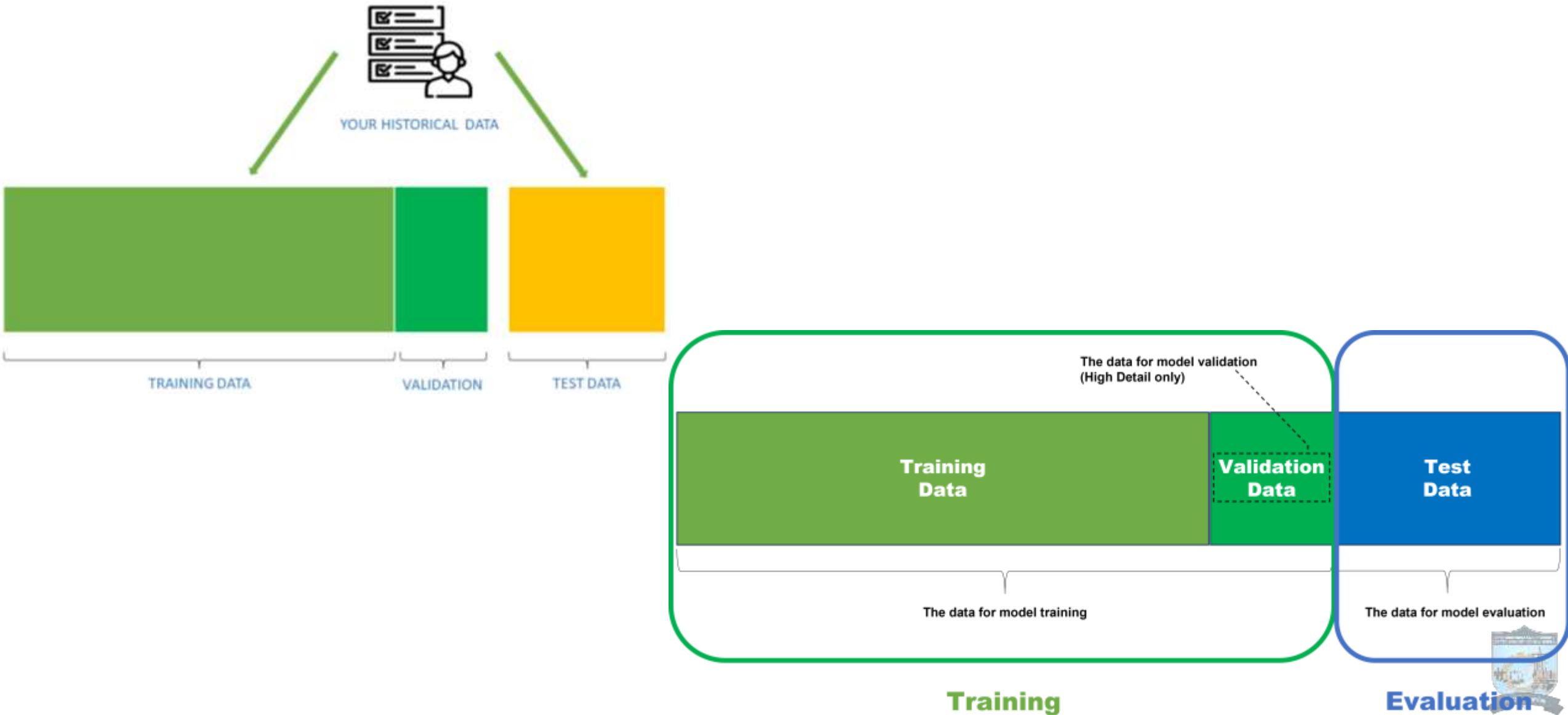
P12: P9, P11



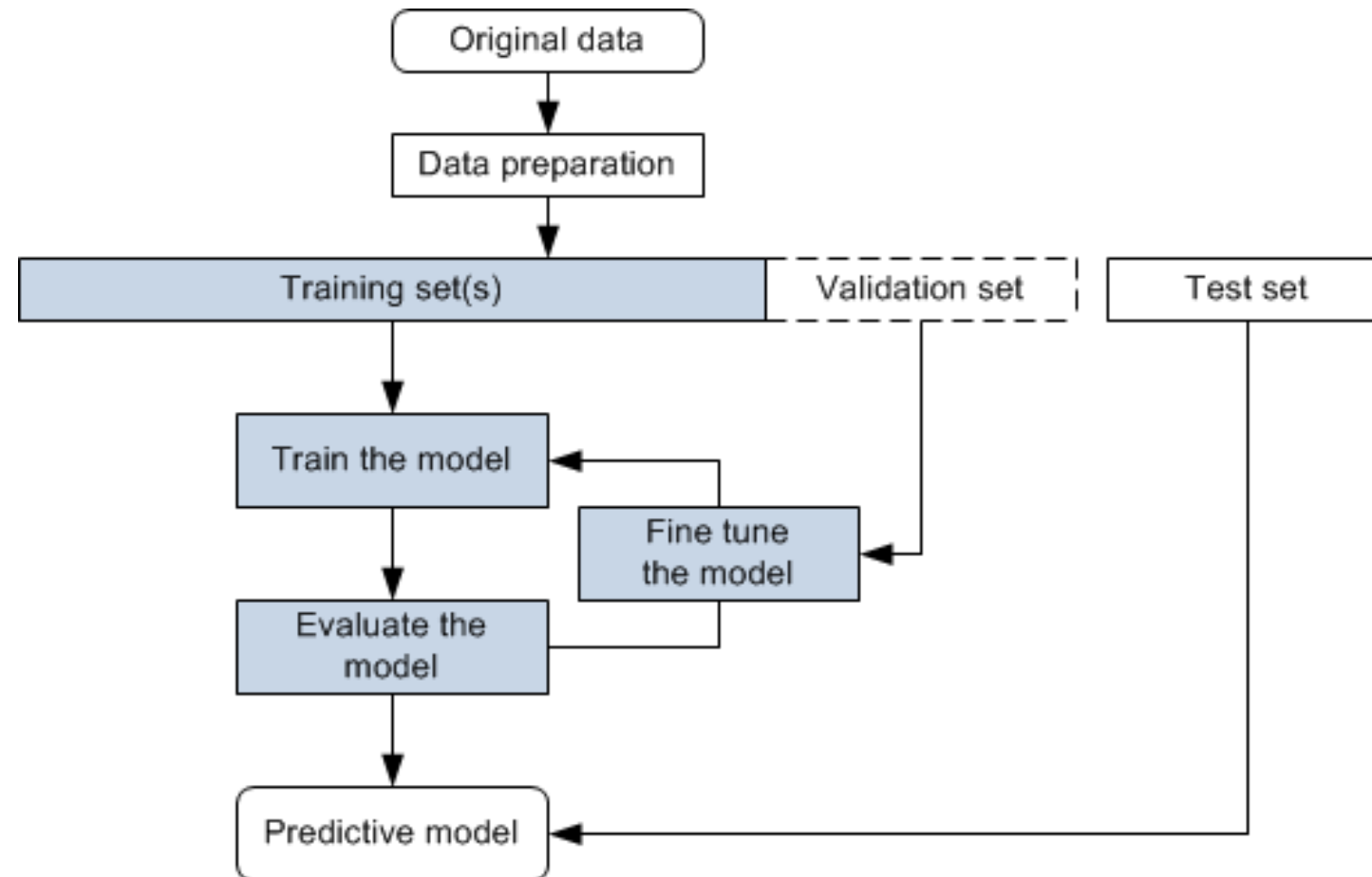
INPUT AND TARGET DATA IN MACHINE LEARNING



TRAINING AND TESTING DATA IN MACHINE LEARNING

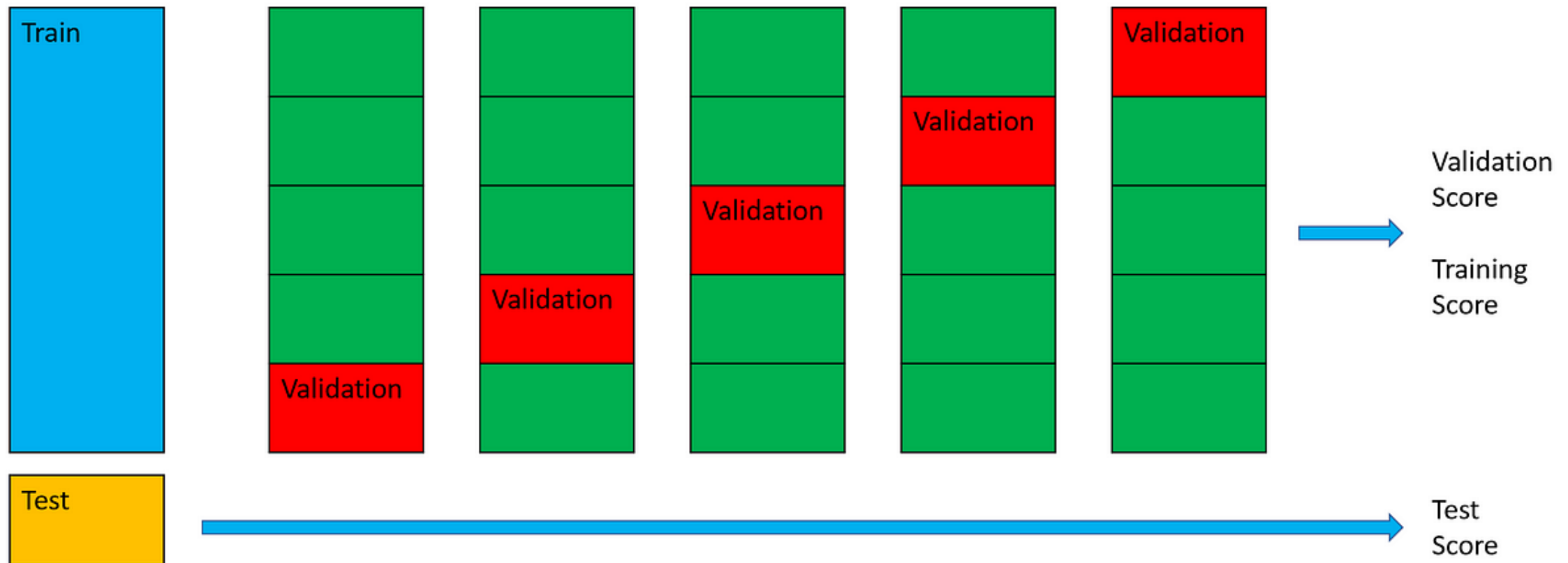


MODEL VALIDATION



MODEL VALIDATION

The process that helps us evaluate the performance of a trained model is called Model Validation. It helps us in validating the machine learning model performance on new or unseen data. It also helps us confirm that the model achieves its intended purpose.



MODEL VALIDATION

Types of Model Validation

Model validation is the step conducted post Model Training, wherein the effectiveness of the trained model is assessed using a testing dataset. This dataset may or may not overlap with the data used for model training.

Model validation can be broadly categorized into two main approaches based on how the data is used for testing:

1. In-Sample Validation: This approach involves the use of data from the same dataset that was employed to develop the model.

- **Holdout method:** The dataset is then divided into training set which is used to train the model and a hold out set which is used to test the performance of the model. This

MODEL VALIDATION

2. Out-of-Sample Validation

This approach relies on entirely different data from the data used for training the model. This gives a more reliable prediction of how accurate the model will be in predicting new inputs.

- **K-Fold Cross-validation:** The data is divided into k number of folds. The model is trained on $k-1$ folds and tested on the fold that is left. This is repeated k times, each time using a different fold for testing. This offers a more extensive analysis than the holdout method.
- **Leave-One-Out Cross-validation (LOOCV):** This is a form of k -fold cross validation where k is equal to the number of instances. Only one piece of data is not used to train the model. This is repeated for each data point. Unfortunately, LOOCV is also time consuming when dealing with large datasets.
- **Stratified K-Fold Cross-validation:** k -fold cross-validation: in this type of cross-validation each fold has the same ratio of classes/categories as the overall dataset.



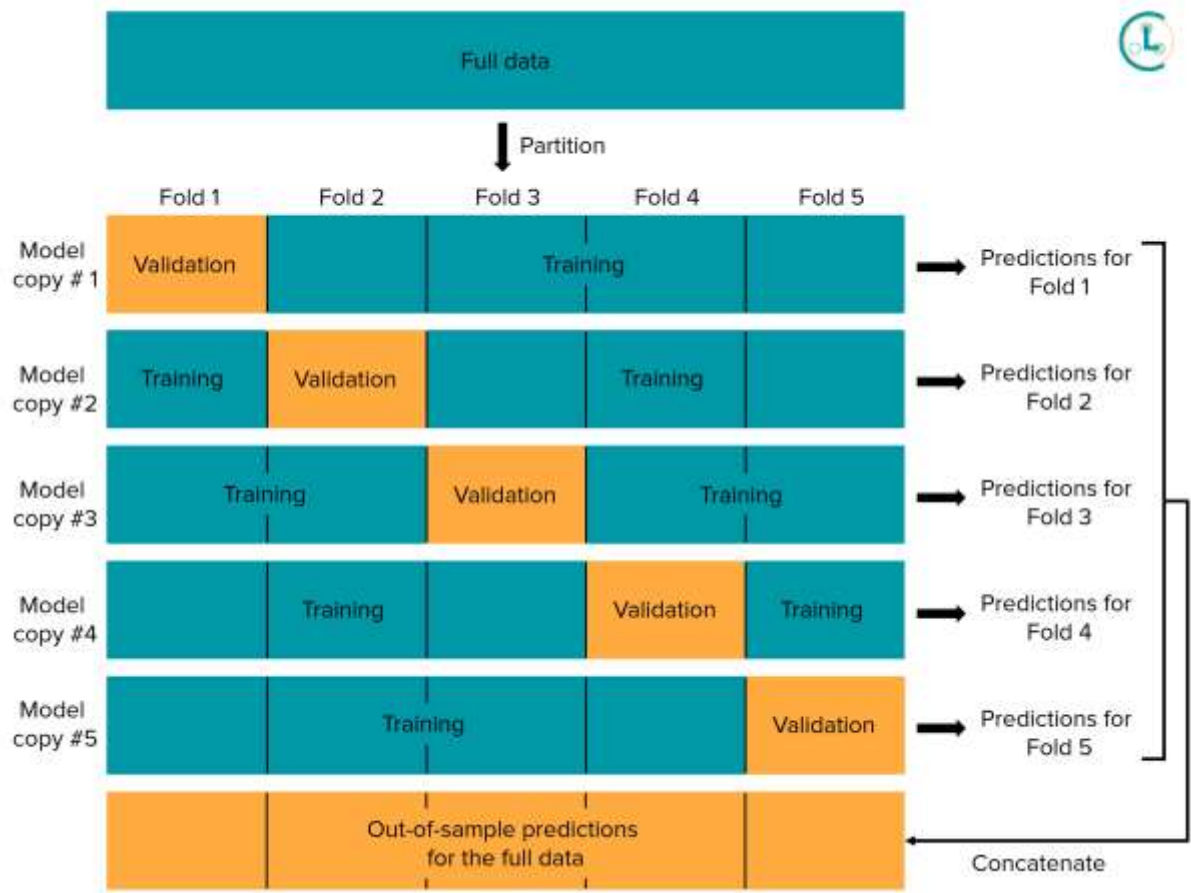
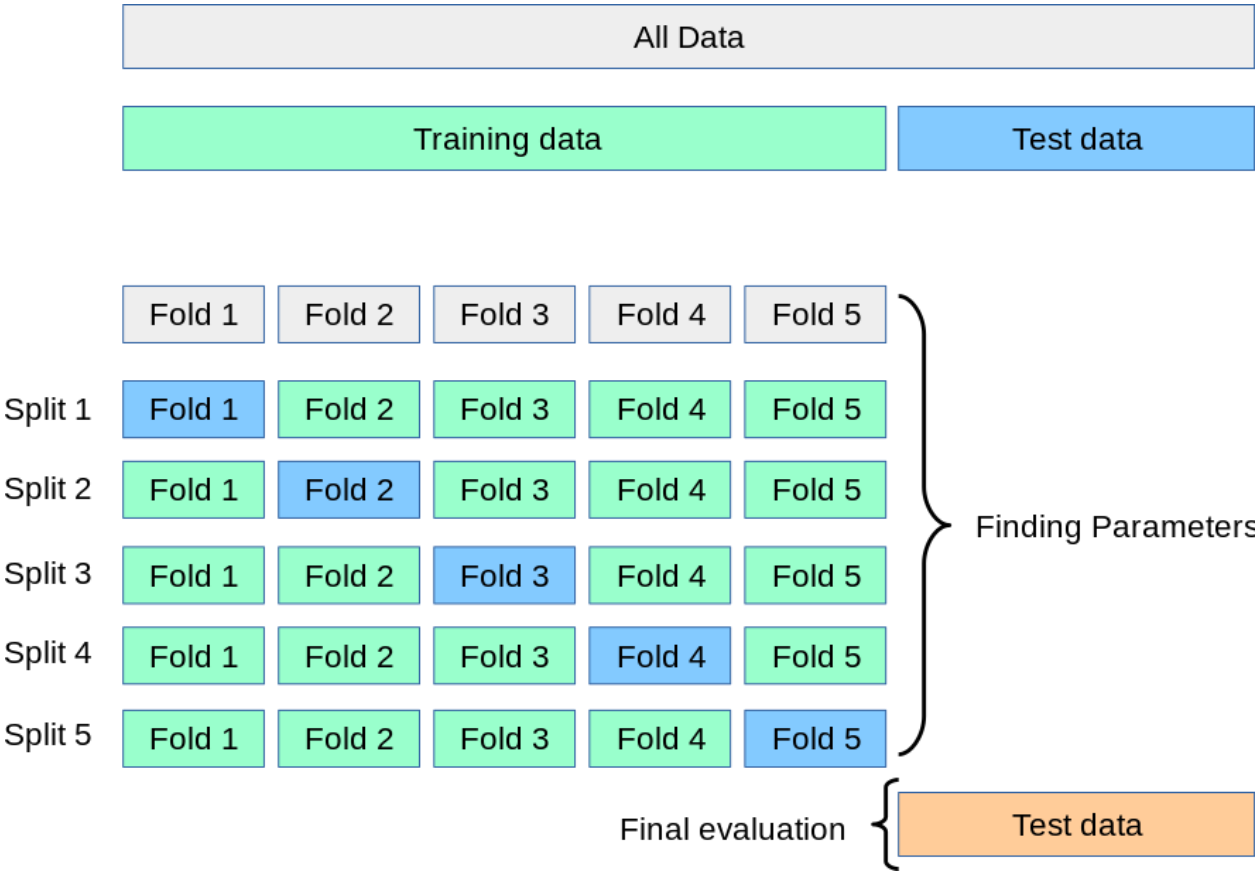
K-FOLD CROSS-VALIDATION

In **k-fold cross-validation**, the initial data are randomly partitioned into k mutually exclusive subsets or "folds," D_1, D_2, \dots, D_k , each of approximately equal size. Training and testing is performed k times.

- In iteration i , partition D_i is reserved as the test set, and the remaining partitions are collectively used to train the model.
- That is, in the first iteration, subsets D_2, \dots, D_k collectively serve as the training set to obtain a first model, which is tested on D_1 ; the second iteration is trained on subsets D_1, D_2, \dots, D_k and tested on D_2 ; and so on.
- Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing.



K-FOLD CROSS-VALIDATION



LEAVE ONE OUT CROSS-VALIDATION

LOOCV (Leave-One-Out Cross-Validation) is a model evaluation technique used to assess the performance of a machine learning model on small datasets. In LOOCV, one observation is used as the test set while the rest form the training set. This process is repeated for each data point in the dataset, resulting in n training-testing cycles, where n is the number of observations. The overall accuracy is averaged across all iterations.

- **Mathematical Expression**

In Leave-One-Out Cross-Validation (LOOCV), each individual observation serves once as the validation set, while the remaining $n-1$ observations are used for training. Instead of refitting the model n times, LOOCV for linear models can be computed efficiently using the following formula:

$$LOO_{Error} = \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{1 - h_{ii}} \right)^2$$

Where,

21-11-2025

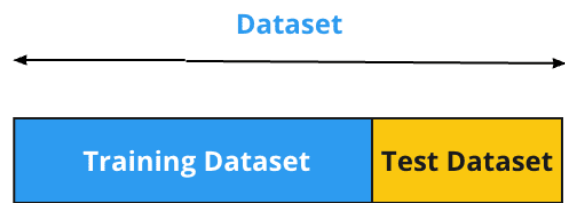
y_i = Actual value of the i^{th} observation

Introduction to AIML

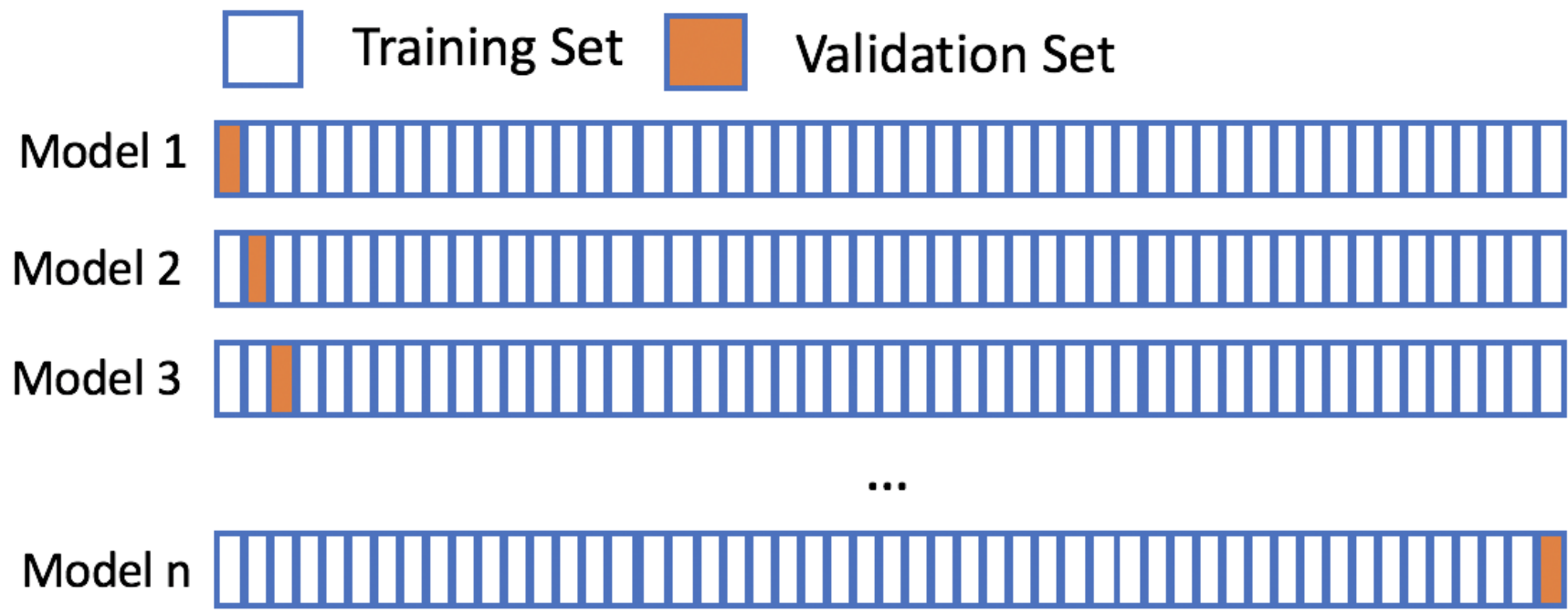
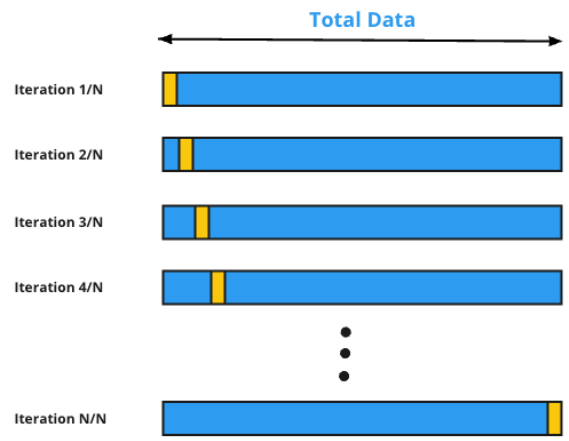
www.veerpreps.com



LEAVE ONE OUT CROSS-VALIDATION



LOOCV: Leave One Out Cross Validation



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Two types of labels are there.

1. **Positive tuples** (tuples of the main class of interest)
2. **Negative tuples** (all other tuples).

Given two classes, for example, the **positive tuples** may be *buys computer = yes* while the **negative tuples** are *buys computer = no*. Suppose we use our **classifier** on a test set of labeled tuples. **P** is the **number of positive tuples** and **N** is the **number of negative tuples**. For each tuple, we **compare** the **classifier's class label prediction** with the **tuple's known class label**.

There are four additional terms we need to know that are the "building blocks" used in computing many evaluation measures.

- **True positives (TP)**: These refer to the positive tuples that were correctly labeled by the classifier. Let *TP* be the number of true positives.
- **True negatives (TN)**: These are the negative tuples that were correctly labeled by the classifier. Let *TN* be the number of true negatives.
- **False positives (FP)**: These are the negative tuples that were incorrectly labeled as positive (e.g., tuples of class *buys computer = no* for which the classifier predicted *buys computer = yes*). Let *FP* be the number of false positives.
- **False negatives (FN)**: These are the positive tuples that were mislabeled as negative (e.g., tuples of class *buys computer = yes* for which the classifier predicted *buys computer = no*). Let *FN* be the number of false negatives.



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Measures	Formula
Accuracy, Recognition Rate	$\frac{TP + TN}{P + N}$
Error Rate, Misclassification Rate	$\frac{FP + FN}{P + N}$
Sensitivity, True Positive Rate, Recall	$\frac{TP}{P}$
Specificity, True Negative Rate	$\frac{TN}{N}$
Precision	$\frac{TP}{TP + FP}$
F, F_1 , F-Score, Harmonic mean of Precision and Recall	$\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$
F_β , Where β Is a non-negative real number	$\frac{(1 + \beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}}$

Actual Class

	Predicted Class		
	Yes	No	
Yes	TP	FN	P
No	FP	TN	N
Total	P'	N'	P + N



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Measures	Formula
Accuracy, Recognition Rate	$\frac{TP + TN}{P + N}$
Error Rate, Misclassification Rate	$\frac{FP + FN}{P + N}$
Sensitivity, True Positive Rate, Recall	$\frac{TP}{P}$
Specificity, True Negative Rate	$\frac{TN}{N}$
Precision	$\frac{TP}{TP + FP}$
F, F_1, F -Score, Harmonic mean of Precision and Recall	$\frac{2 \times Precision \times Recall}{Precision + Recall}$
F_β , Where β Is a non-negative real number	$\frac{(1 + \beta^2) \times Precision \times Recall}{\beta^2 \times Precision + Recall}$

Classes	buys computer = yes	buys computer = no	Total
buys computer = yes	6954	46	7000
buys computer = no	412	2588	3000
Total	7366	2634	10000

Classes	Cancer = yes	Cancer = no	Total
Cancer = yes	90	210	300
Cancer = no	140	9560	9700
Total	230	9770	10000



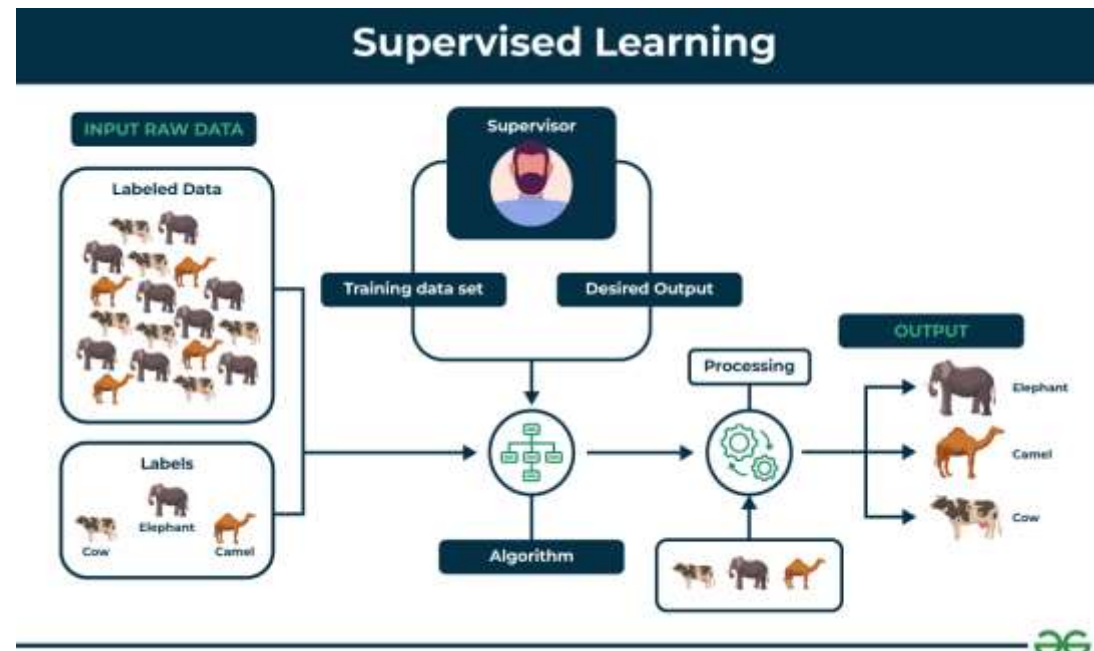
Artificial Intelligence and Machine Learning

MODULE IV

SUPERVISED LEARNING

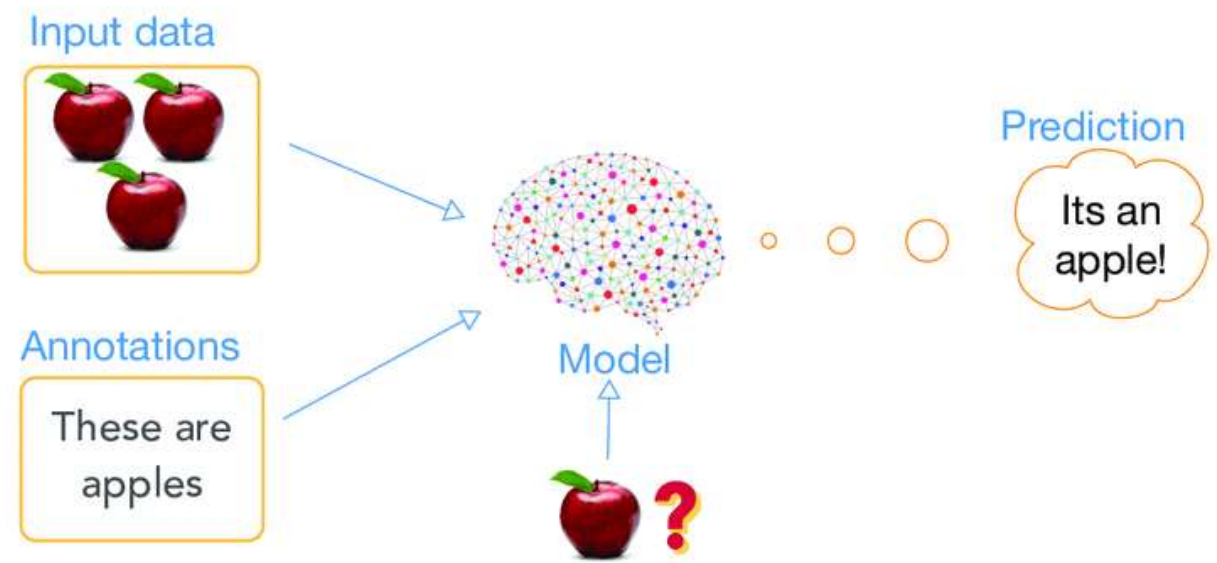
SUPERVISED LEARNING

- **Supervised machine learning** is a fundamental approach for machine learning and artificial intelligence.
- The process is like a teacher guiding a student—hence the term "supervised" learning.
- It is a type of machine learning where a model is trained on labeled data—meaning each input is paired with the correct output.
- The model learns by comparing its predictions with the actual answers provided in the training data. Over time, it adjusts itself to minimize errors and improve accuracy.
- The goal of supervised learning is to make accurate predictions when given new, unseen data.

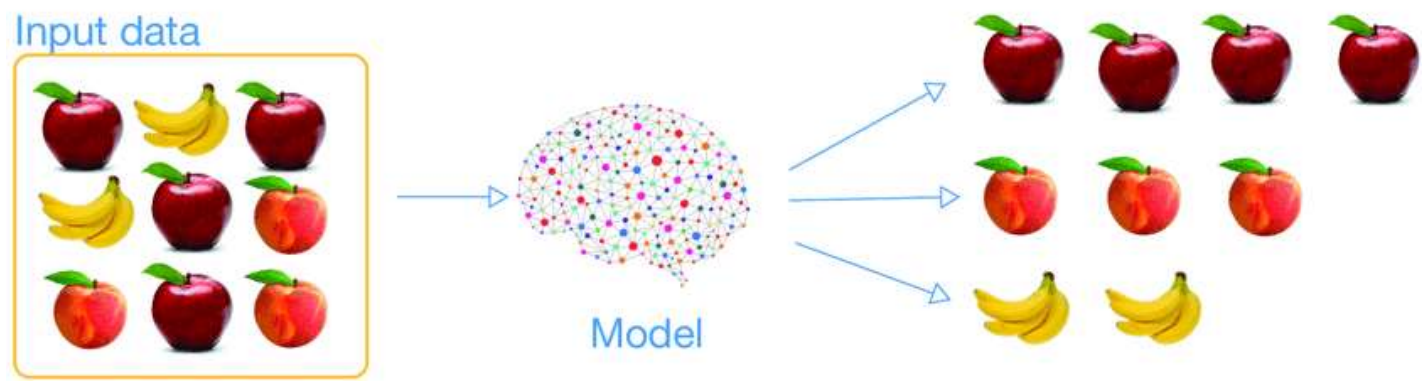


SUPERVISED LEARNING

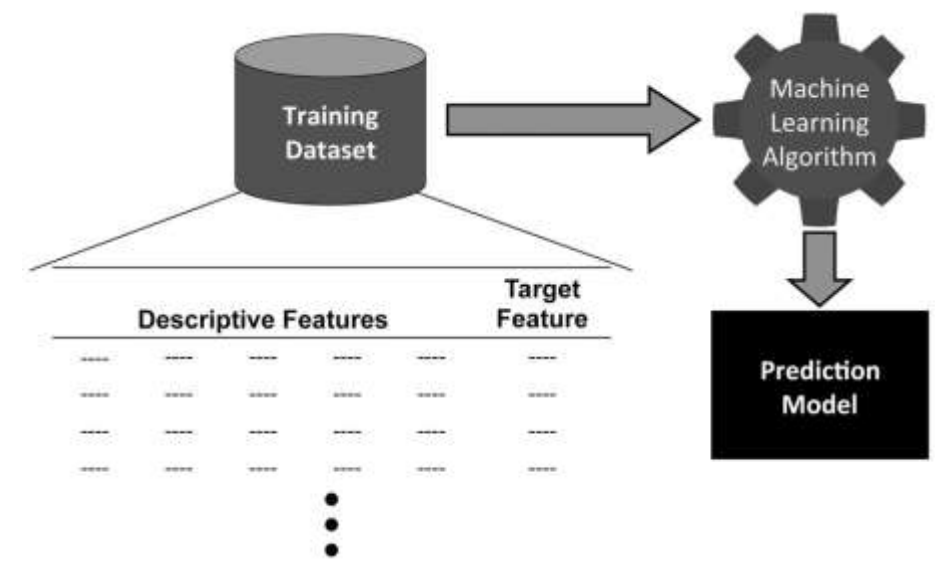
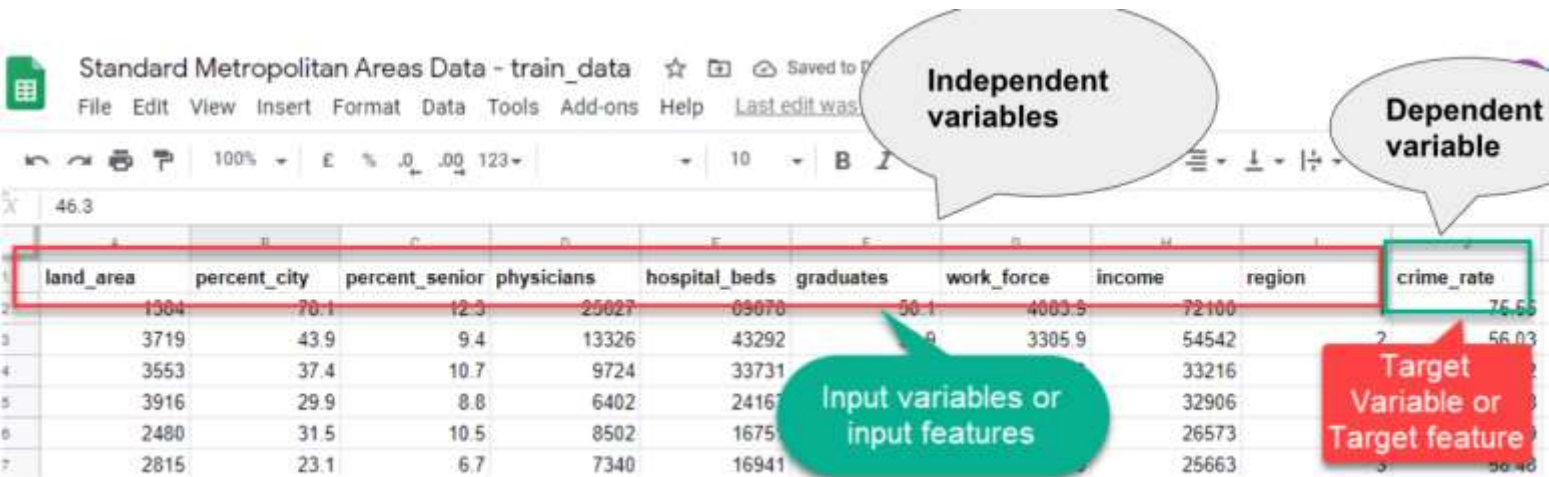
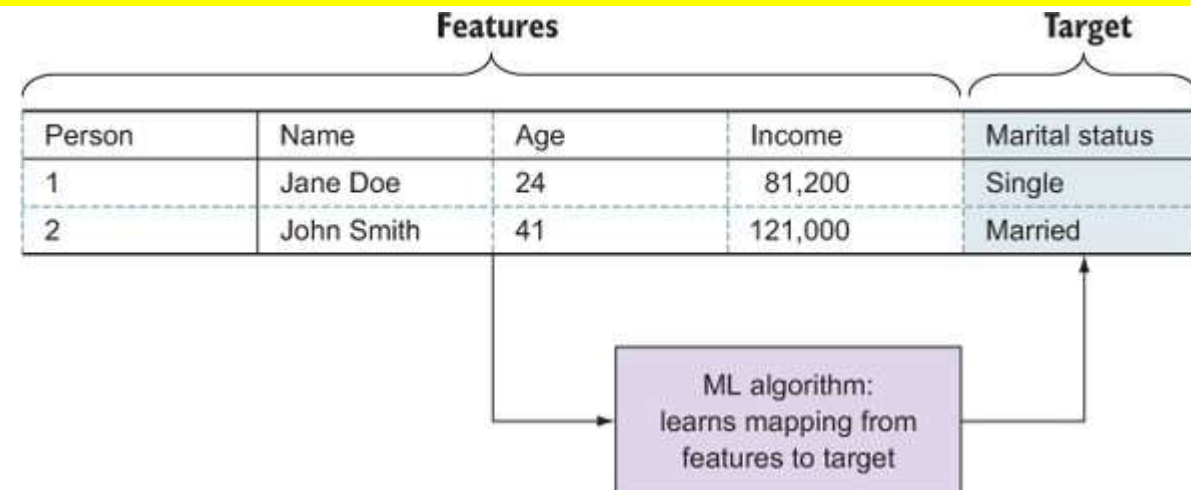
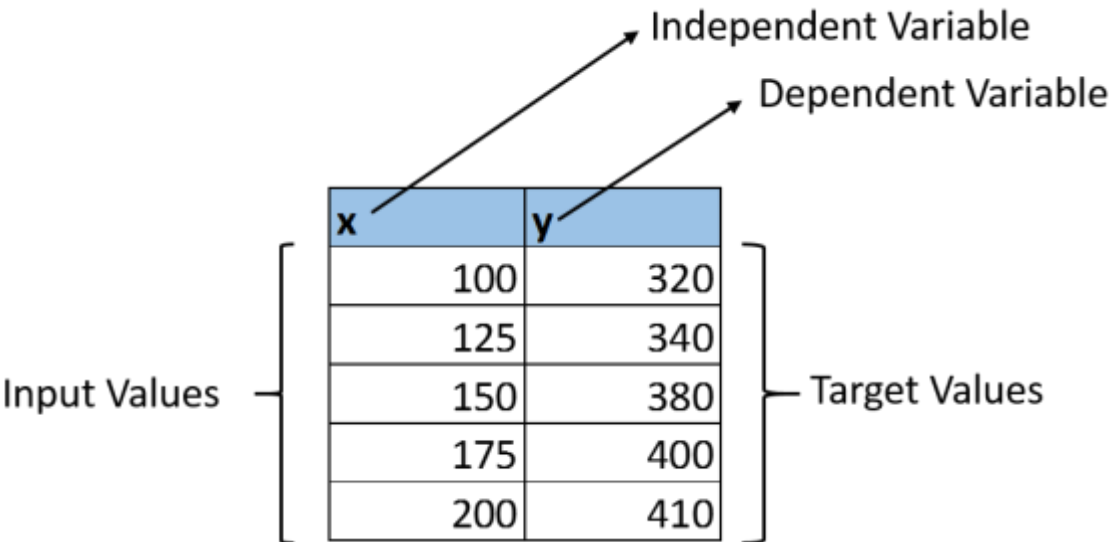
supervised learning



unsupervised learning



INPUT AND TARGET DATA IN MACHINE LEARNING



INPUT AND TARGET DATA IN MACHINE LEARNING

case ID		predictors			target
CUST_ID	CUST_GENDER	EDUCATION	OCCUPATION	AGE	AFFINITY_CARD
101501	F	Masters	Prof.	41	0
101502	M	Bach.	Sales	27	0
101503	F	HS-grad	Cleric.	20	0
101504	M	Bach.	Exec.	45	1
101505	M	Masters	Sales	34	1
101506	M	HS-grad	Other	38	0
101507	M	< Bach.	Sales	28	0
101508	M	HS-grad	Sales	19	0
101509	M	Bach.	Other	52	0
101510	M	Bach.	Sales	27	1

Classification

Index of example	Height	Color of hair	Color of eyes	Target Class
1	Low	Blond	Blue	+
2	Low	Brown	Blue	-
3	Tall	Brown	Hazel	-
4	Tall	Blond	Hazel	-
5	Tall	Brown	Blue	-
6	Low	Blond	Hazel	-
7	Tall	Red	Blue	+
8	Tall	Blond	Blue	+

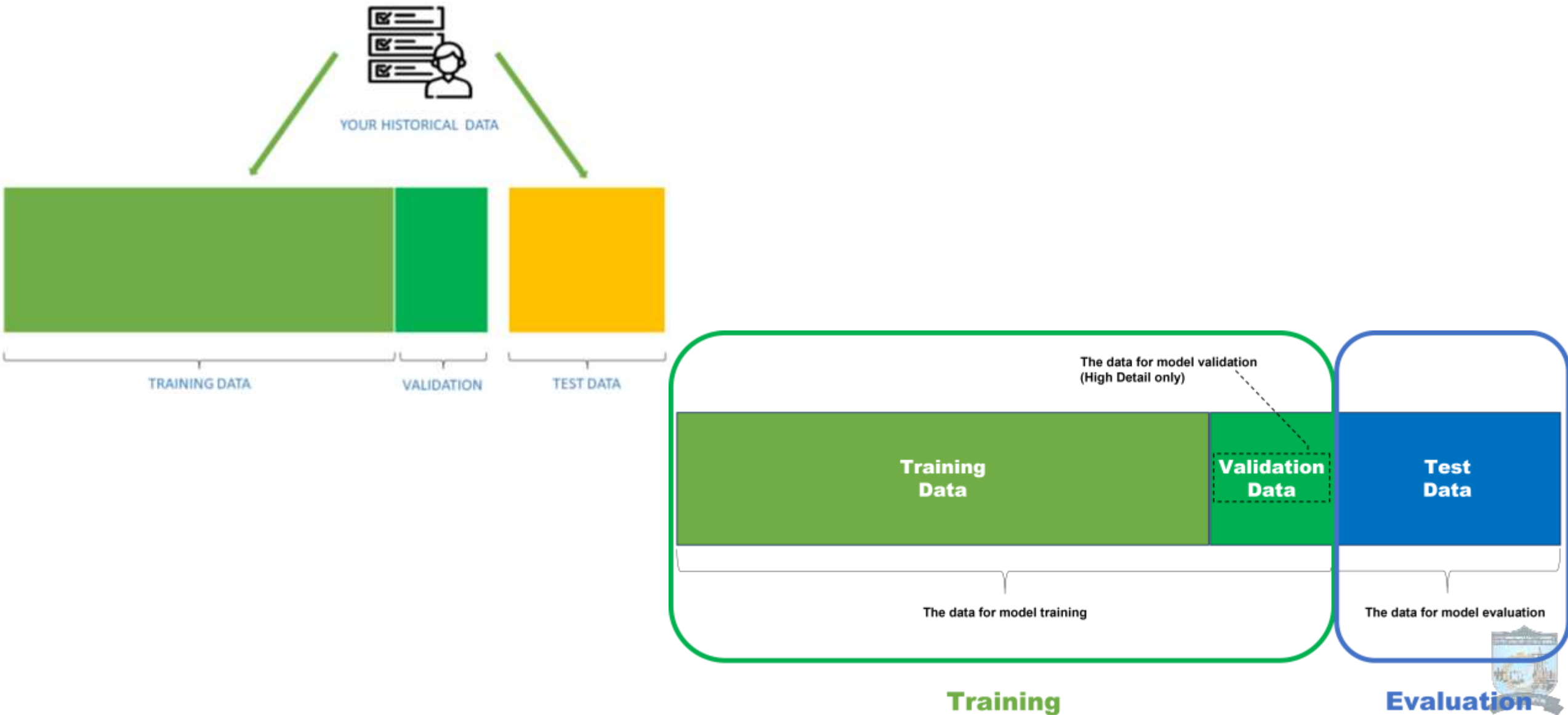
B	C	D	E
Mask	1	1	
Sales Person	Intelligence	Extroversion	\$ Sales/Week
1	89	21	\$ 2,625
2	93	24	\$ 2,700
3	91	21	\$ 3,100
4	122	23	\$ 3,150
5	115	27	\$ 3,175
6	100	18	\$ 3,100
7	98	19	\$ 2,700
8	105	16	\$ 2,475
9	112	23	\$ 3,625
10	109	28	\$ 3,525
11	130	20	\$ 3,225
12	104	25	\$ 3,450
13	104	20	\$ 2,425
14	111	26	\$ 3,025
15	97	28	\$ 3,625
16	115	29	\$ 2,750
17	113	25	\$ 3,150
18	88	23	\$ 2,600
19	108	19	\$ 2,525
20	101	16	\$ 2,650

Regression

	Study Hours	Prep Exams	Final Exam Score
Student 1	3	2	76
Student 2	7	6	88
Student 3	16	5	96
Student 4	14	2	90
Student 5	12	7	98
Student 6	7	4	80
Student 7	4	4	86
Student 8	19	2	89
Student 9	4	8	68
Student 10	8	4	75
Student 11	8	1	72
Student 12	3	3	76

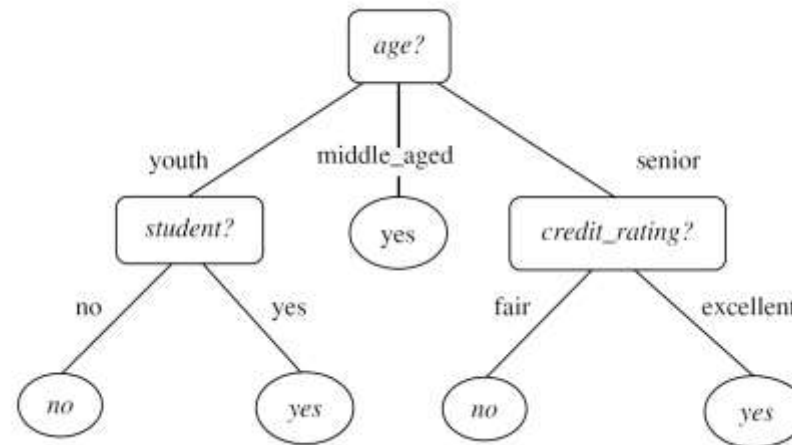


TRAINING AND TESTING DATA IN MACHINE LEARNING



DECISION TREE INDUCTION

Decision tree induction is the learning of decision trees from class-labeled training tuples. A **decision tree** is a flowchart-like tree structure, where each **internal node** (nonleaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (circular node) holds a class label. The topmost node in a tree is the **root** node.



- **"How are decision trees used for classification?"**

Given a tuple, **X**, for which the associated class label is unknown; the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

- **"Why are decision tree classifiers so popular?"**

The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery.

DECISION TREE INDUCTION

- During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as **ID3 (Iterative Dichotomiser)**. This work expanded on earlier work on *concept learning systems*, described by E. B. Hunt, J. Marin, and P. T. Stone.
- Quinlan later presented **C4.5 (a successor of ID3)**, which became a benchmark to which newer supervised learning algorithms are often compared.
- In 1984, a group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) published the book **Classification and Regression Trees (CART)**, which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples.
- **ID3, C4.5, and CART** adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts



DECISION TREE INDUCTION

- The algorithm is called with three parameters: *D*, *attribute list*, and *Attribute selection method*. We refer to *D* as a data partition.
- Initially, it is the complete set of training tuples and their associated class labels. The parameter *attribute list* is a list of attributes describing the tuples.
- *Attribute selection method* specifies a heuristic procedure for selecting the attribute that “best” discriminates the given tuples according to class.
- This procedure employs an attribute selection measure such as *information gain* or the *Gini index*.
- Whether the tree is strictly binary is generally driven by the attribute selection measure.
- Some attribute selection measures, such as the Gini index, enforce the resulting tree to be binary.
- Others, like information gain, do not, therein allowing multiway splits (i.e. two or



DECISION TREE ALGORITHM

Input:

- Data partition, D , which is a set of training tuples and their associated class labels;
- *attribute list*, the set of candidate attributes;
- *Attribute selection method*, a procedure to determine the splitting criterion that “best” partitions the data tuples into individual classes. This criterion consists of a *splitting attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

Method:

- (1) create a node N ;
- (2) **if** tuples in D are all of the same class, C , **then**
- (3) return N as a leaf node labeled with the class C ;
- (4) **if** *attribute list* is empty **then**
- (5) return N as a leaf node labeled with the majority class in D ; // majority voting
- (6) apply **Attribute selection method**(D , *attribute list*) to **find** the “best” *splitting criterion*;
- (7) label node N with *splitting criterion*;
- (8) **if** *splitting attribute* is discrete-valued **and** multiway splits allowed **then** // not restricted to binary trees
- (9) *attribute list* \leftarrow *attribute list* - *splitting attribute*; // remove *splitting attribute*
- (10) **for each** outcome j of *splitting criterion*
- // partition the tuples and grow subtrees for each partition
- (11) let D_j be the set of data tuples in D satisfying outcome j ; // a partition
- (12) **if** D_j is empty **then**
- (13) attach a leaf labeled with the majority class in D to node N ;
- (14) **else** attach the node returned by **Generate decision tree**(D_j , *attribute list*) to node N ;
- endfor**
- (15) return N ;



DECISION TREE ALGORITHM - ATTRIBUTE SELECTION MEASURES

An **attribute selection measure** is a heuristic for selecting the splitting criterion that "best" separates a given data partition, D , of class-labeled training tuples into individual classes. If we were to split D into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all the tuples that fall into a given partition would belong to the same class). Conceptually, the "best" splitting criterion is the one that most closely results in **Information Gain**. Attribute selection measures are also known as **splitting rules** because they determine how the tuples at a given node are to be split. ID3 uses **information gain** as its attribute selection measure. This measure is based on pioneering work by Claude Shannon on information theory, which studied the value or "information content" of messages. Let node N represent or hold the tuples of partition D . The attribute with the highest information gain is chosen as the splitting attribute for node N .

The expected information needed to classify a tuple in D is given by

$$Info(D) = - \sum_{i=1}^m p_i \log_2(p_i)$$


DECISION TREE ALGORITHM - ATTRIBUTE SELECTION MEASURES

Suppose we were to partition the tuples in D on some attribute A having v distinct values, $\{a_1, a_2, \dots, a_v\}$, as observed from the training data.

- If A is discrete-valued, these values correspond directly to the v outcomes of a test on A . Attribute A can be used to split D into v partitions or subsets, $\{D_1, D_2, \dots, D_v\}$, where D_j contains those tuples in D that have outcome a_j of A . These partitions would correspond to the branches grown from node N .
- Ideally, we would like this partitioning to produce an exact classification of the tuples. That is, we would like for each partition to be pure.
- However, it is quite likely that the partitions will be impure (e.g., where a partition may contain a collection of tuples from different classes rather than from a single class).

How much more information would we still need $|D_j|$ (after the partitioning) to arrive at an exact classification? This amount is measured by

$$Info_A(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} \times Info(D_j)$$


DECISION TREE ALGORITHM – ATTRIBUTE SELECTION MEASURES

Information gain is defined as the difference between the original information requirement (i.e., based on just the proportion of classes) and the new requirement (i.e., obtained after partitioning on A):

$$\text{Gain}(A) = \text{Info}(D) - \text{Info}_A(D)$$

Gain(A) tells us how much would be gained by branching on **A**. It is the expected **reduction in the information requirement** caused by knowing the value of **A**.

The attribute A with the **highest information gain**, **Gain(A)**, is chosen as the splitting attribute at node N . This is equivalent to saying that we want to **partition** on the **attribute A** that would do the “**best classification**,” so that the amount of information still required to finish **classifying** the tuples is minimal (i.e., **minimum Info_A(D)**).



DECISION TREE ALGORITHM - EXAMPLE

RID	Age	Income	Student	Credit_Rating	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D10	senior	medium	Yes	fair	Yes
D11	youth	medium	Yes	excellent	Yes
D12	middle aged	medium	No	excellent	Yes



DECISION TREE ALGORITHM - EXAMPLE

RI D	Age	Income	Student	Credit_Rat ing	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D1 0	senior	medium	Yes	fair	Yes
D1 1	youth	medium	Yes	excellent	Yes
D1 2	middle aged	medium	No	excellent	Yes
D1 3	middle aged	high	Yes	fair	Yes
D1 4	senior	medium	No	excellent	No

Total records $N = 14$.

yes = 9

no = 5

Entropy of the whole Dataset $H(S)$:

$$H(S) = -p_{yes} \log_2 p_{yes} - p_{no} \log_2 p_{no}$$

Compute the proportions:

$$p_{yes} = \frac{9}{14} \approx 0.64285, \quad p_{no} = \frac{5}{14} \approx 0.35714$$

Compute terms:

$$-p_{yes} \log_2 p_{yes} = -0.64285 \log_2(0.64285) \approx 0.409$$

$$-p_{no} \log_2 p_{no} = -0.35714 \log_2(0.35714) \approx 0.531$$

Entropy of the Dataset

$$H(S) = 0.409 + 0.531 = 0.940 \text{ bits}$$

DECISION TREE ALGORITHM - EXAMPLE

RI D	Age	Income	Student	Credit_Rat ing	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D1 0	senior	medium	Yes	fair	Yes
D1 1	youth	medium	Yes	excellent	Yes
D1 2	middle aged	medium	No	excellent	Yes
D1 3	middle aged	high	Yes	fair	Yes
D1 4	senior	medium	No	excellent	No

Information gain for each attribute at the root

Age

- youth: 5 (yes=2, no=3) $\rightarrow -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}$

$$H(youth) = 0.9709505945$$

- middle_aged: 4 (yes=4, no=0) \rightarrow

$$H(middle_aged) = 0.0$$

- senior: 5 (yes=3, no=2) \rightarrow

$$H(senior) = 0.9709505945$$

Weighted average entropy of Age:

$$Info_{Age}(D) = \frac{5}{14} \cdot 0.9709505945 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.9709505945 = 0.694$$

$$Gain(Age) = Info(D) - Info_{Age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

DECISION TREE ALGORITHM - EXAMPLE

RI D	Age	Income	Student	Credit_Rat ing	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D1 0	senior	medium	Yes	fair	Yes
D1 1	youth	medium	Yes	excellent	Yes
D1 2	middle aged	medium	No	excellent	Yes
D1 3	middle aged	high	Yes	fair	Yes
D1 4	senior	medium	No	excellent	Yes

Income ∈ {high, medium, low}

high: 4 (yes =2, no =2)

$$H = -\frac{2}{4}\log_2\frac{2}{4} - \frac{2}{4}\log_2\frac{2}{4} = 1.0000000000$$

medium: 6 (yes =4, no =2)

$$H = -\frac{4}{6}\log_2\frac{4}{6} - \frac{2}{6}\log_2\frac{2}{6} = 0.9182958341$$

low: 4 (yes =3, no =1)

$$H = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 0.8112781245$$

Weighted average:

$$Info_{Income}(D) = \frac{4}{14} \cdot 1.0 + \frac{6}{14} \cdot 0.9182958341 + \frac{4}{14} \cdot 0.8112781245 = 0.911$$

$$Gain(Income) = Info(D) - Info_{Income}(D) = 0.940 - 0.911 = 0.029 \text{ bits.}$$



DECISION TREE ALGORITHM - EXAMPLE

RI D	Age	Income	Student	Credit_Rat ing	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D1 0	senior	medium	Yes	fair	Yes
D1 1	youth	medium	Yes	excellent	Yes
D1 2	middle aged	medium	No	excellent	Yes
D1 3	middle aged	high	Yes	fair	Yes
D1 4	senior	medium	No	excellent	Yes

student ∈ {yes, no}

yes: 7 (yes =6, no =1)

$$H = -\frac{6}{7}\log_2\frac{6}{7} - \frac{1}{7}\log_2\frac{1}{7} = 0.5916727786$$

no: 7 (yes =3, no =4)

$$H = -\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} = 0.9852281360$$

Weighted average:

$$Info_{Student}(D) = \frac{7}{14} \cdot 0.5916727786 + \frac{7}{14} \cdot 0.9852281360 = 0.7884504573$$

$$Gain(Student) = Info(D) - Info_{Student}(D) = 0.940 - 0.788 = 0.151 \text{ bits.}$$



DECISION TREE ALGORITHM - EXAMPLE

RI D	Age	Income	Student	Credit_Rat ing	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D1 0	senior	medium	Yes	fair	Yes
D1 1	youth	medium	Yes	excellent	Yes
D1 2	middle aged	medium	No	excellent	Yes
D1 3	middle aged	high	Yes	fair	Yes
D1 4	senior	medium	No	excellent	Yes

Credit_Rating ∈ {fair, excellent}

fair: 8 (yes =6, no =2)

$$H = -\frac{6}{8}\log_2\frac{6}{8} - \frac{2}{8}\log_2\frac{2}{8} = 0.8112781245$$

excellent: 6 (yes =3, no =3)

$$H = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1.0000000000$$

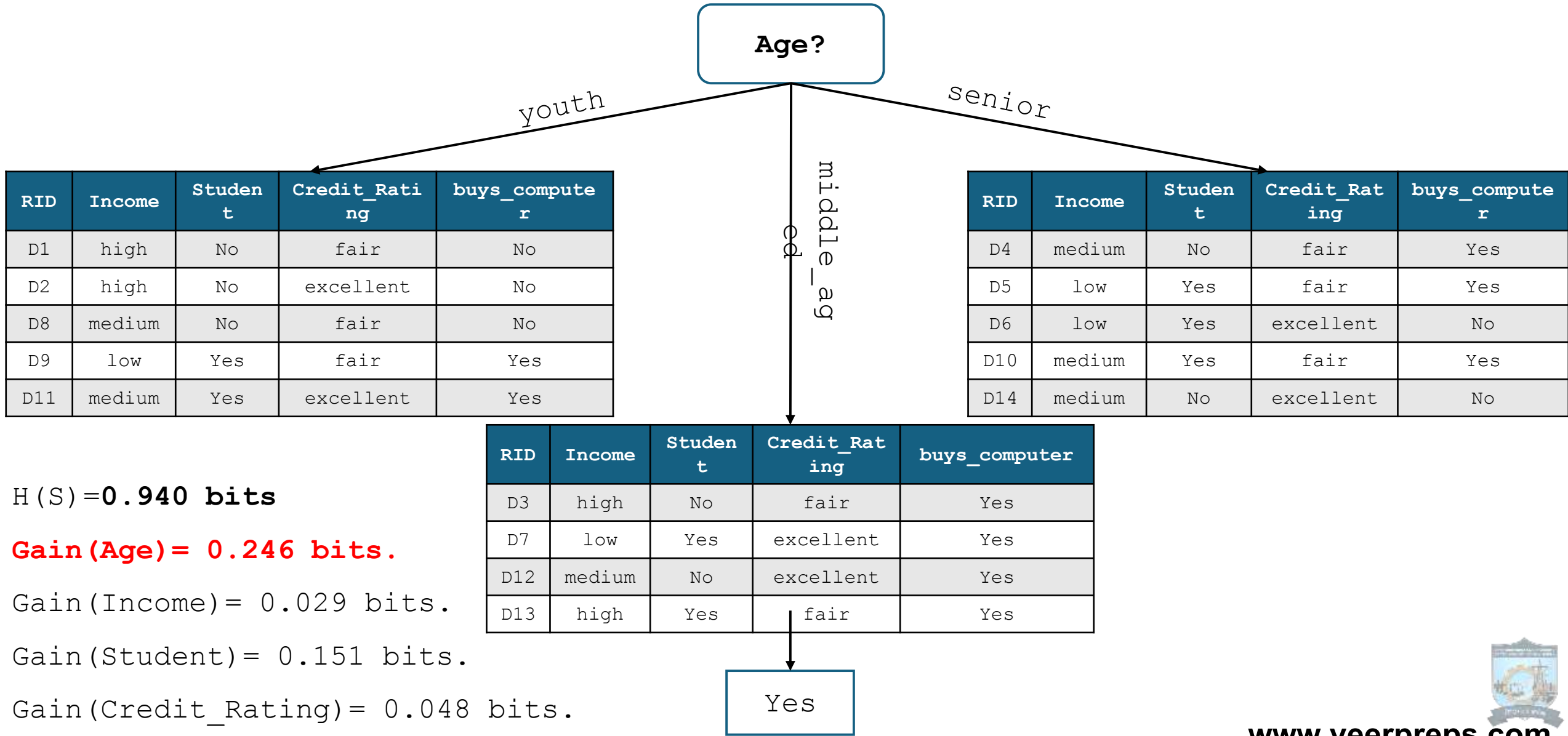
Weighted average:

$$Info_{Credit_Rating}(D) = \frac{8}{14} \cdot 0.8112781245 + \frac{6}{14} \cdot 1.0 = 0.8921589283$$

$$Gain(Credit_Rating) = Info(D) - Info_{Credit_Rating}(D) = 0.940 - 0.892 = 0.048 \text{ bits.}$$



DECISION TREE ALGORITHM - EXAMPLE



DECISION TREE ALGORITHM - EXAMPLE

Age = youth (5 records: yes =2, no =3)

Base entropy:

$$H(youth) = 0.9709505945$$

• Evaluate remaining attributes (Student, Income, Credit_Rating)

a) Student (yes/no) – within youth

- yes: 2 (yes =2, no =0) → $H = 0$
- no: 3 (yes =0, no =3) → $H = 0$

Weighted average:

$$Info_{Student}(youth) = \frac{2}{5}(0) + \frac{3}{5}(0) = 0$$

$$Gain(youth, Student) = 0.9709505945 - 0 = \boxed{0.9709505945}$$

b) Income – within youth

- high: 2 (yes =0, no =2) → $H = 0$
- medium: 2 (yes =1, no =1) → $H = 1.0$
- low: 1 (yes =1, no =0) → $H = 0$

Weighted average:

$$Info_{Income}(youth) = \frac{2}{5}(0) + \frac{2}{5}(1) + \frac{1}{5}(0) = 0.4$$

$$Gain(youth, Income) = 0.9709505945 - 0.4 = 0.5709505945$$

RID	Income	Student	Credit_Rating	buys_computer
D1	high	No	fair	No
D2	high	No	excellent	No
D8	medium	No	fair	No
D9	low	Yes	fair	Yes
D11	medium	Yes	excellent	Yes



DECISION TREE ALGORITHM - EXAMPLE

RID	Income	Student	Credit_Rating	buys_computer
D1	high	No	fair	No
D2	high	No	excellent	No
D8	medium	No	fair	No
D9	low	Yes	fair	Yes
D11	medium	Yes	excellent	Yes

c) Credit – within youth

- fair: 3 (yes = 1, no = 2) $\rightarrow H = 0.9182958341$
- excellent: 2 (yes = 1, no = 1) $\rightarrow H = 1.0$

$$Info_{Credit_Rating}(youth) = \frac{3}{5}(0.9182958341) + \frac{2}{5}(1.0) = 0.9509775004$$

$$Gain(youth, Credit_Rating) = 0.0199730940$$

Decision at Age=youth: pick **Student** (highest gain; makes both leaves pure):

- Student = yes \rightarrow **YES**
- Student = no \rightarrow **NO**



DECISION TREE ALGORITHM - EXAMPLE

Age = senior (5 records: yes = 3, no = 2)

Base entropy:

$$H(senior) = 0.9709505945$$

Evaluate remaining attributes (**Credit**, **Income**, **Student**)

a) **Credit – within senior**

- fair: 3 (yes = 3, no = 0) → $H = 0$
- excellent: 2 (yes = 0, no = 2) → $H = 0$

$$Info_{Credit}(senior) = \frac{3}{5}(0) + \frac{2}{5}(0) = 0$$

$$Gain(senior\ Credit) = 0.9709505945 - 0 = \boxed{0.9709505945}$$

Perfect split → choose **Credit** here.

b) **Income – within senior (for completeness)**

- medium: 3 (yes = 2, no = 1) → $H = 0.9182958341$
- low: 2 (yes = 1, no = 1) → $H = 1.0$

$$Info_{Income}(senior) = \frac{3}{5}(0.9182958341) + \frac{2}{5}(1.0) = \boxed{0.9509775004}$$

$$Gain(senior\ Income) = 0.9709505945 - 0.9509775004 = \boxed{0.0199730940}$$

RID	Income	Student	Credit_Rating	buys_computer
D4	medium	No	fair	Yes
D5	low	Yes	fair	Yes
D6	low	Yes	excellent	No
D10	medium	Yes	fair	Yes
D14	medium	No	excellent	No



DECISION TREE ALGORITHM - EXAMPLE

Age = senior (5 records: yes = 3, no = 2)

Base entropy:

$$H(senior) = 0.9709505945$$

Evaluate remaining attributes (Credit, Income, Student)

c) Student – within senior

- yes: 3 (yes = 2, no = 1) → $H = 0.9182958341$
- no: 2 (yes = 1, no = 1) → $H = 1.0$

$$\text{Info}_{Student}(senior) = \frac{3}{5}(0.9182958341) + \frac{2}{5}(1.0) = \boxed{0.9509775004}$$

$$\text{Gain}(senior \text{ Student}) = 0.9709505945 - 0.9509775004 = \boxed{0.0199730940}$$

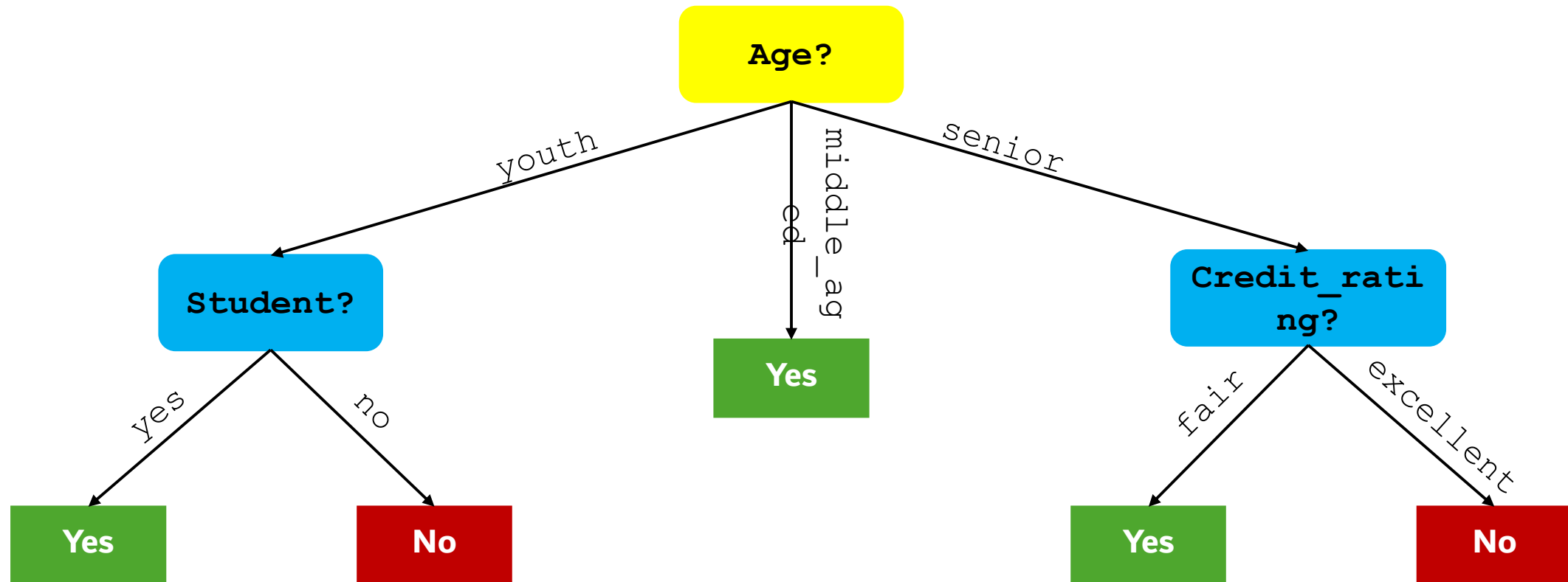
Decision at Age=senior: pick Credit (highest gain; makes both leaves pure):

- Credit = fair → YES
- Credit = excellent → NO

RID	Income	Student	Credit_Rating	buys_computer
D4	medium	No	fair	Yes
D5	low	Yes	fair	Yes
D6	low	Yes	excellent	No
D10	medium	Yes	fair	Yes
D14	medium	No	excellent	No



DECISION TREE ALGORITHM - EXAMPLE



Test Tuple 1: (age = youth, income = medium, student = yes, credit = fair)

Test Tuple 2: (age = senior, income = low, student = yes, credit = excellent)



PROBABILITY

Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

1. $0 \leq P(A) \leq 1$, where $P(A)$ is the probability of an event A .

2. $P(A) = 0$, indicates total uncertainty in an event A .

3. $P(A) = 1$, indicates total certainty in an event A .

$$\text{Probability of Occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- $P(\neg A)$ = probability of a not happening event.
- $P(\neg A) + P(A) = 1$.

Conditional probability

The probability of an **event A** based on the occurrence of another **event B** is termed conditional Probability. It is denoted as $P(A|B)$ and represents the **probability of A when event B has already happened**. Conditional probability is a probability of occurring an event when another event has already happened.

Let's suppose, we want to calculate the event A when event B has already occurred, "**The probability of A under the conditions of B**", it can be written as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$



PROBABILITY

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

$P(A \cap B)$ = Joint probability of A and B
probability of B.

$P(B)$ = Marginal

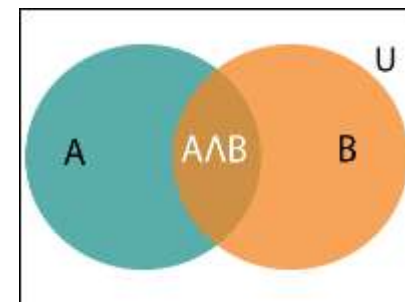
Joint probability: When the probability of two more events occurring together and at the same time is measured it is marked as Joint Probability. For two events A and B, it is denoted by $P(A \cap B)$ or $P(A \wedge B)$.

probability of B, then it will be given as:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of $P(A \cap B)$ by $P(B)$.

In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?



BAYES CLASSIFICATION METHODS

Bayesian classifiers are statistical classifiers. They can predict class membership probabilities such as the probability that a given tuple belongs to a particular class. Bayesian classification is based on Bayes' theorem

Bayes' Theorem

Let X be a data tuple. In Bayesian terms, X is considered "**evidence**". As usual, it is described by measurements made on a **set of n attributes**. Let H be some hypothesis such as that the data tuple X belongs to a specified **class C** . For classification problems, we want to determine $P(H|X)$, the probability that the hypothesis H holds given the "**evidence**" or observed data tuple X . In other words, we are looking for the probability that tuple X belongs to class C given that H is true. Now the attribute description of X .

$$P(H|X) = \frac{P(X|H)P(H)}{P(X|H)P(H) + P(X|\neg H)P(\neg H)} = P(H|X) = \frac{P(X|H)P(H)}{p(x)}$$

$P(H|X)$ is **posterior probability** or a *posteriori* probability, of H conditioned on X .

$P(H)$ is the **prior probability**, or a *priori* probability.

$P(X|H)$ is the **Likelihood**.



NAÏVE BAYES CLASSIFIER - ALGORITHM

1. Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n -dimensional attribute vector, $\mathbf{X} = (x_1, x_2, \dots, x_n)$ depicting n measurements made on the tuple from n attributes, respectively, A_1, A_2, \dots, A_n .
2. Suppose that there are m classes, C_1, C_2, \dots, C_m . Given a tuple, \mathbf{X} , the classifier will predict that \mathbf{X} belongs to the class having the highest posterior probability, conditioned on \mathbf{X} . That is, the naïve Bayesian classifier predicts that tuple \mathbf{X} belongs to the class C_i if and only if

$$P(C_i|\mathbf{X}) > P(C_j|\mathbf{X}) \quad \text{for } 1 \leq j \leq m, j \neq i$$

Thus, we maximize $P(C_i|\mathbf{X})$. The class C_i for which $P(C_i|\mathbf{X})$ is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem,

$$P(C_i|\mathbf{X}) = \frac{P(\mathbf{X}|C_i)P(C_i)}{P(\mathbf{X})}$$

3. As $P(\mathbf{X})$ is constant for all classes, only $P(\mathbf{X}|C_i)$ needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = \dots = P(C_m)$, and we would therefore maximize $P(\mathbf{X}|C_i)$. Otherwise, we maximize $P(\mathbf{X}|C_i)P(C_i)$. Note that the class prior probabilities may be estimated by $P(C_i) = |C_{i,D}|/|D|$, where $|C_{i,D}|$ is the number of training tuples of class C_i in D .

NAÏVE BAYES CLASSIFIER - ALGORITHM

4. Given data sets with many attributes, it would be extremely computationally expensive to compute $P(X|C_i)$. To reduce computation in evaluating $P(X|C_i)$, the naïve assumption of **class-conditional independence** is made. This presumes that the attributes' values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$P(X|C_i) = \prod_{k=1}^n P(x_k|C_i) = P(x_1|C_i) \times P(x_2|C_i) \times \dots \times P(x_n|C_i)$$

We can easily estimate the probabilities $P(x_1|C_i), P(x_2|C_i), \dots, P(x_n|C_i)$ from the training tuples. Recall that here x_k refers to the value of attribute A_k for tuple \mathbf{X} . For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute $P(X|C_i)$, we consider the following:

- a) If A_k is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D having the value x_k for A_k , divided by $|C_{i,D}|$, the number of tuples of class C_i in D .
- b) If A_k is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation σ , defined by

$$g(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

μ = mean of the values of attribute A_k
 σ = standard deviation of the values of attribute A_k

So that,



NAÏVE BAYES CLASSIFIER - EXAMPLE

Test Tuple: $X = (age = youth, income = medium, student = yes, credit_rating = fair)$

RI D	Age	Income	Student	Credit_Rat ing	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D1 0	senior	medium	Yes	fair	Yes
D1 1	youth	medium	Yes	excellent	Yes
D1 2	middle aged	medium	No	excellent	Yes
D1 3	middle aged	high	Yes	fair	Yes
D1	senior	medium	No	excellent	No

The prior probability of each class, can be computed based on the training tuples:

$$P(buys_computer = yes) = 9 / 14 = 0.643$$

$$P(buys_computer = no) = 5 / 14 = 0.357$$

Compute the conditional probabilities

RI D	Age	Class: buys_computer
D9	youth	Yes
D1	youth	Yes

$$P(age = youth | buys_computer = yes) = 2 / 9 = 0.222$$

RI D	Age	Class: buys_computer
D1	youth	No
D2	youth	No
D8	youth	No

$$P(age = youth | buys_computer = no) = 3 / 5 = 0.600$$



NAÏVE BAYES CLASSIFIER - EXAMPLE

Test Tuple: $X = (age = youth, income = medium, student = yes, credit_rating = fair)$

RID	Age	Income	Student	Credit_Rating	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D10	senior	medium	Yes	fair	Yes
D11	youth	medium	Yes	excellent	Yes
D12	middle aged	medium	No	excellent	Yes
D13	middle aged	high	Yes	fair	Yes
D14	senior	medium	No	excellent	No

The prior probability of each class, can be computed based on the training tuples:

$$P(buys_computer = yes) = 9 / 14 = 0.643$$

$$P(buys_computer = no) = 5 / 14 = 0.357$$

Compute the conditional probabilities of

RID	Income	Class: buys_computer
D4	medium	Yes
D10	medium	Yes
D11	medium	Yes
D12	medium	Yes

$$P(income = medium | buys_computer = yes) = 4 / 9 = 0.444$$

RID	Age	Class: buys_computer
D8	medium	No
D14	medium	No

$$P(income = medium | buys_computer = no) = 2 / 5 = 0.400$$



NAÏVE BAYES CLASSIFIER - EXAMPLE

Test Tuple: $X = (age = youth, income = medium, student = yes, credit_rating = fair)$

RID	Age	Income	Student	Credit_Rating	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D10	senior	medium	Yes	fair	Yes
D11	youth	medium	Yes	excellent	Yes
D12	middle aged	medium	No	excellent	Yes
D13	middle aged	high	Yes	fair	Yes
D14	senior	medium	No	excellent	No

The prior probability of each class, can be computed based on the training tuples:

$$P(buys_computer = yes) = 9 / 14 = 0.643$$

$$P(buys_computer = no) = 5 / 14 = 0.357$$

Compute the conditional probabilities of

RID	Income	Class: buys_computer
D4	medium	Yes
D10	medium	Yes
D11	medium	Yes
D12	medium	Yes

$$P(income = medium | buys_computer = yes) = 4 / 9 = 0.444$$

RID	Age	Class: buys_computer
D8	medium	No
D14	medium	No

$$P(income = medium | buys_computer = no) = 2 / 5 = 0.400$$



NAÏVE BAYES CLASSIFIER - EXAMPLE

Test Tuple: $X = (age = youth, income = medium, student = yes, credit_rating = fair)$

RID	Age	Income	Student	Credit_Rating	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D10	senior	medium	Yes	fair	Yes
D11	youth	medium	Yes	excellent	Yes
D12	middle aged	medium	No	excellent	Yes
D13	middle aged	high	Yes	fair	Yes
D14	senior	medium	No	excellent	No

The prior probability of each class, can be computed based on the training tuples:

$$P(buys_computer = yes) = 9 / 14 = 0.643$$

$$P(buys_computer = no) = 5 / 14 = 0.357$$

Compute the conditional probabilities of

RID	Student	Class: buys_computer
D5	Yes	Yes
D7	Yes	Yes
D9	Yes	Yes
D10	Yes	Yes
D11	Yes	Yes
D13	Yes	Yes

$$P(student = yes | buys_computer = yes) = 6 / 9 = 0.667$$

RID	Age	Class: buys_computer
D6	Yes	No

$$P(student = yes | buys_computer = no) = 1 / 5 = 0.200$$



NAÏVE BAYES CLASSIFIER - EXAMPLE

Test Tuple: $X = (age = youth, income = medium, student = yes, credit_rating = fair)$

RID	Age	Income	Student	Credit_Rating	Class: buys_computer
D1	youth	high	No	fair	No
D2	youth	high	No	excellent	No
D3	middle aged	high	No	fair	Yes
D4	senior	medium	No	fair	Yes
D5	senior	low	Yes	fair	Yes
D6	senior	low	Yes	excellent	No
D7	middle aged	low	Yes	excellent	Yes
D8	youth	medium	No	fair	No
D9	youth	low	Yes	fair	Yes
D10	senior	medium	Yes	fair	Yes
D11	youth	medium	Yes	excellent	Yes
D12	middle aged	medium	No	excellent	Yes
D13	middle aged	high	Yes	fair	Yes
D14	senior	medium	No	excellent	No

The prior probability of each class, can be computed based on the training tuples:

$$P(buys_computer = yes) = 9 / 14 = 0.643$$

$$P(buys_computer = no) = 5 / 14 = 0.357$$

Compute the conditional probabilities of

RID	Credit_Rating	Class: buys_computer
D3	fair	Yes
D4	fair	Yes
D5	fair	Yes
D9	fair	Yes
D10	fair	Yes
D13	fair	Yes

$$P(credit_rating = fair | buys_computer = yes) = 6 / 9 = 0.667$$

RID	Credit_Rating	Class: buys_computer
D6	fair	No
D8	fair	No

$$P(credit_rating = fair | buys_computer = no) = 2 / 5 = 0.400$$



NAÏVE BAYES CLASSIFIER - EXAMPLE

Test Tuple: $X = (age = youth, income = medium, student = yes, credit_rating = fair)$

$$P(age = youth | buys_computer = yes) = 2/9 = 0.222$$

$$P(age = youth | buys_computer = no) = 3/5 = 0.600$$

$$P(student = yes | buys_computer = yes) = 6/9 = 0.667$$

$$P(student = yes | buys_computer = no) = 1/5 = 0.200$$

$$P(income = medium | buys_computer = yes) = 4/9 = 0.444$$

$$P(income = medium | buys_computer = no) = 2/5 = 0.400$$

$$P(credit_rating = fair | buys_computer = yes) = 6/9 = 0.667$$

$$P(credit_rating = fair | buys_computer = no) = 2/5 = 0.400$$

$$\begin{aligned} P(X | buys_computer = yes) &= P(age = youth | buys_computer = yes) \\ &\times P(student = yes | buys_computer = yes) \\ &\times P(income = medium | buys_computer = yes) \\ &\times P(credit_rating = fair | buys_computer = yes) \\ &= 0.222 \times 0.444 \times 0.667 \times 0.667 = \mathbf{0.044} \end{aligned}$$

$$\begin{aligned} P(X | buys_computer = no) &= P(age = youth | buys_computer = no) \\ &\times P(student = yes | buys_computer = no) \times P(income = medium | buys_computer = no) \\ &\times P(credit_rating = fair | buys_computer = no) \\ &= 0.600 \times 0.400 \times 0.200 \times 0.400 = \mathbf{0.019} \end{aligned}$$

$$\begin{aligned} P(X | buys_computer = yes) \cdot P(buys_computer = yes) \\ = \mathbf{0.044 \times 0.643 = 0.028} \end{aligned}$$

$$\begin{aligned} P(X | buys_computer = no) \cdot P(buys_computer = no) \\ = \mathbf{0.019 \times 0.357 = 0.007} \end{aligned}$$



LAZY LEARNERS (OR LEARNING FROM YOUR NEIGHBORS)

- The classification methods discussed so far decision tree induction, Bayesian classification are all examples of *eager learners*. **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.
- Imagine a contrasting lazy approach, in which the learner instead waits until the last minute before doing any model construction to classify a given test tuple. That is, when given a training tuple, a **lazy learner** simply stores it (or does only a little minor processing) and waits until it is given a test tuple.
- Because lazy learners store the training tuples or "**instances**", they are also referred to as **instance-based learners**, even though all learning is essentially based



K-NEAREST-NEIGHBOR CLASSIFIERS

- Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it.
- The training tuples are described by n attributes. Each tuple represents a point in an n -dimensional space. In this way, all the training tuples are stored in an n -dimensional pattern space.
- When given an unknown tuple, a **k -nearest-neighbor classifier** searches the pattern space for the k training tuples that are closest to the unknown tuple. These k training tuples are the k "nearest neighbors" of the unknown tuple. "Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $X_1 = (x_{11}, x_{12}, \dots, x_{1n})$ and $X_2 = (x_{21}, x_{22}, \dots, x_{2n})$, is

$$\text{dist}(X_1, X_2) = \sqrt{\sum_{i=1}^n (x_{1i} - x_{2i})^2}$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple X_1 and in tuple X_2 , square this difference, and accumulate it. The square root is taken of the total accumulated distance count.



K-NEAREST-NEIGHBOR CLASSIFIERS

Typically, we normalize the values of each attribute before using Eq. (9.22). This helps prevent attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). Min-max normalization, for example, can be used to transform a value v of a numeric attribute A to v' in the range $[0, 1]$ by computing

$$v' = \frac{v - \min_A}{\max_A - \min_A}$$

where \min_A and \max_A are the minimum and maximum values of attribute A .

- For k -nearest-neighbor classification, the unknown tuple is assigned the most common class among its k -nearest neighbors. When $k=1$, the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space.
- Nearest-neighbor classifiers can also be used for numeric prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the k -nearest neighbors of the unknown tuple.



K-NEAREST-NEIGHBOR CLASSIFIERS

How can I determine a good value for k , the number of neighbors?

- **Cross-Validation:** Utilize techniques like cross-validation to test different k values and select the one that maximizes the model's performance. This helps ensure that the chosen k generalizes well to unseen data.
- **Elbow Method:** Plot the error rate or accuracy against various k values and identify the point of diminishing returns, often referred to as the "elbow." This can help pinpoint a suitable k value.
- **Domain Knowledge:** Consider the context of your problem and domain expertise. Depending on the specific task and dataset, certain k values may align better with the inherent patterns in the data.



INTRODUCTION TO REGRESSION

Regression in machine learning refers to a supervised learning technique where the goal is to predict a continuous numerical value based on one or more independent features. It finds relationships between variables so that predictions can be made. we have two types of variables present in regression:

- **Dependent Variable (Target):** The variable we are trying to predict e.g **house price**.
- **Independent Variables (Features):** The input variables that influence the prediction e.g **locality, number of rooms**.

Regression analysis problem works with if output variable is a real or continuous value such as "**salary**" or "**weight**". Many different regression models can be used but the simplest model in them is linear regression.



TYPES OF REGRESSION

Regression can be classified into different types based on the number of predictor variables and the nature of the relationship between variables:

1. Simple Linear Regression: Linear regression is one of the simplest and most widely used statistical models. This assumes that there is a linear relationship between the independent and dependent variables. This means that the change in the dependent variable is proportional to the change in the independent variables. For example, predicting the price of a house based on its size.

2. Multiple Linear Regression: Multiple linear regression extends simple linear regression by using multiple independent variables to predict target variable. For example, predicting the price of a house based on multiple features such as size, location, number of rooms, etc.

3. Polynomial Regression: Polynomial regression is used to model with non-linear relationships between the dependent variable and the independent variables. It adds polynomial terms to the linear regression model to capture more complex relationships.

TYPES OF REGRESSION

4. Ridge & Lasso Regression: Ridge & lasso regression are regularized versions of linear regression that help avoid overfitting by penalizing large coefficients. When there's a risk of overfitting due to too many features we use these type of regression algorithms.

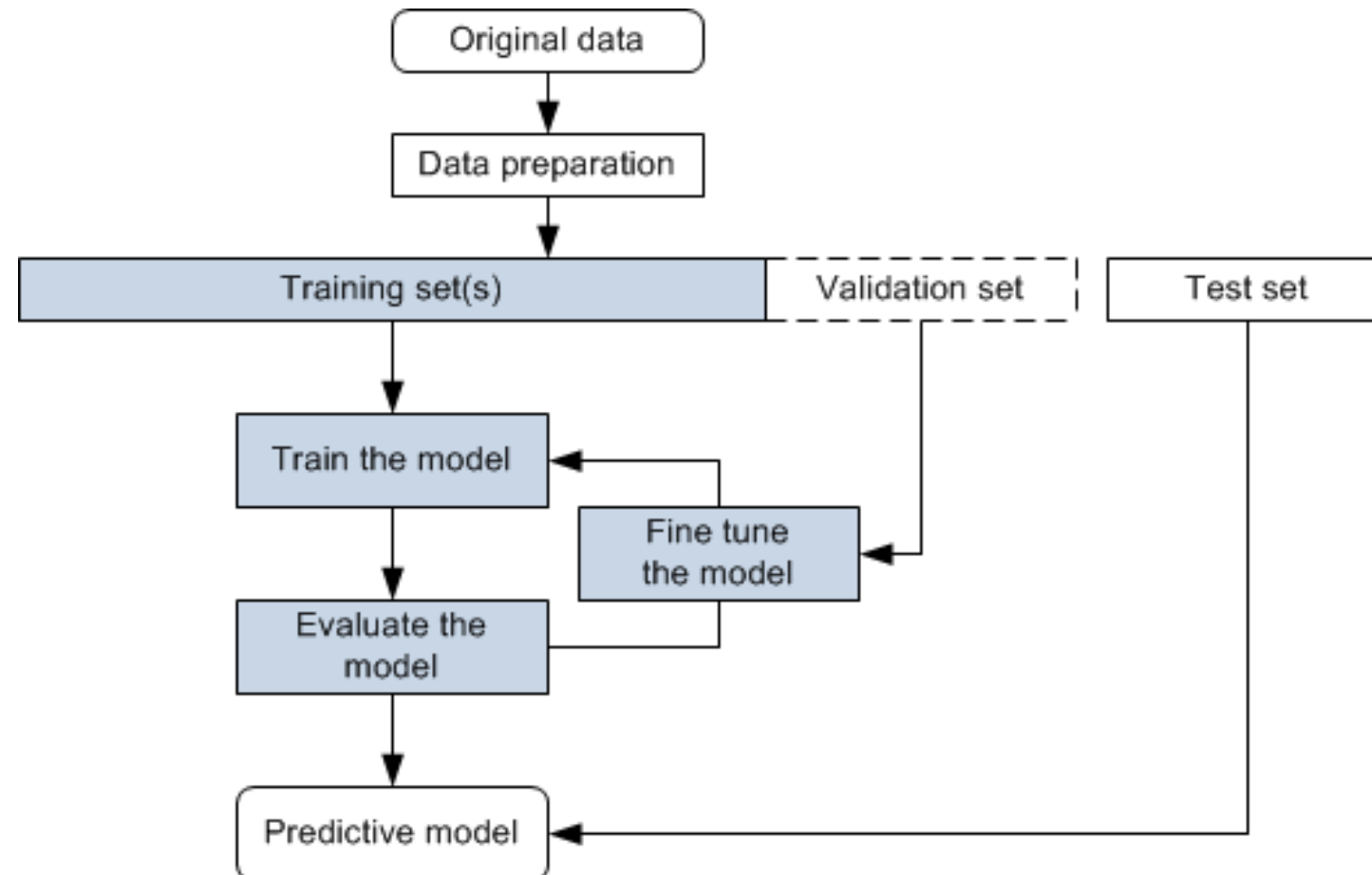
5. Support Vector Regression (SVR): SVR is a type of regression algorithm that is based on the Support Vector Machine (SVM) algorithm. SVM is a type of algorithm that is used for classification tasks but it can also be used for regression tasks. SVR works by finding a hyperplane that minimizes the sum of the squared residuals between the predicted and actual values.

6. Decision Tree Regression: Decision tree Uses a tree-like structure to make decisions where each branch of tree represents a decision and leaves represent outcomes. For example, predicting customer behavior based on features like age, income, etc. there we use decision tree regression.

7. Random Forest Regression: Random Forest is an ensemble method that builds multiple

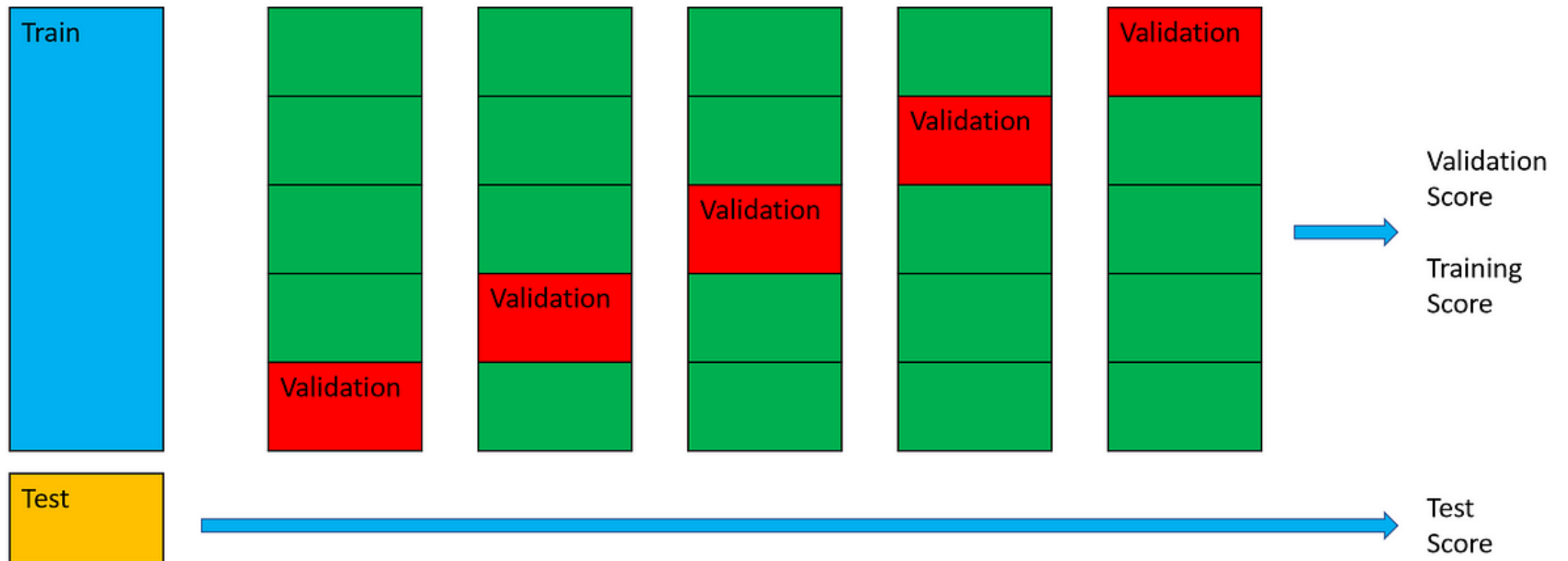


MODEL VALIDATION



MODEL VALIDATION

The process that helps us evaluate the performance of a trained model is called Model Validation. It helps us in validating the machine learning model performance on new or unseen data. It also helps us confirm that the model achieves its intended purpose.



MODEL VALIDATION

Types of Model Validation

Model validation is the step conducted post Model Training, wherein the effectiveness of the trained model is assessed using a testing dataset. This dataset may or may not overlap with the data used for model training.

Model validation can be broadly categorized into two main approaches based on how the data is used for testing:

1. In-Sample Validation: This approach involves the use of data from the same dataset that was employed to develop the model.

- **Holdout method:** The dataset is then divided into training set which is used to train the model and a hold out set which is used to test the performance of the model. This

MODEL VALIDATION

2. Out-of-Sample Validation

This approach relies on entirely different data from the data used for training the model. This gives a more reliable prediction of how accurate the model will be in predicting new inputs.

- **K-Fold Cross-validation:** The data is divided into k number of folds. The model is trained on $k-1$ folds and tested on the fold that is left. This is repeated k times, each time using a different fold for testing. This offers a more extensive analysis than the holdout method.
- **Leave-One-Out Cross-validation (LOOCV):** This is a form of k -fold cross validation where k is equal to the number of instances. Only one piece of data is not used to train the model. This is repeated for each data point. Unfortunately, LOOCV is also time consuming when dealing with large datasets.
- **Stratified K-Fold Cross-validation:** k -fold cross-validation: in this type of cross-validation each fold has the same ratio of classes/categories as the overall dataset.



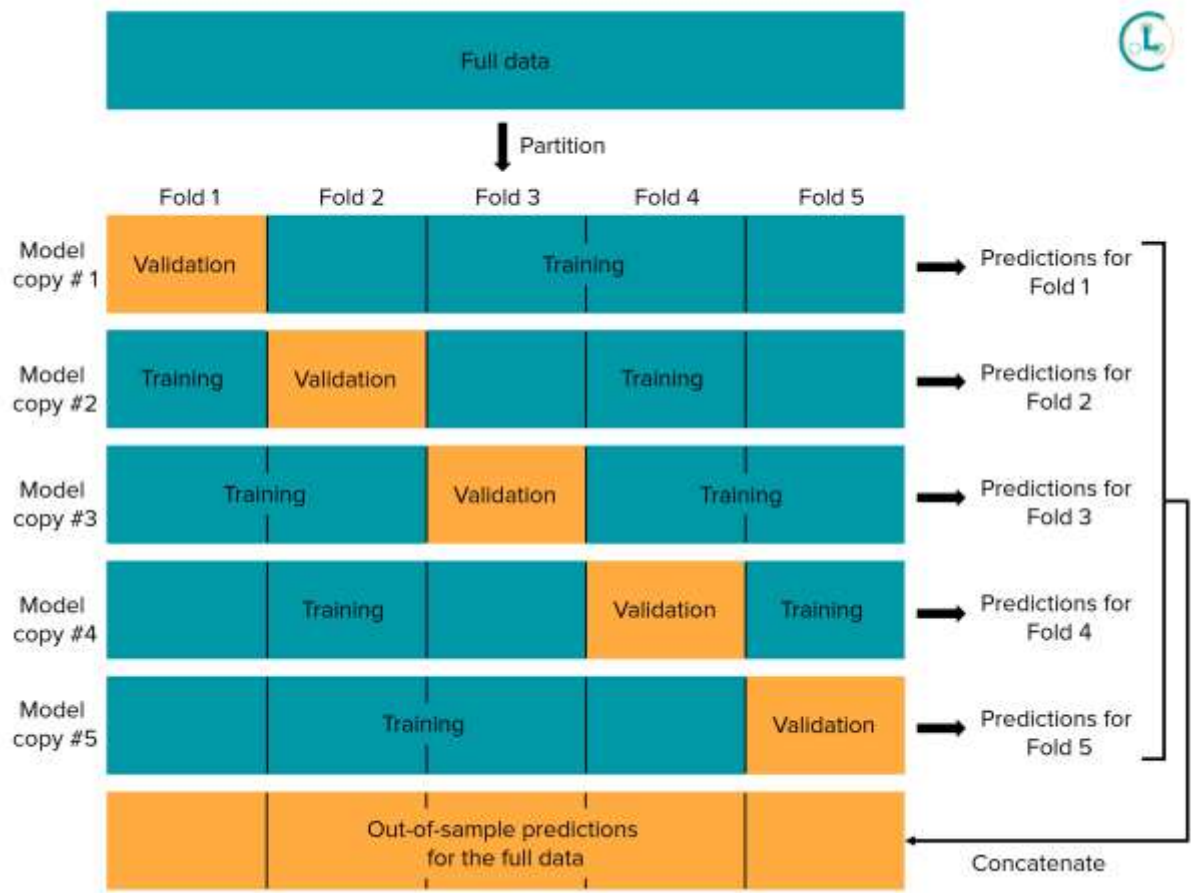
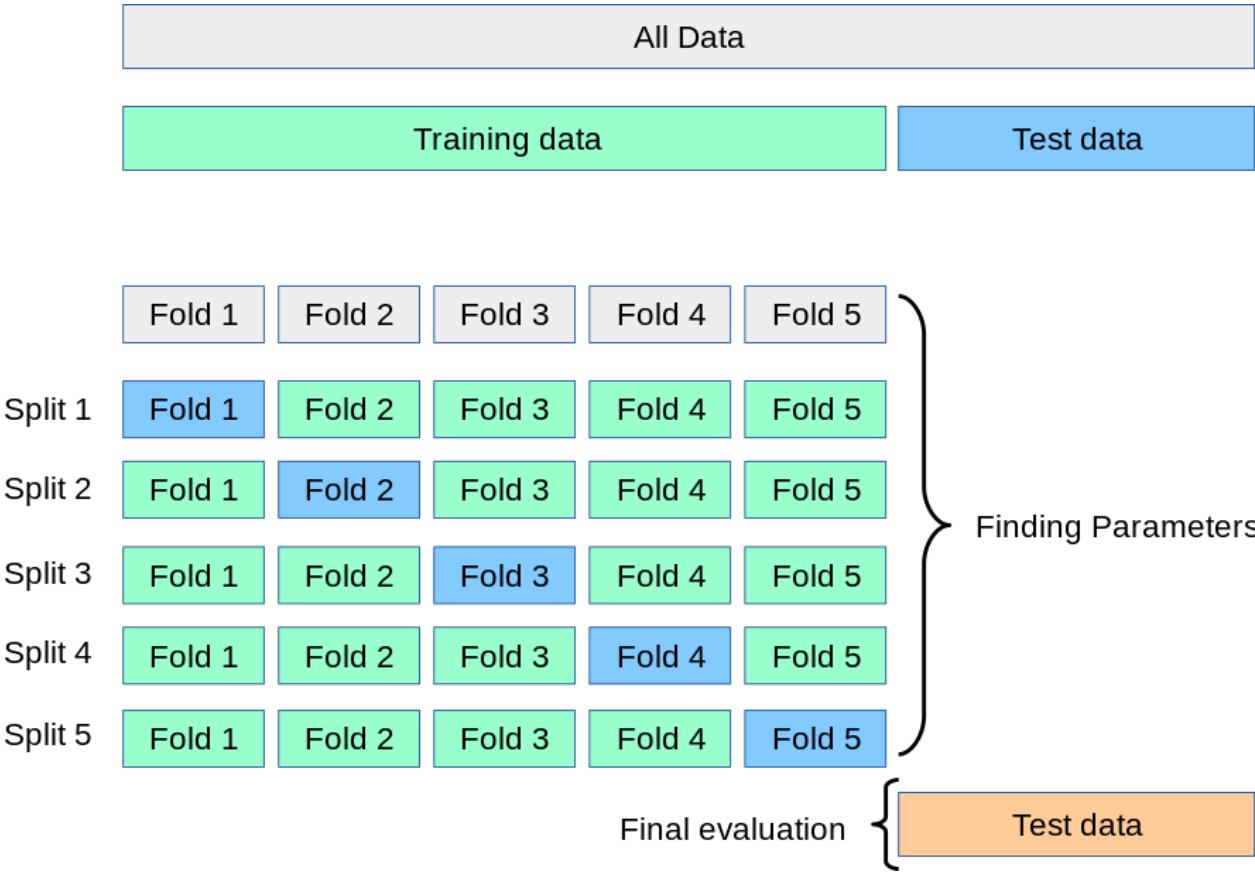
K-FOLD CROSS-VALIDATION

In ***k*-fold cross-validation**, the initial data are randomly partitioned into k mutually exclusive subsets or "folds," D_1, D_2, \dots, D_k , each of approximately equal size. Training and testing is performed k times.

- In iteration i , partition D_i is reserved as the test set, and the remaining partitions are collectively used to train the model.
- That is, in the first iteration, subsets D_2, \dots, D_k collectively serve as the training set to obtain a first model, which is tested on D_1 ; the second iteration is trained on subsets D_1, D_2, \dots, D_k and tested on D_2 ; and so on.
- Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing.



K-FOLD CROSS-VALIDATION



LEAVE ONE OUT CROSS-VALIDATION

LOOCV (Leave-One-Out Cross-Validation) is a model evaluation technique used to assess the performance of a machine learning model on small datasets. In LOOCV, one observation is used as the test set while the rest form the training set. This process is repeated for each data point in the dataset, resulting in n training-testing cycles, where n is the number of observations. The overall accuracy is averaged across all iterations.

- **Mathematical Expression**

In Leave-One-Out Cross-Validation (LOOCV), each individual observation serves once as the validation set, while the remaining $n-1$ observations are used for training. Instead of refitting the model n times, LOOCV for linear models can be computed efficiently using the following formula:

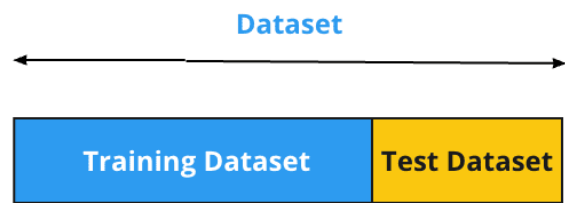
$$LOO_{Error} = \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{1 - h_{ii}} \right)^2$$

Where,

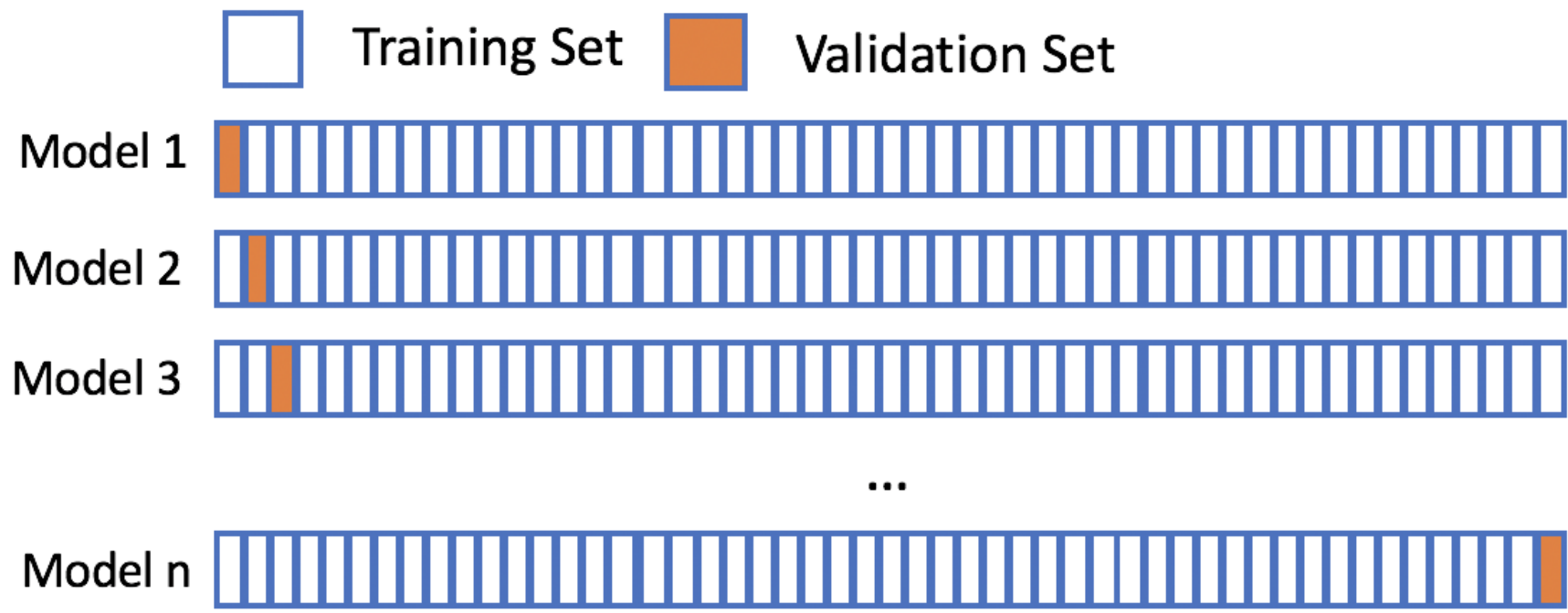
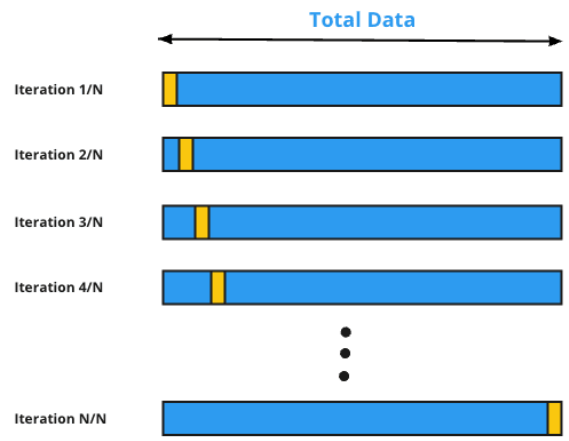
y_i = Actual value of the i^{th} observation



LEAVE ONE OUT CROSS-VALIDATION



LOOCV: Leave One Out Cross Validation



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Two types of labels are there.

1. **Positive tuples** (tuples of the main class of interest)
2. **Negative tuples** (all other tuples).

Given two classes, for example, the **positive tuples** may be *buys computer = yes* while the **negative tuples** are *buys computer = no*. Suppose we use our **classifier** on a test set of labeled tuples. **P** is the **number of positive tuples** and **N** is the **number of negative tuples**. For each tuple, we **compare** the **classifier's class label prediction** with the **tuple's known class label**.

There are four additional terms we need to know that are the "building blocks" used in computing many evaluation measures.

- **True positives (TP)**: These refer to the positive tuples that were correctly labeled by the classifier. Let *TP* be the number of true positives.
- **True negatives (TN)**: These are the negative tuples that were correctly labeled by the classifier. Let *TN* be the number of true negatives.
- **False positives (FP)**: These are the negative tuples that were incorrectly labeled as positive (e.g., tuples of class *buys computer = no* for which the classifier predicted *buys computer = yes*). Let *FP* be the number of false positives.
- **False negatives (FN)**: These are the positive tuples that were mislabeled as negative (e.g., tuples of class *buys computer = yes* for which the classifier predicted *buys computer = no*). Let *FN* be the number of false negatives.



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Measures	Formula
Accuracy, Recognition Rate	$\frac{TP + TN}{P + N}$
Error Rate, Misclassification Rate	$\frac{FP + FN}{P + N}$
Sensitivity, True Positive Rate, Recall	$\frac{TP}{P}$
Specificity, True Negative Rate	$\frac{TN}{N}$
Precision	$\frac{TP}{TP + FP}$
F, F_1 , F-Score, Harmonic mean of Precision and Recall	$\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$
F_β , Where β Is a non-negative real number	$\frac{(1 + \beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}}$

Actual Class

Predicted Class			Total
	Yes	No	
Yes	TP	FN	P
No	FP	TN	N
Total	P'	N'	P + N



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Sample	Actual Class	Predicted Class
1	Positive	Positive
2	Positive	Negative
3	Positive	Positive
4	Positive	Positive
5	Positive	Positive
6	Positive	Negative
7	Positive	Positive
8	Negative	Negative
9	Negative	Positive
10	Negative	Positive
11	Negative	Positive
12	Negative	Negative
13	Negative	Negative
14	Negative	Negative
15	Negative	Negative
16	Positive	Positive
17	Positive	Positive
18	Positive	Positive
19	Negative	Negative
20	Negative	Positive

Actual Class	Predicted Class		Total
	Yes	No	
	TP	FN	
	FP	TN	
	P'	N'	



METRICS FOR EVALUATING CLASSIFIER PERFORMANCE

Measures	Formula
Accuracy, Recognition Rate	$\frac{TP + TN}{P + N}$
Error Rate, Misclassification Rate	$\frac{FP + FN}{P + N}$
Sensitivity, True Positive Rate, Recall	$\frac{TP}{P}$
Specificity, True Negative Rate	$\frac{TN}{N}$
Precision	$\frac{TP}{TP + FP}$
$F, F_1, F\text{-Score},$ Harmonic mean of Precision and Recall	$\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$
F_β , Where β Is a non-negative real number	$\frac{(1 + \beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}}$

Classes	buys computer = yes	buys computer = no	Total
buys computer = yes	6954	46	7000
buys computer = no	412	2588	3000
Total	7366	2634	10000

Classes	Cancer = yes	Cancer = no	Total
Cancer = yes	90	210	300
Cancer = no	140	9560	9700
Total	230	9770	10000



METRICS FOR EVALUATING REGRESSOR PERFORMANCE

Mean Absolute Error (MAE): The average absolute difference between the predicted and actual values of the target variable.

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Mean Squared Error (MSE): The average squared difference between the predicted and actual values of the target variable.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Root Mean Squared Error (RMSE): Square root of the mean squared error.

$$RMSE = \sqrt{\sum_{i=1}^n \frac{(y_i - \hat{y}_i)^2}{n}}$$

y_i = Actual Values

\hat{y}_i = Predicted Values

n = Number of non missing data points



METRICS FOR EVALUATING REGRESSOR PERFORMANCE

R^2 measures how well your regression model explains the variability of the target (dependent) variable. How much of the variation in the actual data can my model explain?

$$R^2 = 1 - \frac{SS_{res}}{SS_{total}}$$

$SS_{res} \rightarrow \sum_{i=1}^n (y_i - \hat{y}_i)^2 \rightarrow$ Residual Sum of Squares

$SS_{total} \rightarrow \sum_{i=1}^n (y_i - \bar{y})^2 \rightarrow$ Total Sum of Squares

R ² Value	Interpretation
1.0	Perfect model – predictions exactly match actual values
0.9	90% of the variance in the data is explained by the model
0.5	Model explains half of the variation – moderate fit
0.0	Model does no better than predicting the mean
<0.0	Model is worse than just predicting the mean value!

- R^2 doesn't tell you if the model is biased. It only tells how much of the variance is captured, not whether predictions are systematically high or low.
- R^2 can be misleading for non-linear models. It's best for linear regression or when the relationship between variables is roughly linear.

METRICS FOR EVALUATING REGRESSOR PERFORMANCE

Sample	Actual Value	Predicted Value
1	10.2	9.8
2	12.5	13.1
3	14.0	13.5
4	9.7	10.5
5	15.5	14.9
6	11.0	11.7
7	8.5	8.9
8	16.2	17.0
9	13.4	12.8
10	10.8	11.1
11	18.0	17.4
12	7.6	8.1
13	14.8	15.2
14	9.9	9.3
15	17.1	16.8
16	12.2	11.9
17	10.1	9.5
18	13.0	13.9
19	15.9	16.3
20	8.8	9.0

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$RMSE = \sqrt{\sum_{i=1}^n \frac{(y_i - \hat{y}_i)^2}{n}}$$

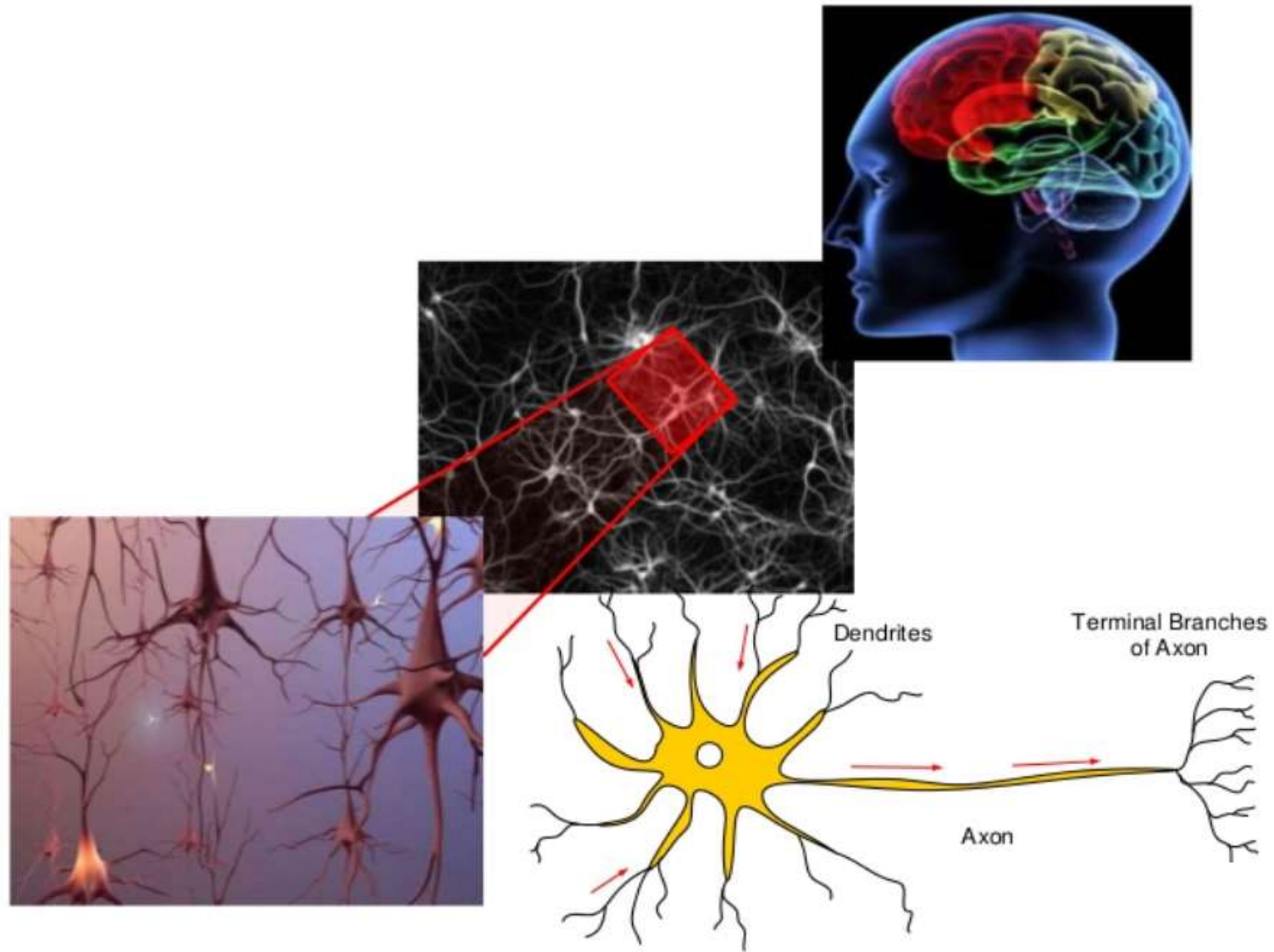
$$R^2 = 1 - \frac{SS_{res}}{SS_{total}}$$

Artificial Intelligence and Machine Learning

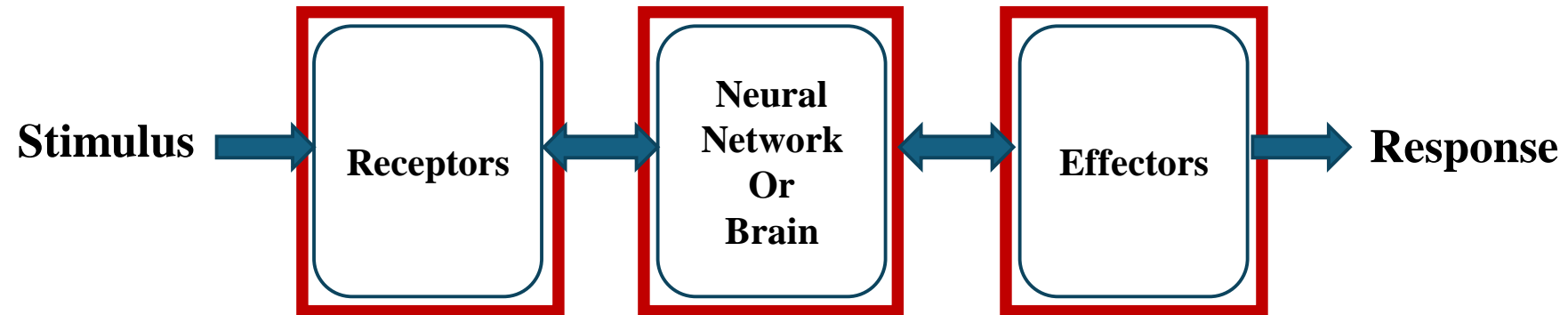
MODULE V

NEURAL NETWORK

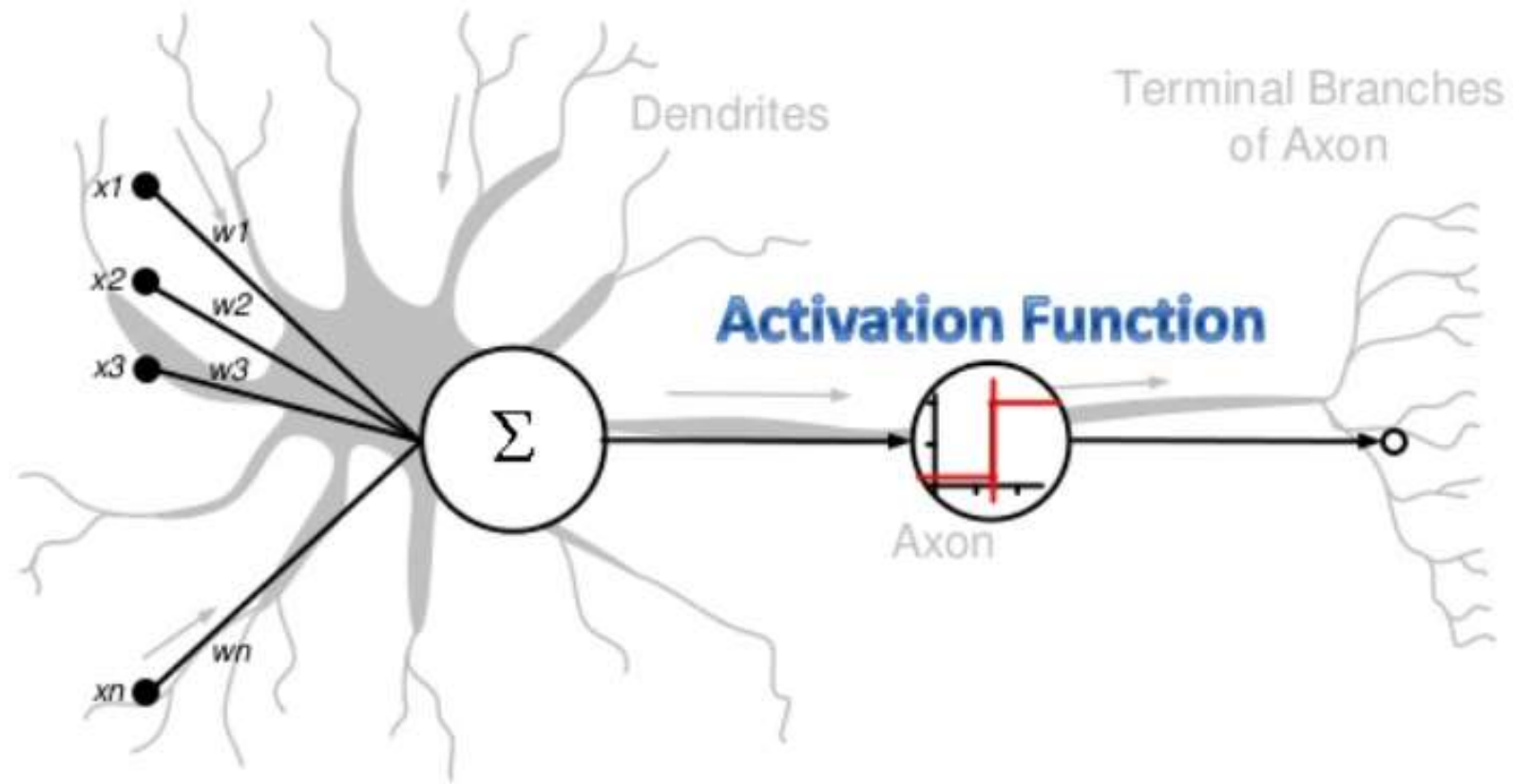
TYPICAL HUMAN BRAIN



BLOCK DIAGRAM OF BIOLOGICAL NERVOUS SYSTEM



HUMAN BRAIN NEURON VS ARTIFICIAL NEURON



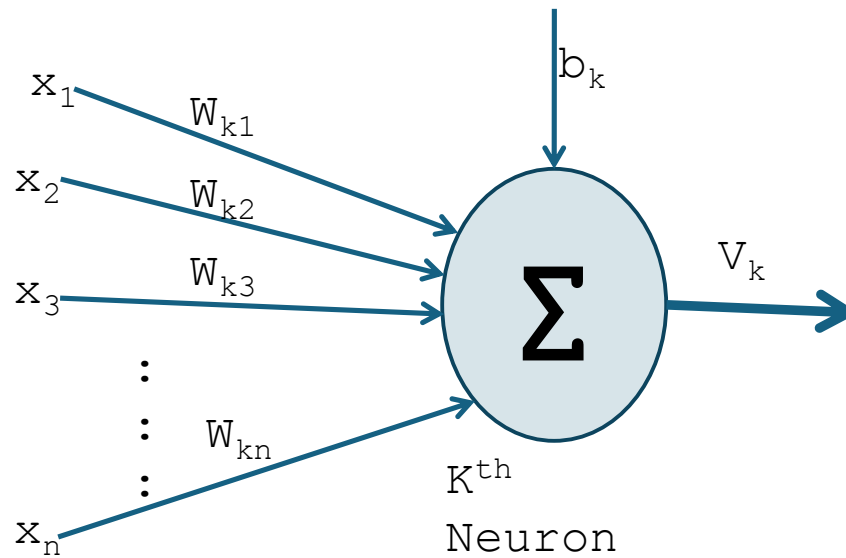
ARTIFICIAL NEURON

- An **artificial neuron** is a connection point in an **artificial neural network**. Artificial neural networks (ANNs), like the human body's **biological neural network**, have a layered architecture and each network node, or connection point, can process input and forward output to other nodes in the network.
- In both **artificial and biological architectures**, the nodes are called **neurons**, and the connections are characterized by synaptic weights, which represent the significance of the connection. As new data is received and processed, the synaptic weights change, and this is how learning occurs.
- **Artificial neurons** are modeled after the hierarchical arrangement of **neurons in biological sensory systems**. In the visual system, for example, light input passes through neurons in successive layers of the retina before being passed to neurons in the thalamus of the brain and then on to neurons in the brain's visual cortex.
- As the neurons pass signals through an increasing number of layers, the brain



ARTIFICIAL NEURON

W = Weights
 b = Bias



$$V_k = W_{k1} * x_1 + W_{k2} * x_2 + W_{k3} * x_3 + \dots + W_{kn} * x_n + b_k$$



ARTIFICIAL NEURON

Weights: Weights are numerical values assigned to the connections between neurons.

They find how much influence each input has on the network's final output.

- During forward propagation, inputs are multiplied by their respective weights before being passed through an activation function. This helps decide **how strongly an input will affect the output.**
- During training, weights are **updated iteratively** through **optimization algorithms** like **gradient descent** to minimize the difference between predicted and actual outcomes.
- **Well-tuned weights** help the network not only make accurate predictions on training data but also generalize to new, unseen data.
- The weights in the network are **initialized to small random numbers** (e.g., ranging from **-1.0 to 1.0** or **0.5 to 0.5**)



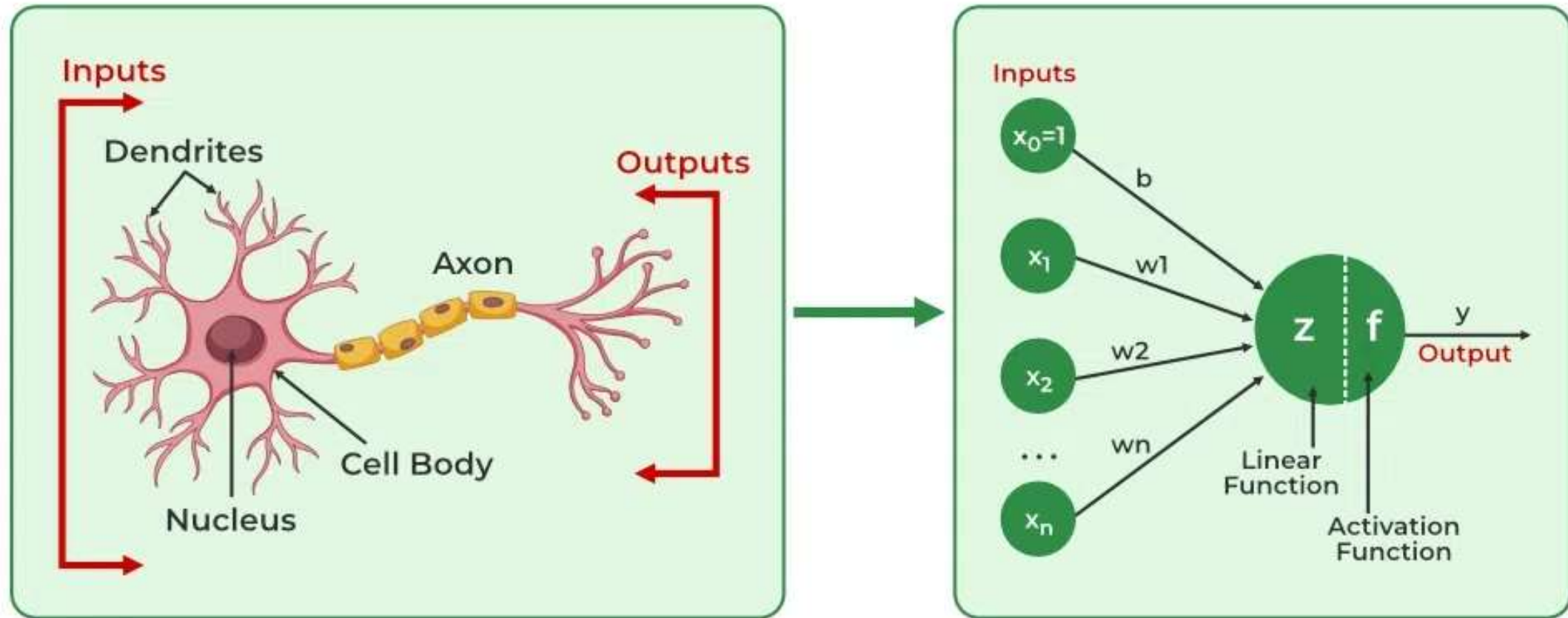
ARTIFICIAL NEURON

Biases are additional parameters that adjust the output of a neuron. Unlike weights, they are not tied to any specific input but instead shift the activation function to better fit the data.

- Biases help neurons **activate even when the weighted sum of inputs** is not enough. This allows the network to recognize patterns that don't necessarily pass through the origin.
- **Without biases**, neurons would **only activate** when the **input reaches a specific threshold**. It makes the network more flexible by enabling activation across a wider range of conditions.
- During training, **biases are updated alongside weights** through backpropagation.

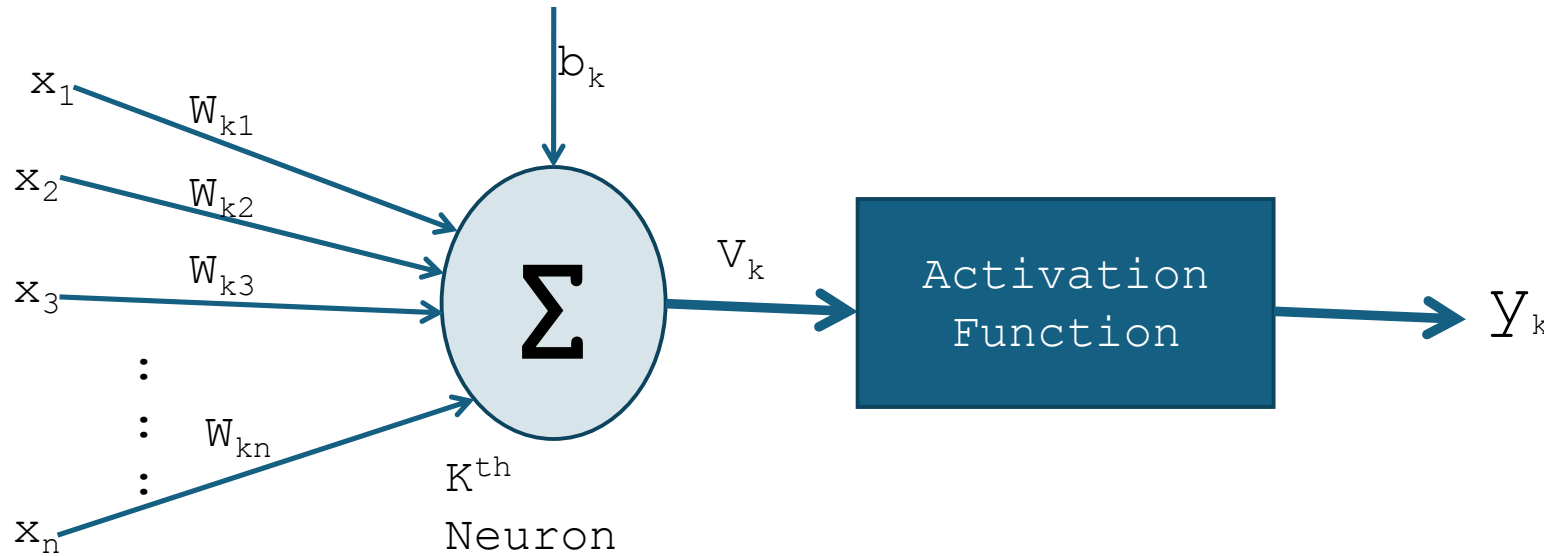


ARTIFICIAL NEURON



ARTIFICIAL NEURON

$f(V_k)$ = Activation Function

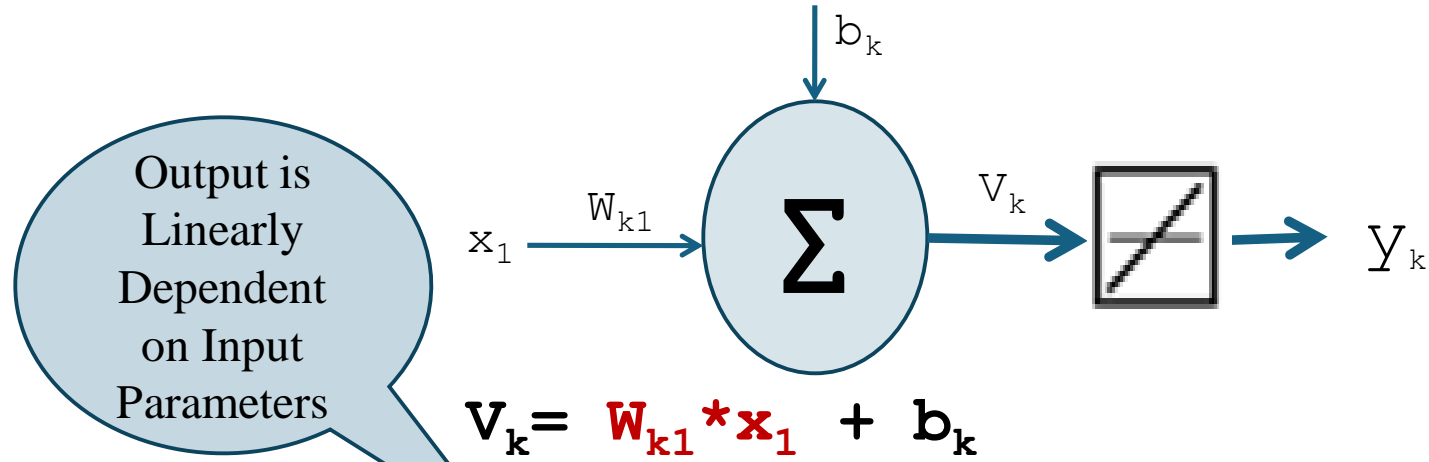


$$V_k = \sum_{j=1}^n (W_{kj} * x_j) + b_k$$

$$y_k = f(V_k)$$

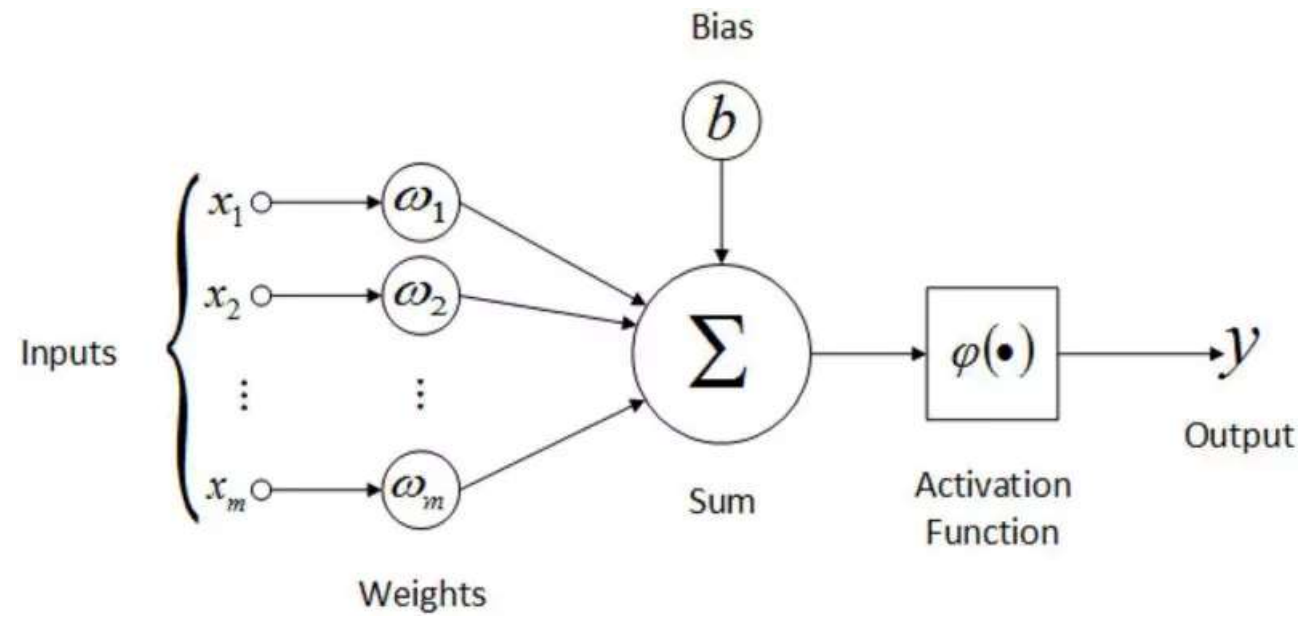


SINGLE NEURON MODEL



Output is Linearly Dependent on Input Parameters

$$V_k = W_{k1} * x_1 + b_k$$
$$Y_k = f(V_k) = W_{k1} * x_1 + b_k$$



SINGLE NEURON MODEL

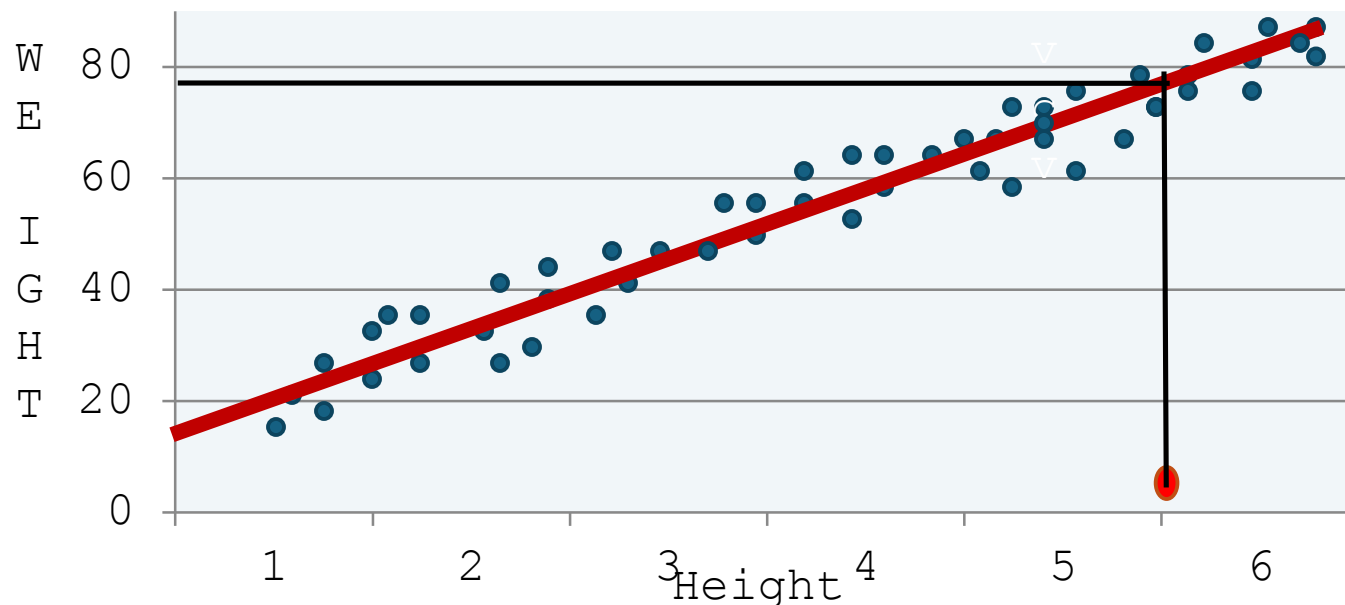
• Application

- For data Fitting applications where we have to fit a straight line to a large

$$y=mx+c$$

Where m=Slope Of Straight Line

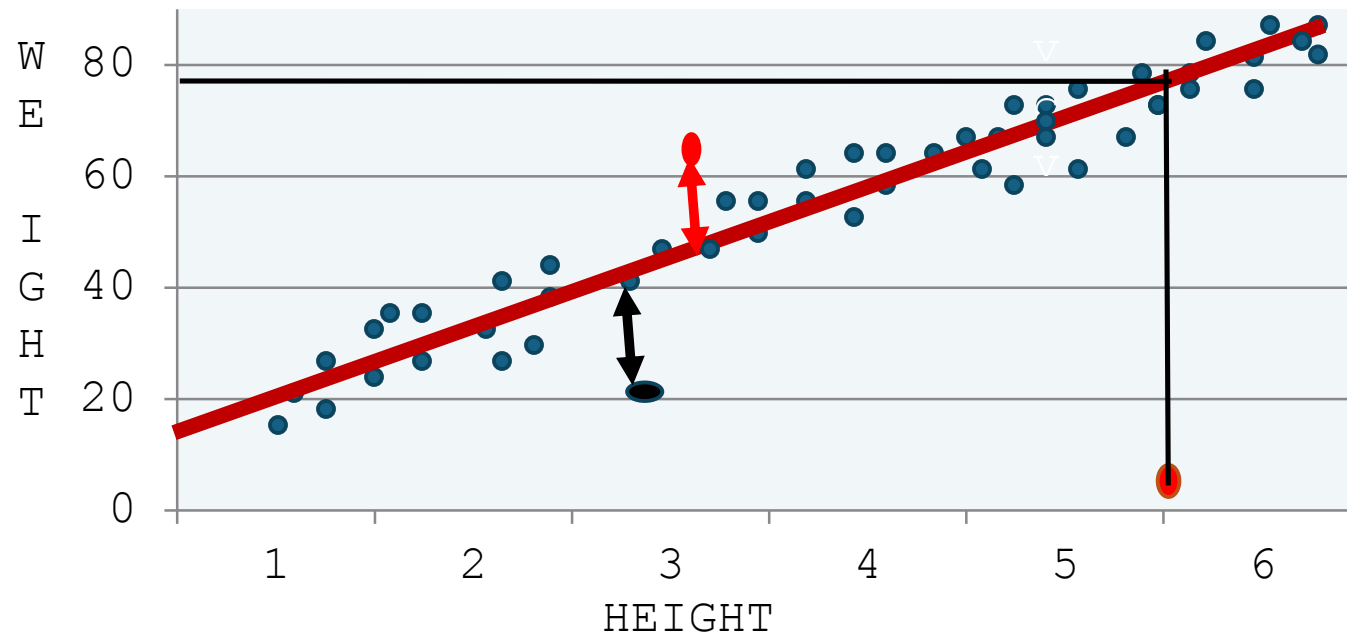
X=Height c=Intercept y=Weight



SINGLE NEURON MODEL

Error Calculation

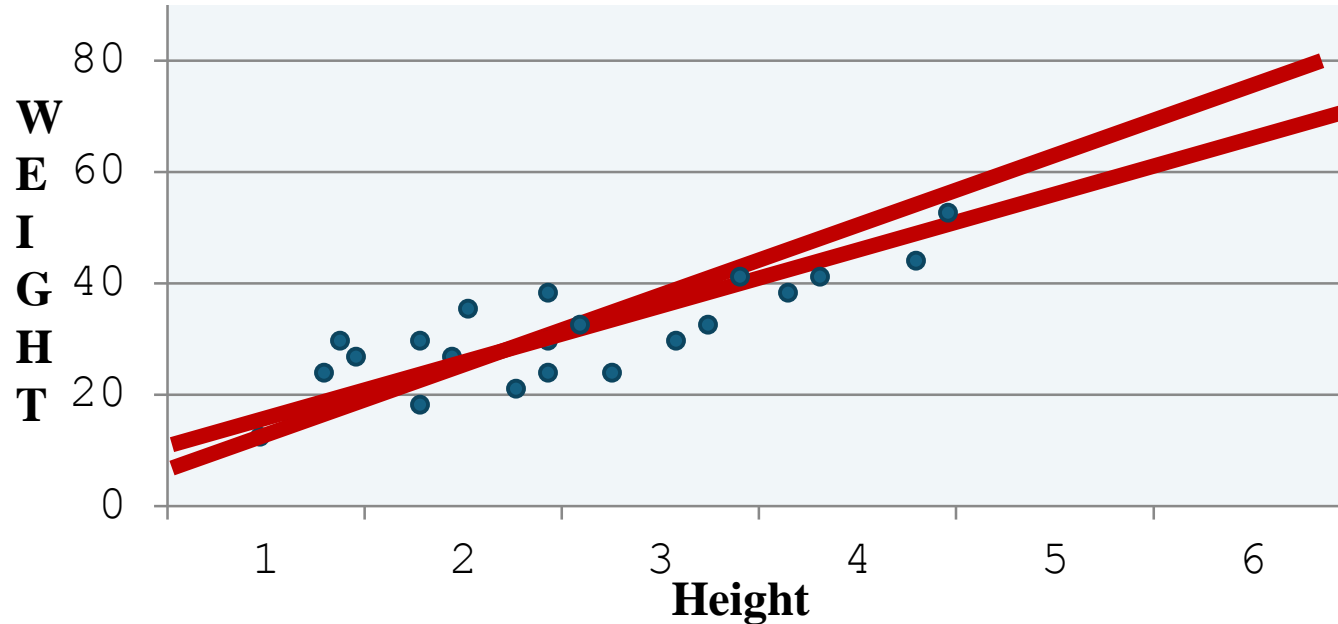
- The error $E_i = (\text{Actual Value} - \text{Predicted value}) = (T_i - y_i)$
- For making +ve $E_i = (T_i - y_i)^2$ [Error for i^{th} input instance]



SINGLE NEURON MODEL

• Error Calculation

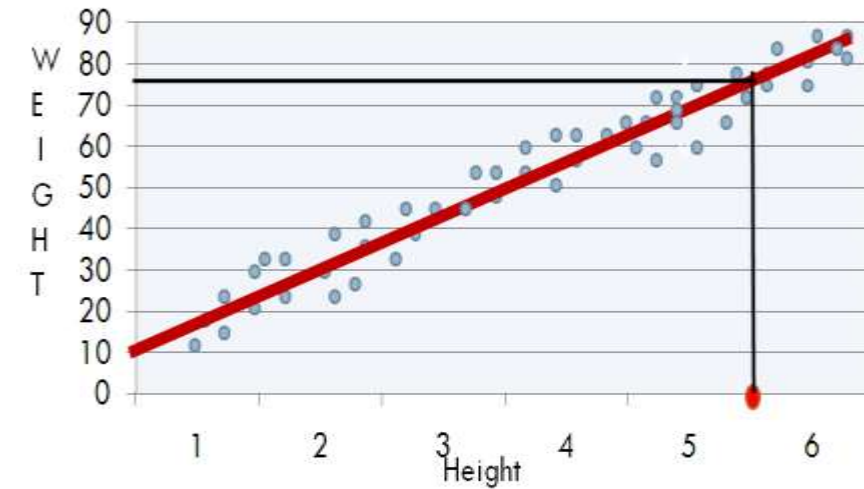
- It is done to adjust the slope (m) and intercept for better fitting next time.



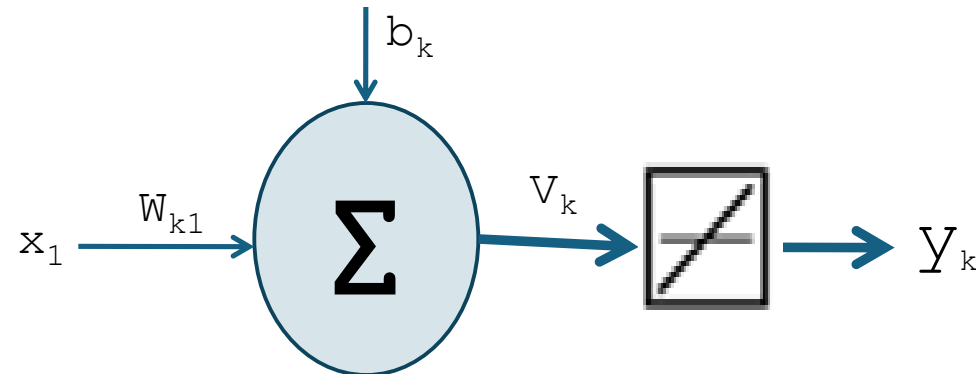
SINGLE NEURON MODEL

$$y_k = W_{k1} * x_1 + b_k$$

$$y = m * x + c$$



Output is
Linearly
Dependent on
Input
Parameters



$$V_k = W_{k1} * x_1 + b_k$$

$$y_k = f(V_k) = W_{k1} * x_1 + b_k$$



SINGLE NEURON MODEL

The **Activation function** symbolizes the activation of the neuron represented by the unit. They transform the input signal of a node in a neural network into an output signal that is then passed on to the next layer. Without activation functions, neural networks would be restricted to modeling only linear relationships between inputs and outputs.

- Activation functions introduce non-linearities, allowing neural networks to learn highly complex mappings between inputs and outputs.
- This function is also referred to as a *squashing function*, because it maps a large input domain onto the smaller range of 0 to 1.

Types of Activation Function

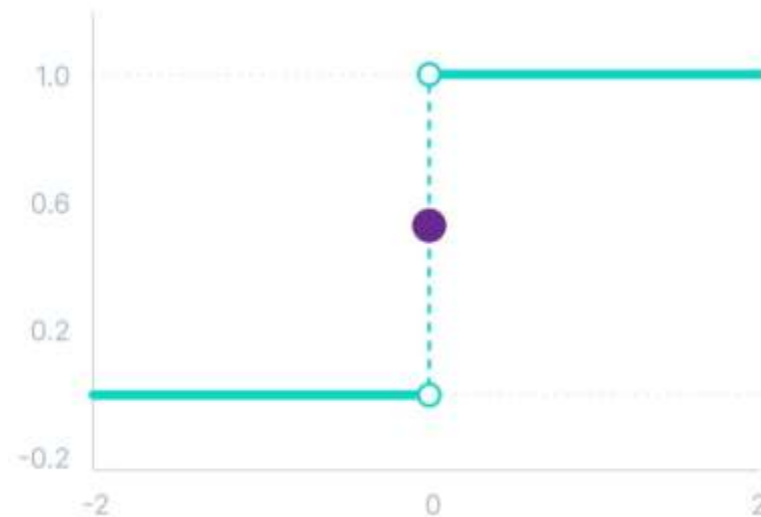
- Binary Step Function
- Linear Activation Function
- Non-Linear Activation Function



SINGLE NEURON MODEL

1. Binary Step Function Binary step function depends on a threshold value that decides whether a neuron should be activated or not. The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.

Binary Step Function



Mathematically it can be represented as:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Here are some of the limitations of binary step function:

- It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.

• The gradient of the step function is zero, which causes a hindrance in training.



SINGLE NEURON MODEL

2. Linear Activation Function

As you can see the function is a line or linear. Therefore, the output of the functions will not be confined between

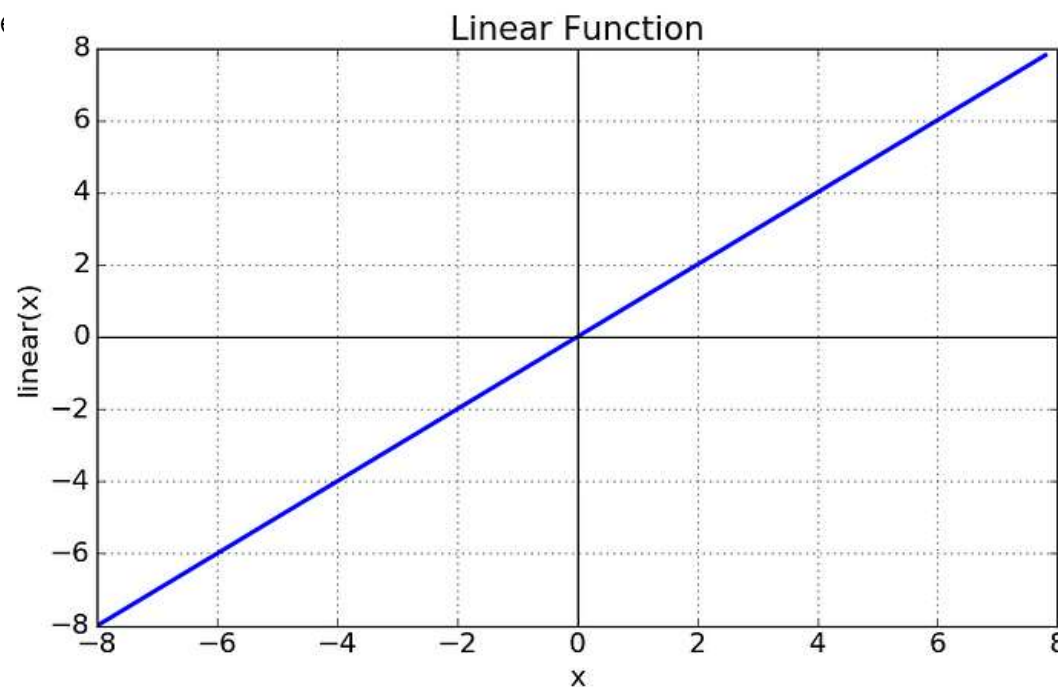


Fig: Linear Activation Function

Equation: $f(x) = x$

Range: (-infinity to infinity)

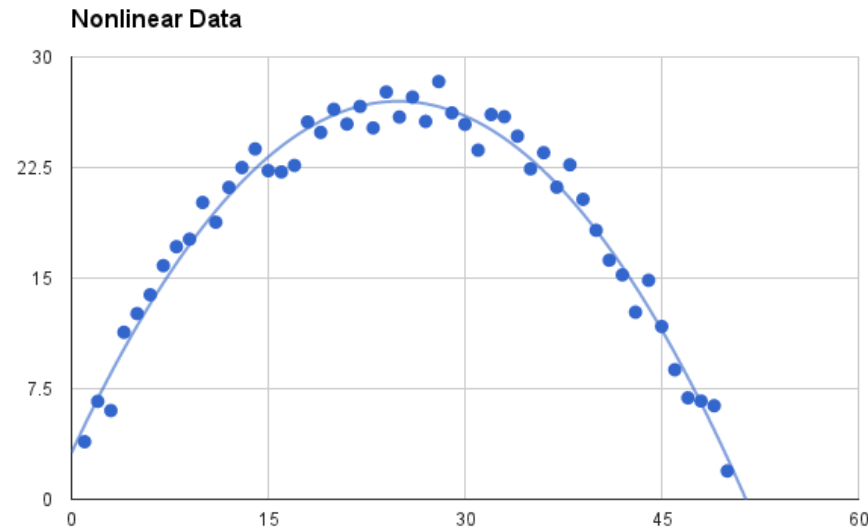
It doesn't help with the complexity or various parameters of usual data that is fed to the neural networks.



SINGLE NEURON MODEL

3. Non-linear Activation Function

The Nonlinear Activation Functions are the most used activation functions. Nonlinearity helps to makes the graph look something like this



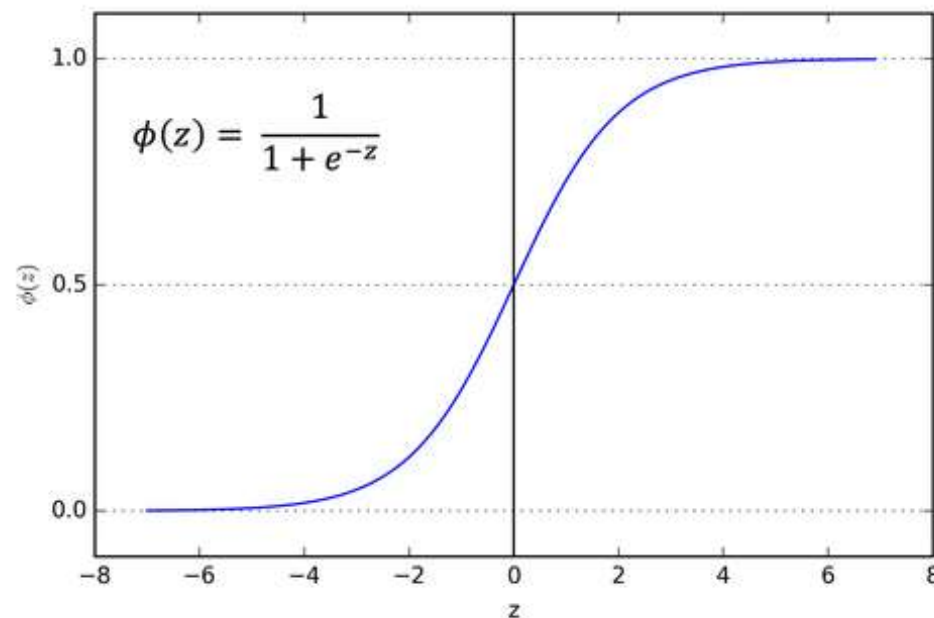
It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output. The main terminologies needed to understand for nonlinear functions are:

- **Derivative or Differential:** Change in y -axis w.r.t. change in x -axis. It is also known as slope.
- **Monotonic function:** A function which is either entirely non-increasing or non-decreasing.



SINGLE NEURON MODEL

a. Sigmoid or Logistic Activation Function: The Sigmoid Function curve looks like a S-shape.



$$\phi(z) = \frac{1}{1 + e^{-z}}$$

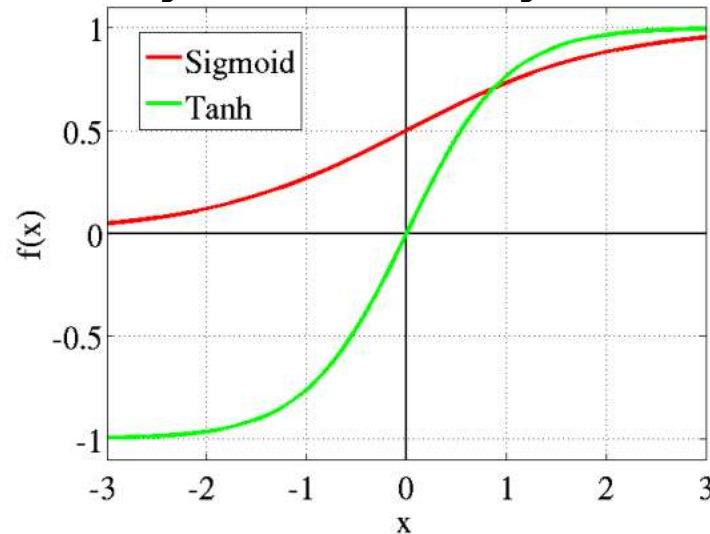
The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to **predict the probability** as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

- The function is **differentiable**. That means, we can find the slope of the sigmoid curve at any two points.
- The function is **monotonic** but function's derivative is not.
- The logistic sigmoid function can cause a neural network to get stuck.



SINGLE NEURON MODEL

b. Tanh or hyperbolic tangent Activation Function: tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped). It is a shifted version of the sigmoid, allowing it to stretch across the y-axis.



$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

Alternatively, it can be expressed using the sigmoid function:

$$\tanh(x) = 2 \times \text{sigmoid}(2x) - 1$$

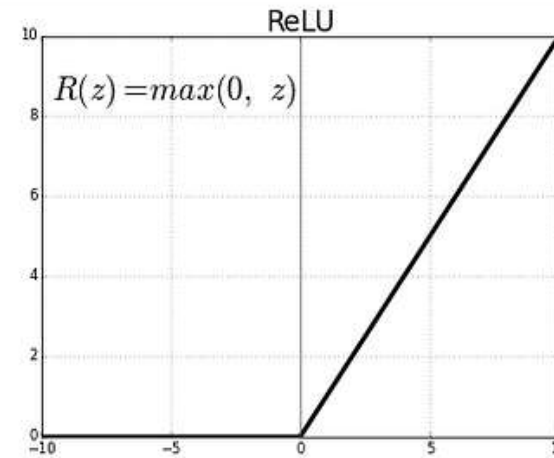
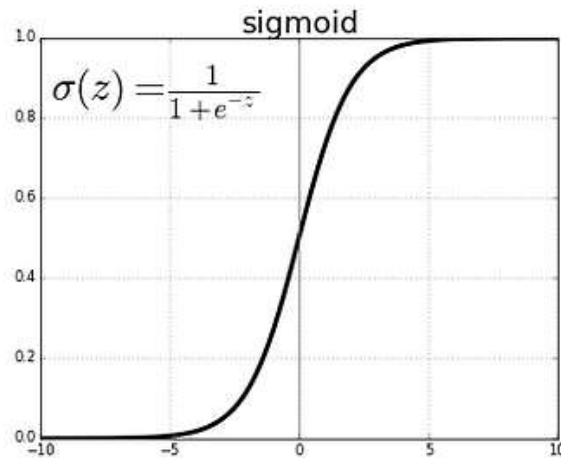
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.
- Range: Outputs values from -1 to +1.
- Non-linear: Enables modeling of complex data patterns.
- Use in Hidden Layers: Commonly used in hidden layers due to its zero-centered output, facilitating easier learning for subsequent layers.
- The function is **differentiable**. The function is **monotonic** while its **derivative is not monotonic**. The tanh function is mainly used classification between two classes.
- Both tanh and logistic sigmoid activation functions are used in feed-forward nets.



SINGLE NEURON MODEL

c. ReLU (Rectified Linear Unit) Activation Function

The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning.



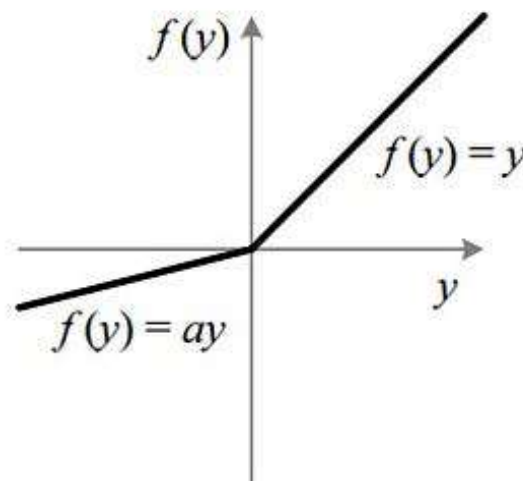
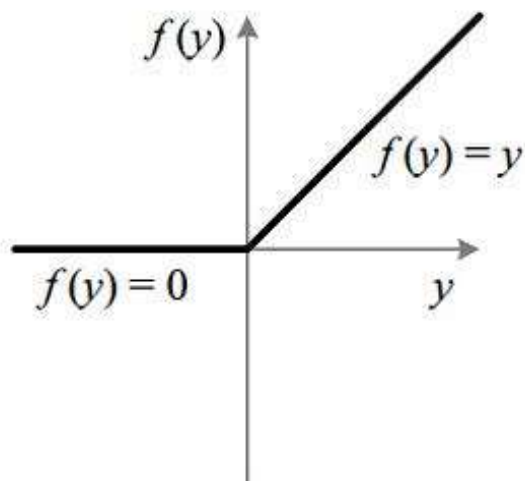
$$R(z) = \max(0, z)$$

- The ReLU is half rectified (from bottom). $f(z)$ is zero when z is less than *zero* and $f(z)$ is equal to z when z is above or equal to *zero*.
- **Range:** $[0 \text{ to } \infty]$ The function and its derivative **both are monotonic**.
- But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

SINGLE NEURON MODEL

d. Leaky ReLU

It is an attempt to solve the dying ReLU problem



$$f(x) = \begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases}$$

- The leak helps to increase the range of the ReLU function. Usually, the value of **a** is 0.01 or so.
- When **a** is not 0.01 then it is called **Randomized ReLU**. Therefore, the **range** of the Leaky ReLU is $(-\infty$ to $\infty)$. Both Leaky and Randomized ReLU functions are monotonic in nature. Also, their derivatives also monotonic in nature.



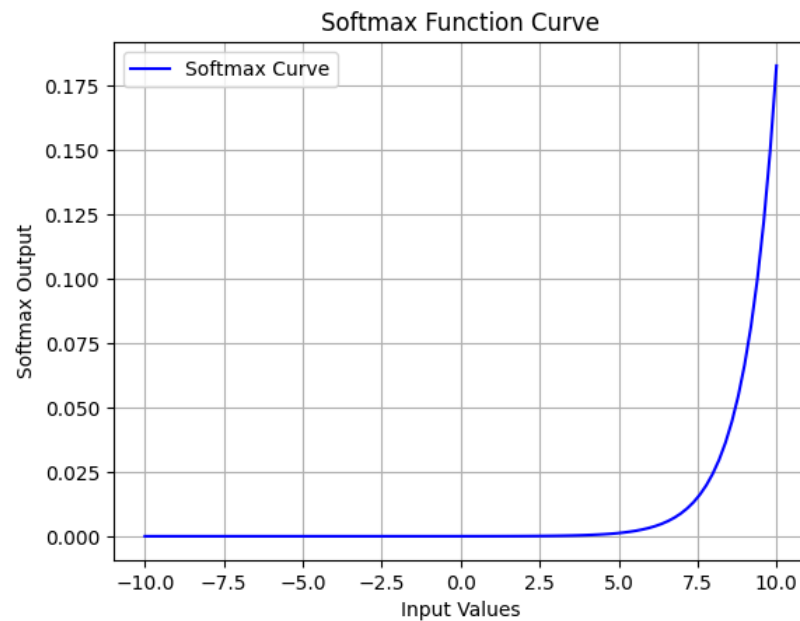
SINGLE NEURON MODEL

4. Exponential Linear Units

a. Softmax Function

Softmax function is designed to handle multi-class classification problems. It transforms raw output scores from a neural network into probabilities. It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.

- Softmax is a non-linear activation function.
- The Softmax function ensures that each class is assigned a probability, helping to identify which



$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

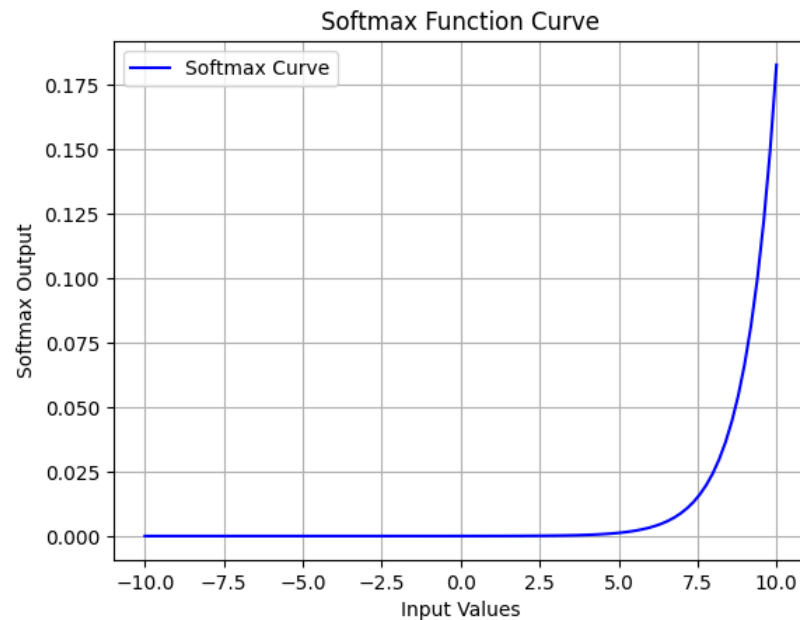


SINGLE NEURON MODEL

e. Softmax Function

Softmax function is designed to handle multi-class classification problems. It transforms raw output scores from a neural network into probabilities. It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.

- Softmax is a non-linear activation function.
- The Softmax function ensures that each class is assigned a probability, helping to identify whi



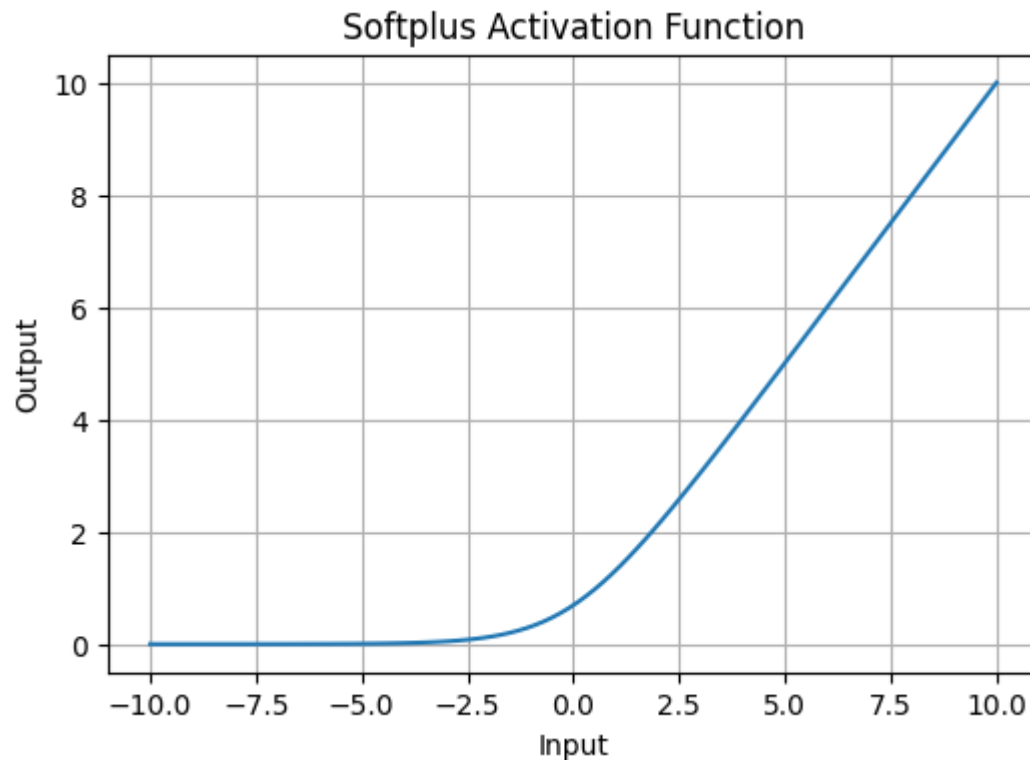
$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$



SINGLE NEURON MODEL

f. SoftPlus Function: SoftPlus function ensures that the output is always positive and differentiable at all points which is an advantage over the traditional ReLU function.

- **Nature:** The SoftPlus function is non-linear.
- **Range:** The function outputs values in the range $(0, \infty)$, similar to ReLU, but without the hard zero threshold that ReLU has.
- **Smoothness:** SoftPlus is a smooth, continuous function, meaning it avoids the sharp discontinuities of ReLU which can sometimes lead to problems during optimization.









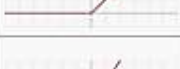


$$A(z) = \log(1 + e^z)$$



SINGLE NEURON MODEL

Why derivative/differentiation is used?

When updating the curve, to know in which direction and how much to change or update the curve depending upon the slope. That is why we use differentiation in almost every part of Machine Learning and Deep Learning.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) ^[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) ^[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$



NEURAL NETWORK ARCHITECTURES

A **Network Architecture/Topology** is the arrangement of a network along with its nodes and connecting lines. According to the topology, Neural Network can be classified as the following kinds:

1. Feed forward Network: It is a non-recurrent network having processing units/nodes in layers and all the nodes in a layer are connected with the nodes of the previous layers. The connection has different weights upon them. There is no feedback loop means the signal can only flow in one direction, from input to output. It may be divided into the following two types:

a) Single layer feed forward network: The concept is of feed forward ANN having only one weighted layer. In other words, we can say the input layer is fully connected to the output layer.

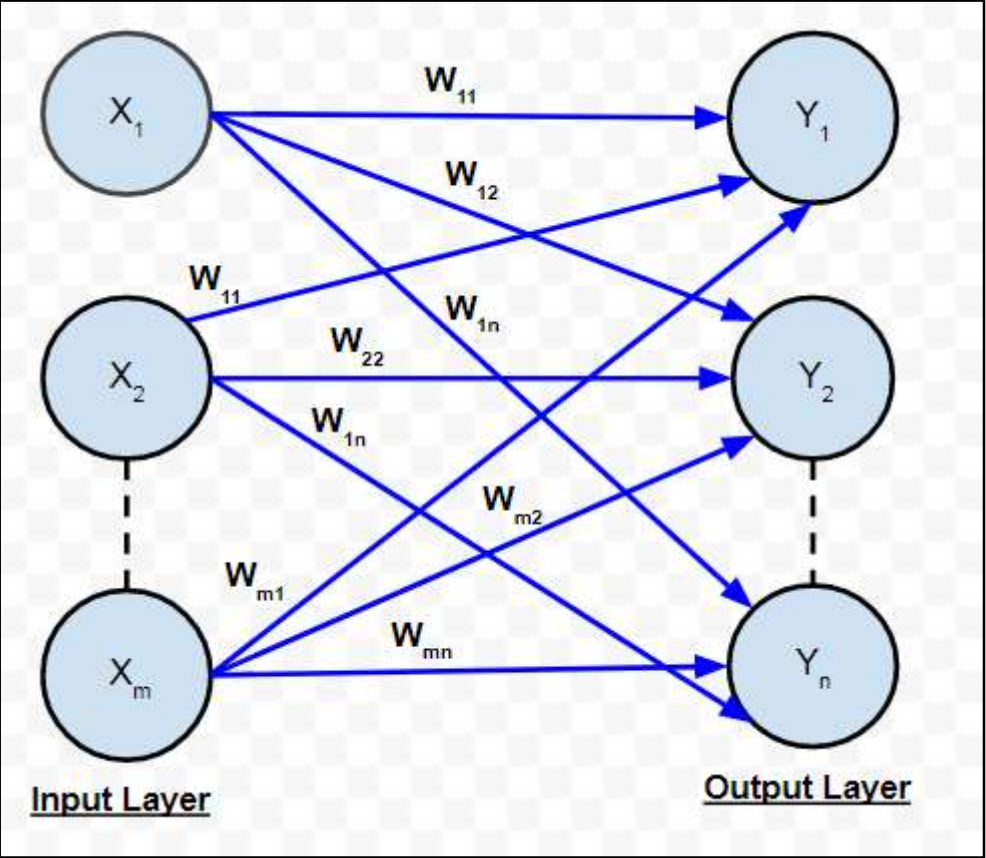
➤ The output layer is formed when different weights are applied to input nodes and the cumulative effect per node is taken. After this, the neurons collectively give the output layer to compute the output signals.

b) Multilayer feed forward network: The concept is of feed forward ANN having more than one weighted layer. As this network has one or more layers between the input and the output layer, it is called hidden layers.

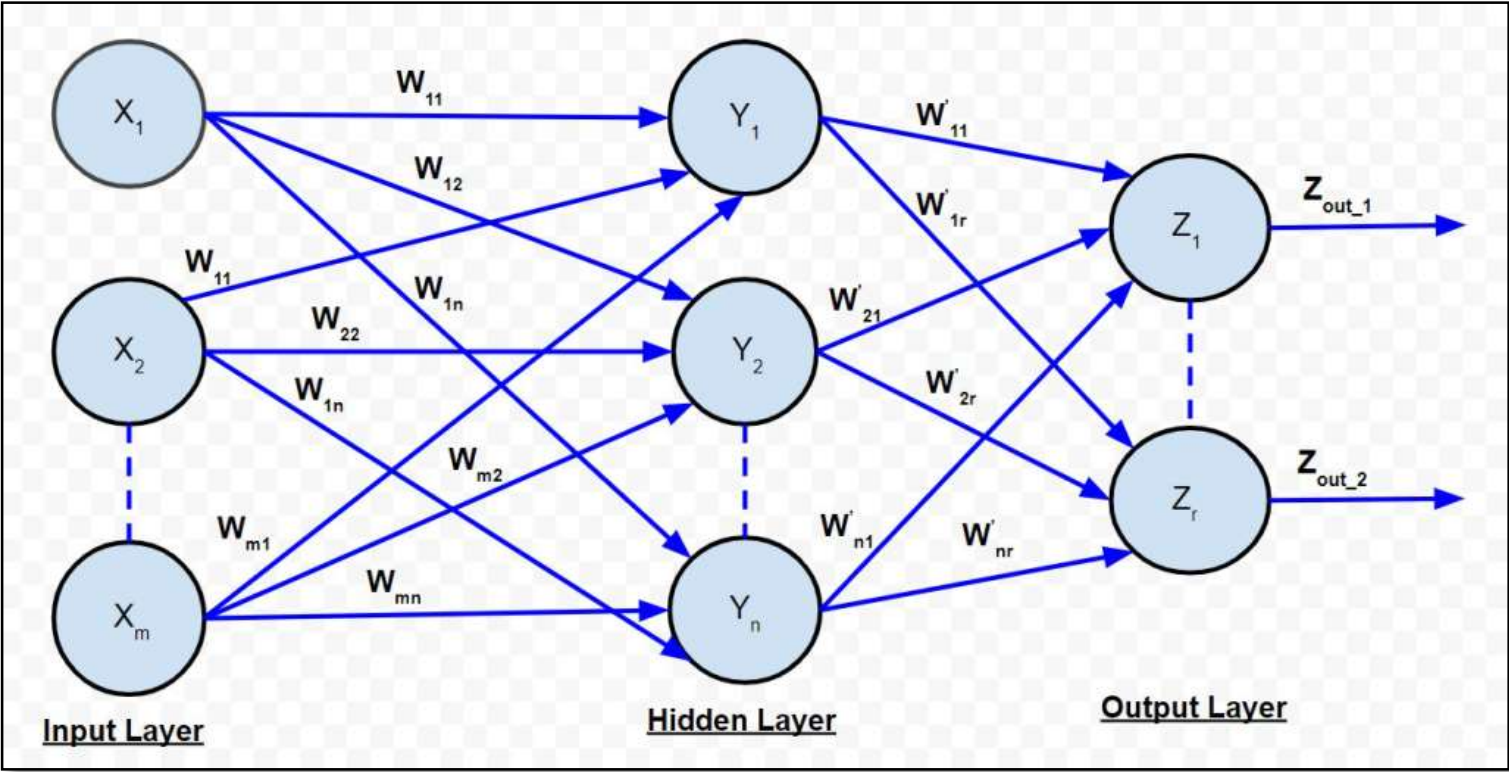
➤ The existence of one or more hidden layers enables the network to be computationally stronger, a feed forward network because of information flow



NEURAL NETWORK ARCHITECTURES



Single layer feed forward network



Multi-layer feed forward network



NEURAL NETWORK ARCHITECTURES

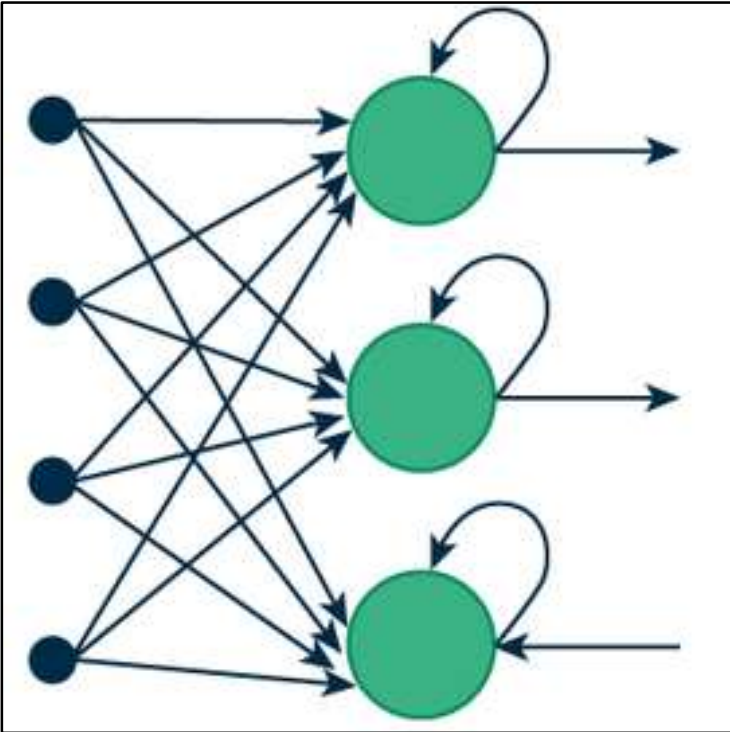
2. Recurrent Network: A recurrent network is a type of **feedback network** in which connections between the units form a **directed cycle**, allowing information to **flow in both forward and backward directions** through feedback loops. This gives the network a kind of **memory**, as the output from one step is used as input for the next. Recurrent networks are especially useful for processing **sequential or time-dependent data**, such as speech, text, or time series signals.

It may be divided into the following types:

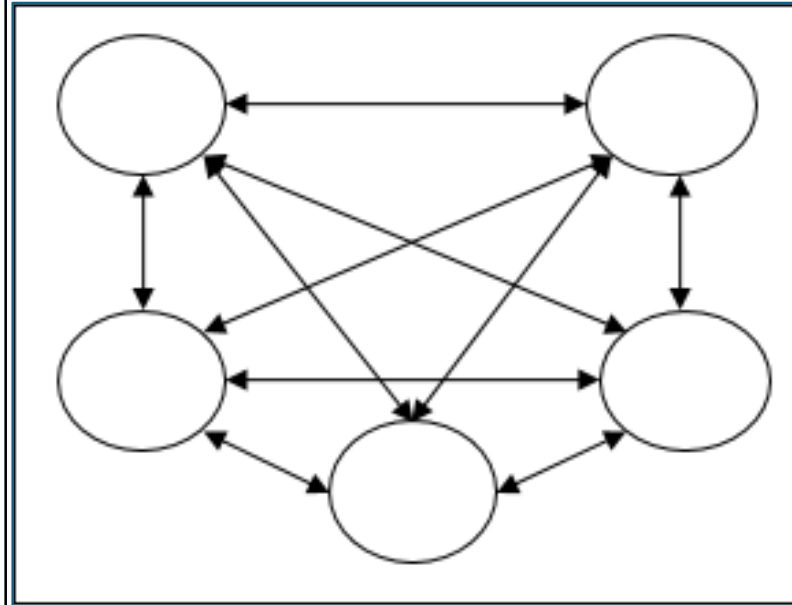
- a) Simple Recurrent Network (SRN):** Also known as an **Elman network**, this structure has connections from the hidden layer back to itself through **context units** that store the previous hidden state. The network learns temporal patterns by updating its state at each time step based on both the current input and the previous state.
- b) Fully Recurrent Network:** In this type, **every neuron is connected to every other neuron**, forming complete feedback loops. Each neuron acts both as an **input and an output unit**, and the system continues to update its state dynamically until it reaches equilibrium.



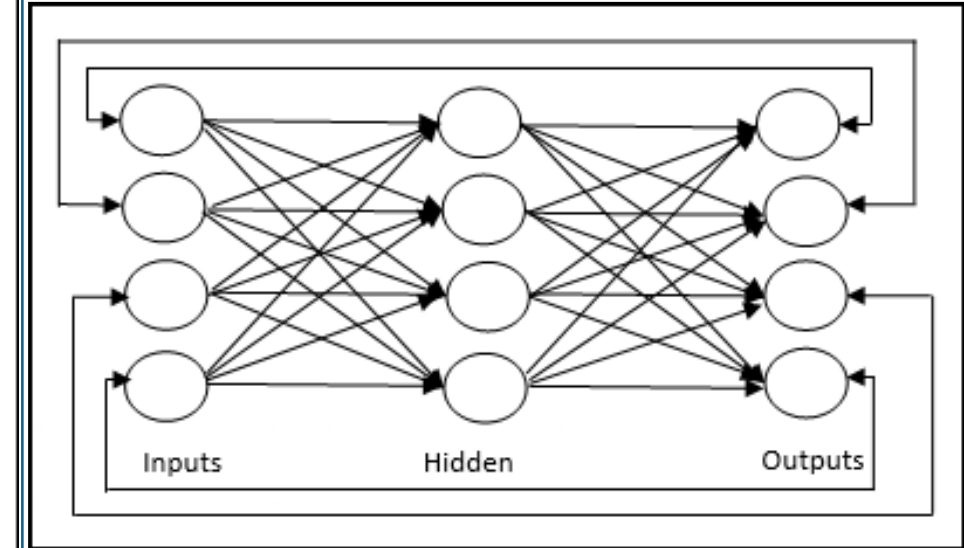
NEURAL NETWORK ARCHITECTURES



Simple Recurrent Network
(SRN)



Fully Recurrent
Network



Jordan
Network



ARTIFICIAL NEURAL NETWORK (ANN)

A single perceptron, or neuron, can be likened to a logistic regression model. An Artificial Neural Network (ANN) consists of multiple perceptrons/neurons arranged in layers. ANNs are also referred to as Feed-Forward Neural Networks because data is processed in a unidirectional flow from input to output.

- ANNs are capable of learning any nonlinear function, earning them the title of Universal Function Approximators. They have the ability to learn weights that map any input to its corresponding output.
- A key factor behind their universal approximation capability is the activation function. Activation functions introduce nonlinearity into the network, enabling it to learn and model complex relationships between inputs and outputs.



TRAINING OF ANN

We can train the neural network by feeding it by teaching patterns and letting it change its weight according to some learning rule. We can categorize the learning situations as follows.

- **Supervised Learning:** This type of learning is done under the supervision of a teacher. This learning process is dependent. During the training of ANN under supervised learning, the input vector is presented to the network, which will give an output vector. This output vector is compared with the desired output vector. An error signal is generated, if there is a difference between the actual output and the desired output vector. On the basis of this error signal, the weights are adjusted until the actual output is matched with the desired output.
- **Unsupervised Learning:** This type of learning is done without the supervision of a teacher. This learning process is independent. During the training of ANN under unsupervised learning, the input vectors of similar type are combined to form clusters. When a new input pattern is applied, then the neural network gives an

TRAINING OF ANN

- **Reinforcement Learning:** This type of learning is used to reinforce or strengthen the network over some critic information. This learning process is similar to supervised learning; however, we might have very less information. During the training of network under reinforcement learning, the network receives some feedback from the environment. This makes it somewhat similar to supervised learning. However, the feedback obtained here is evaluative not instructive, which means there is no teacher as in supervised learning. After receiving the feedback, the network performs adjustments of the weights to get better critic information in future.



CLASSIFICATION BY BACKPROPAGATION

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.

Input:

- D , a data set consisting of the training tuples and their associated target values;
- l , the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.

Steps:

1. Initialize all **weights** and **biases** in *network*;
2. while terminating condition is not satisfied{
3. for each training tuple X in D {
4. // Propagate the inputs forward:
5. for each input layer unit j {
6. $O_j = I_j$; // output of an input unit is its actual input value
7. for each hidden or output layer unit j {
8. $I_j = \sum_i w_{ij} O_i + \theta_j$; // compute the net input of unit j w.r.t the previous layer, i
9. $O_j = f(I_j) = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j (Assume Sigmoid AF)
10. // Backpropagate the errors:

Continued...



CLASSIFICATION BY BACKPROPAGATION

```
10.      // Backpropagate the errors:
11.      for each unit  $j$  in the output layer
12.           $Err_j = O_j(1 - O_j)(T_j - O_j)$ ; // compute the error
13.      for each unit  $j$  in the hidden layers, from the last to the first hidden layer
14.           $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$ ; // compute the error with respect to the next higher layer,  $k$ 
15.      for each weight  $w_{ij}$  in network{
16.           $\Delta w_{ij} = (l) Err_j O_i$ ; // weight increment
17.           $w_{ij} = w_{ij} + \Delta w_{ij}$ ; } // weight update
18.      for each bias  $\theta_j$  in network{
19.           $\Delta \theta_j = (l) Err_j$ ; // bias increment
20.           $\theta_j = \theta_j + \Delta \theta_j$ ; // bias update
21.      }
22. }
```



CLASSIFICATION BY BACKPROPAGATION

The model updates its weights and biases **immediately after processing each single training example (tuple)**. This is referred to as **case updating**. Alternatively, The model processes all training examples (the entire training set), accumulates all the calculated adjustments (increments), and then updates the weights and biases only once at the very end of the cycle. This full cycle is called an epoch. This latter strategy is called **epoch updating**, where one iteration through the training set is an **epoch**. In the

Strategy	How it Relates to the Epoch
Epoch Updating (Batch Gradient Descent)	Updates the weights once after the entire epoch is complete. (The original, theoretical approach).
Case Updating (Stochastic Gradient Descent)	Updates the weights N times (where N is the dataset size) during one epoch .
Mini-Batch Updating (Most Common)	Updates the weights $N/ Batch_Size$ times during one epoch .

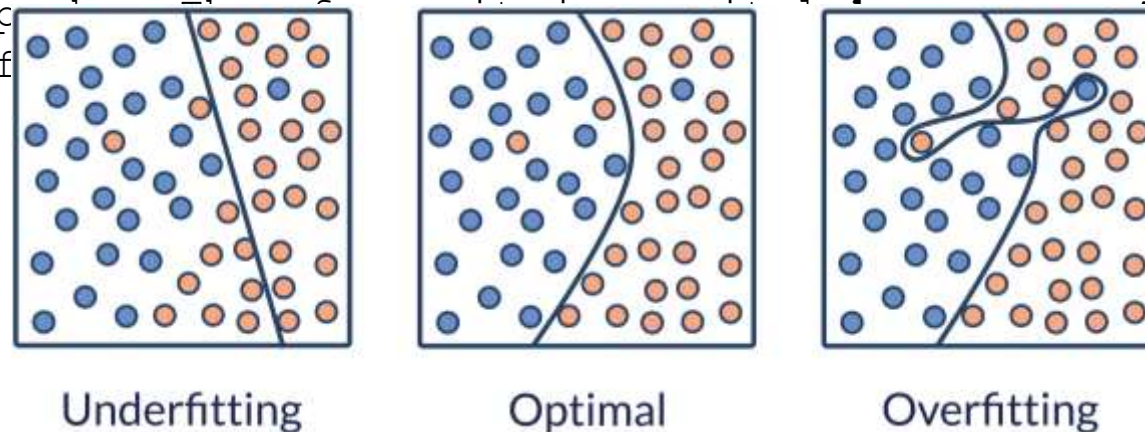


CLASSIFICATION BY BACKPROPAGATION

The concept of an **epoch** is present as a hyperparameter for three main reasons:

1. It Defines the Training Loop.
2. It Acts as the Anchor for All Other Strategies.
3. It Prevents Underfitting and Controls Overfitting .
 - Even with highly efficient updating methods (like Mini-Batch), the model needs to see the data multiple times (i.e., multiple epochs) to fully learn.
 - **Low Epoch Count:** The model will likely **underfit**, failing to capture the complex relationships in the data.
 - **High Epoch Count:** The model starts to **overfit** because it begins to memorize the noise and specific details of the training set rather than the general underlying patterns.

The primary way to control this crucial balance between underfitting and overfitting is by adjusting the number of epochs. **Epoch**—a setting that controls the learning process itself

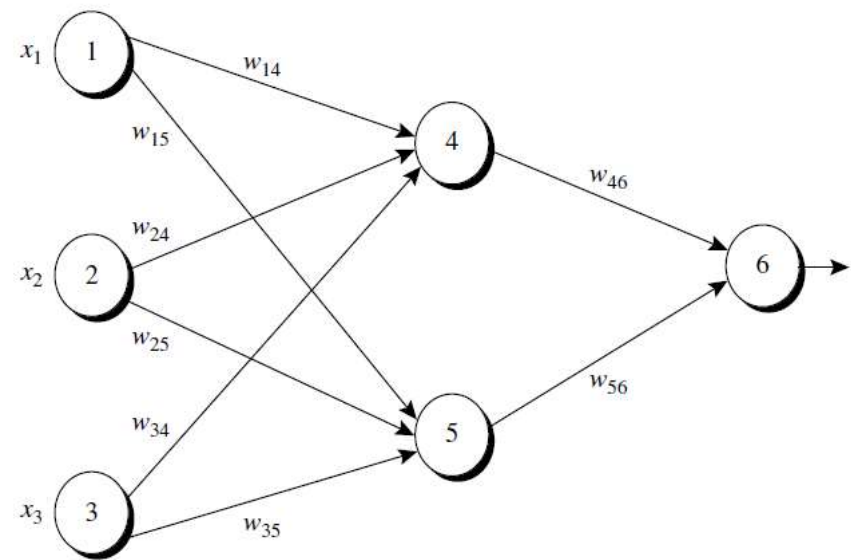


CLASSIFICATION BY BACKPROPAGATION

- Training stops when all weight updates (Δw_{ij}) in the previous epoch become very small, falling below a predefined threshold. This indicates that the weights and biases are no longer changing significantly from one epoch to another. Such minimal change suggests the model has **converged** to a stable point in the **error surface**. Continuing training beyond this point would **not lead to meaningful improvement** and would only **consume extra computational resources**.
- Training stops when the proportion of misclassified samples in the previous epoch falls below **a specified threshold**. This means the model has achieved a satisfactory level of performance – for example, a sufficiently **low error rate** or a **desired accuracy level** on the **training or validation data**. Once the model reaches the required accuracy, further training becomes unnecessary. This helps save time while ensuring the model meets its performance target.
- Training **stops when a fixed number of epochs** has been **completed**. This simply enforces a predefined upper limit on the number of training cycles the model can go through. This acts as a safety or control mechanism to prevent the training process from running indefinitely. It also helps **limit overfitting and computational cost**, especially when other stopping criteria are not met.



CLASSIFICATION BY BACKPROPAGATION



Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

- Training tuple $(X) = [1,0,1]$
- Class label $(y) = 1$
- learning rate $(l/\eta) = 0.9$
- Activation Function = Sigmoid Activation Function



CLASSIFICATION BY BACKPROPAGATION

Initial Input, Weight, and Bias Values, **Activation Function** =

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6	y	l/η
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1	1	0.9

$$I_j = \sum_i w_{ij} o_i + \theta_j$$

$$o_j = f(I_j) = \frac{1}{1 + e^{-I_j}}$$

Unit (j)	Net Input (I_j)	Output (O_j)
4	$0.2 + 0 - 0.5 - 0.4 = -0.7$	$1 / (1 + e^{0.7}) = 0.332$
5	$-0.3 + 0 + 0.2 + 0.2 = 0.1$	$1 / (1 + e^{-0.1}) = 0.525$
6	$(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$	$1 / (1 + e^{0.105}) = 0.474$

$$Err_j = O_j(1 - O_j)(T_j - O_j)$$

(Output Layer)

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$$

(Hidden Layer)

(Output

(Hidden

Unit (j)	Error _j
6	$(0.474)(1 - 0.474)(1 - 0.474) = 0.1311$
5	$(0.525)(1 - 0.525)(0.1311)(-0.2) = -0.0065$
4	$(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$



CLASSIFICATION BY BACKPROPAGATION

Initial Input, Weight, and Bias Values, **Activation Function** =

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6	y	l/η
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1	1	0.9

Unit (j)	Output (O_j)
4	0.332
5	0.525
6	0.474
Unit (j)	Error $_j$
6	0.1311
5	-0.0065
4	-0.0087

$$\Delta w_{ij} = (l)Err_j O_i$$
$$w_{ij} = w_{ij} + \Delta w_{ij}$$
$$\Delta \theta_j = (l)Err_j$$
$$\theta_j = \theta_j + \Delta \theta_j$$

Weight/Bi as	New Value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$



CLASSIFICATION BY BACKPROPAGATION – LEARNING RATE ($l/\alpha/\eta$)

The **learning rate** is a crucial **hyperparameter** that determines the **size of the step** an optimization algorithm takes toward minimizing the **loss function**. It controls **how fast or slow** a model learns during training.

- **Goal:** The training process seeks to **minimize the loss function**, which measures how far the model's predictions deviate from the actual target values.
- **Step Size:** The learning rate scales the **gradient**, which represents the direction and steepness of the slope of the loss function. The model parameters (weights and biases) are updated in the **opposite direction** of the gradient to reduce the loss:

$$\text{New Weight} = \text{Old Weight} - (\text{Learning Rate} \times \text{Gradient})$$



CLASSIFICATION BY BACKPROPAGATION – LEARNING RATE ($l/\alpha/\eta$)

Range of Learning Rate

In most deep learning and machine learning applications, the learning rate typically lies in the range:

$$10^{-4} \leq \eta \leq 10^{-1}$$

Typical starting points include:

- **0.1 or 0.01** for shallow networks or linear models
- **0.001 or 0.0001** for deep neural networks (e.g., CNNs, RNNs, Transformers)

The optimal value often depends on the optimizer used and must be determined experimentally

Trade-off	Learning Rate Value	Effect on Training	Outcome
	Too High	The model takes very large steps toward the minimum.	May overshoot the minimum, causing the loss to oscillate or diverge .
	Too Low	The model takes extremely small steps.	Slow convergence , may get stuck in a poor local minimum.
	Just Right	Balanced step size.	Fast and stable convergence toward the minimum.



CLASSIFICATION BY BACKPROPAGATION – GRADIENT

The **gradient** is a **vector** that represents the **direction and rate of the steepest increase** of a function. In the context of **machine learning and neural networks**, it tells us **how much the loss function changes** with respect to each model parameter (weights and biases).

If we have a loss function $L(w)$, where w represents the weights of the model, the **gradient** of L with respect to w is:

$$\nabla L(w) = \frac{\partial L}{\partial w}$$

This derivative (or partial derivative for multiple parameters) tells us **how sensitive** the loss is to small changes in each weight.

- If the gradient is **positive**, increasing the weight increases the loss.
- If the gradient is **negative**, increasing the weight decreases the loss.
- If the gradient is **zero**, the loss is at a **local minimum** or **maximum**.



CLASSIFICATION BY BACKPROPAGATION – GRADIENT

The gradient is used to update model parameters as follows:

$$w_{new} = w_{old} \pm \eta \frac{\partial L}{\partial w}$$

- $w \rightarrow$ model weight
- $L \rightarrow$ loss function
- $\eta \rightarrow$ learning rate
- $\frac{\partial L}{\partial w} \rightarrow$ gradient

This update rule ensures the weights move **toward the direction of decreasing loss**.

Let's consider a simple quadratic loss function:

$$L(w) = (w - 3)^2$$

Then the gradient is:

$$\frac{dL}{dw} = 2(w - 3)$$

If $w = 0$:

$$\frac{dL}{dw} = 2(0 - 3) = -6$$

The negative sign means the slope is downward toward **increasing** w – so the algorithm should increase w to reduce loss.



CLASSIFICATION BY BACKPROPAGATION – GRADIENT

The loss function as a **hill or valley**:

- The **height** represents the **loss value**.
- The **position** represents the current weight values.
- The **gradient** is like the **slope** of the hill at that point – it points **uphill** (the direction of increasing loss).

To minimize loss, the optimizer moves **downhill**, which means in the **opposite direction of the gradient**.

Concept	Meaning
Gradient	The slope or rate of change of the loss function with respect to model parameters.
Direction	Points toward the direction of maximum increase in loss.
Used For	Updating weights in the opposite direction (to minimize loss).
Zero Gradient	Indicates a minimum (or flat region) of the loss function.



CLASSIFICATION BY BACKPROPAGATION – GRADIENT DESCENT AND ITS VARIANTS

Gradient Descent (GD) is an optimization algorithm used to iteratively adjust model parameters to minimize the loss function. Different variants of GD use different amounts of data to estimate the gradient in each iteration.

1. Batch Gradient Descent (BGD)

- **Data Used:** The **entire training dataset** is used to compute the gradient once per epoch.
- **Updates:** Parameters are updated **after processing the whole dataset**.
- **Pros:** Smooth, stable convergence toward the global minimum.
- **Cons:** Very **slow** and **computationally expensive** for large datasets.

2. Stochastic Gradient Descent (SGD)

- **Data Used:** The gradient is computed using **only one training sample** at a time.
- **Updates:** Parameters are updated **after each sample**, leading to many updates per epoch.
- **Pros:** Very **fast** and can help escape **local minima** due to noisy updates.



CLASSIFICATION BY BACKPROPAGATION – GRADIENT DESCENT AND ITS VARIANTS

Gradient Descent (GD) is an optimization algorithm used to iteratively adjust model parameters to minimize the loss function. Different variants of GD use different amounts of data to estimate the gradient in each iteration.

3. Mini-Batch Gradient Descent (MBGD)

- **Data Used:** The gradient is calculated using a **small subset (mini-batch)** of the dataset (commonly 32, 64, or 128 samples).
- **Updates:** One update per mini-batch.
- **Pros:**
 - Faster and more **efficient** than full batch gradient descent.
 - **More stable** than pure SGD.
 - Works well with GPU hardware through vectorized computations.
- **Cons:** Requires tuning the **batch size**, which affects speed and convergence quality.



CLASSIFICATION BY BACKPROPAGATION – GRADIENT DESCENT AND ITS VARIANTS

Other Advanced Optimizers

Optimizer	Key Idea	Description
Momentum	Adds a fraction of the previous update to the current one	Accelerates convergence and reduces oscillations – like a ball rolling downhill gaining speed.
Nesterov Accelerated Gradient (NAG)	Looks ahead at the next step before computing the gradient	Provides smoother and faster convergence than standard momentum.
Adagrad	Adjusts learning rate for each parameter individually	Works well for sparse data, but learning rate may shrink too much over time.
RMSProp	Maintains a moving average of squared gradients	Prevents learning rate decay; suitable for non-stationary problems.
Adam (Adaptive Moment Estimation)	Combines Momentum + RMSProp	Automatically adapts the learning rate for each parameter; widely used in deep learning due to its fast, reliable convergence.



DERIVATION OF BACKPROPAGATION FOR A NETWORK

- Input layer unit j : $O_j = I_j$
- For each hidden / output unit j :

$$I_j = \sum_i w_{ij} O_i + \theta_j, \quad O_j = f(I_j) = \sigma(I_j) = \frac{1}{1 + e^{-I_j}}$$

- Sum-of-squares loss (single pattern):

$$E = \frac{1}{2} \sum_j (T_j - O_j)^2$$

- Learning rate: ℓ (you used l).
- $w_{ij_{new}} \leftarrow w_{ij_{old}} + \Delta w_{ij}$
- $\theta_{j_{new}} \leftarrow \theta_{j_{old}} + \Delta \theta_j$
- The gradients $\frac{\partial E}{\partial w_{ij}}$ and $\frac{\partial E}{\partial \theta_j}$ to get updates.

1. Derivative of the sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \frac{d\sigma}{dz} = \sigma(z) (1 - \sigma(z))$$

So for neuron j :

$$\frac{dO_j}{dnet_j} = O_j(1 - O_j)$$



DERIVATION OF BACKPROPAGATION FOR A NETWORK

Weight can influence the rest of the network through . Therefore, we can use chain rule to write

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial net_j} O_i\end{aligned}$$

- $net_j = \sum_i w_{ji} O_i$
- $\frac{\partial net_j}{\partial w_{ij}} = O_i$

To derive $\frac{\partial E}{\partial net_j}$

We consider two cases,

- **Case 1:** Where Unit j is an Output unit for the Network.
- **Case 2:** Where Unit j is a Hidden/Internal unit for the Network.



DERIVATION OF BACKPROPAGATION FOR A NETWORK

Case 1: Output-layer error signal (delta)

We compute the partial derivative of the loss w.r.t. the net i:

$$\frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial O_j} \cdot \frac{dO_j}{dnet_j}$$

Now

$$\frac{\partial E}{\partial O_j} = \frac{\partial \frac{1}{2} \sum_j (T_j - O_j)^2}{\partial O_j}$$

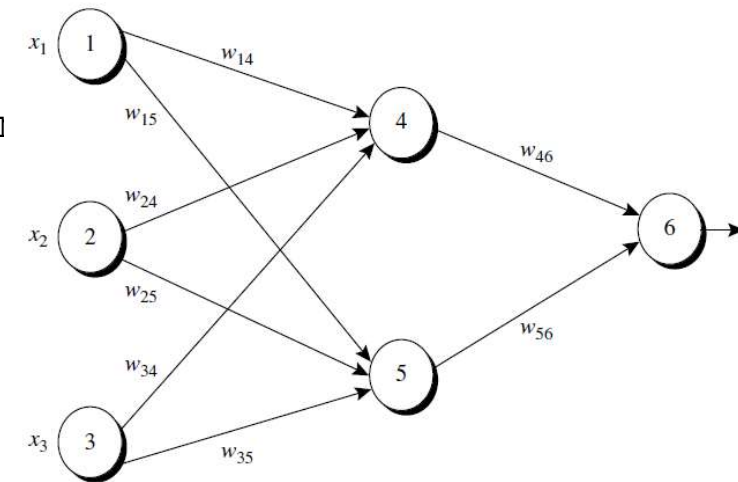
$$\frac{1}{2} \times 2(T_j - O_j) \frac{\partial (T_j - O_j)}{\partial O_j} \\ = -(T_j - O_j)$$

$$\frac{dO_j}{dnet_j} = \frac{d\sigma(net_j)}{dnet_j}$$

$$= \sigma(net_j) (1 - \sigma(net_j)) \\ = O_j(1 - O_j)$$

Thus

$$\frac{\partial E}{\partial net_j} = -(T_j - O_j) O_j(1 - O_j)$$



The **error signal** (often called δ_j or here **Err_j** (as the negative of that derivative or as the quantity used directly in updates)). Using your sign convention (so that a **positive Err_j** points to increase in the parameter in the **direction that reduces the error**), we set:

$$\boxed{Err_j = O_j(1 - O_j) (T_j - O_j)}$$

(That is, $Err_j = -\frac{\partial E}{\partial net_j}$)



DERIVATION OF BACKPROPAGATION FOR A NETWORK

➤ Gradient w.r.t. weights and bias (output neuron)

Weight w_{ij} connects neuron i (previous layer) output O_i to neuron j .

Chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = -Err_j \cdot \frac{\partial}{\partial w_{ij}} \left(\sum_i w_{ij} O_i - \theta_j \right)$$

Using the relation $\frac{\partial E}{\partial net_j} = -Err_j$,

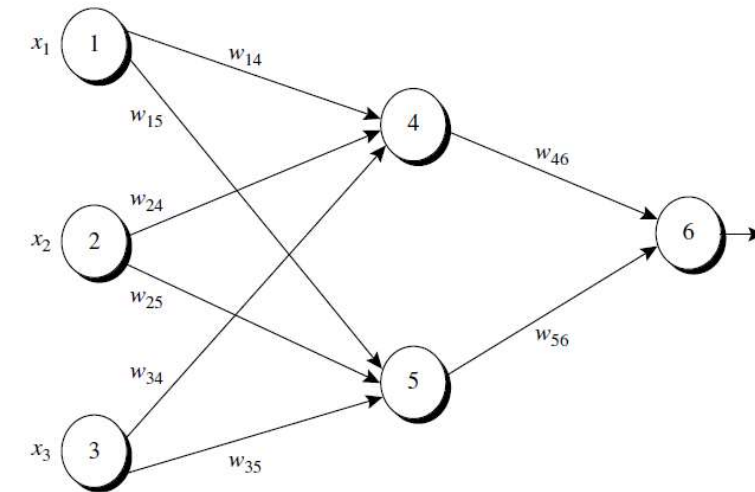
- $\frac{\partial E}{\partial w_{ij}} = -Err_j O_i.$

Using gradient-descent parameter update $w_{ij} \leftarrow w_{ij} - \ell \frac{\partial E}{\partial w_{ij}}$,

$$\Delta w_{ij} = -\ell \frac{\partial E}{\partial w_{ij}} = -\ell (-Err_j O_i) = \ell Err_j O_i.$$

So

$$\boxed{\Delta w_{ij} = \ell Err_j O_i.}$$



DERIVATION OF BACKPROPAGATION FOR A NETWORK

➤ Gradient w.r.t. weights and bias (output neuron)

For the bias (θ_j) recall net_j uses $-\theta_j$:

$$\frac{\partial net_j}{\partial \theta_j} = -1, \frac{\partial E}{\partial \theta_j} = \frac{\partial E}{\partial net_j} \cdot (-1) = -Err_j(-1) = Err_j$$

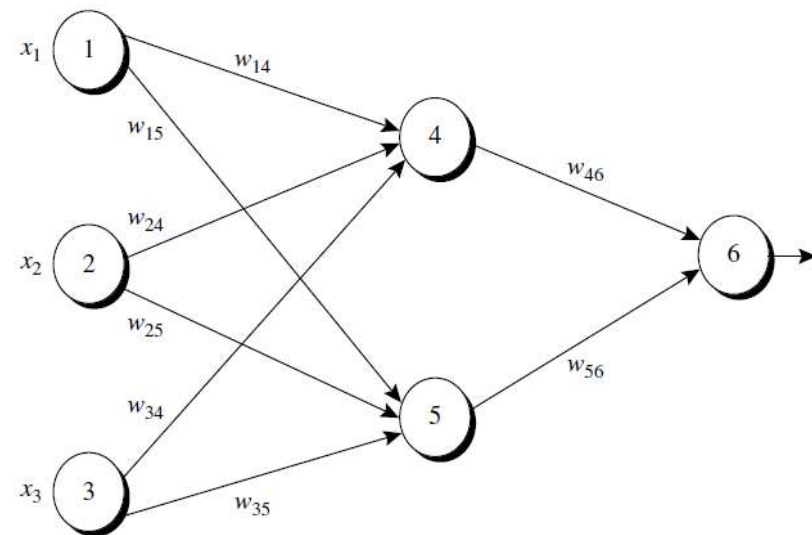
Then the update

$$\Delta \theta_j = -\ell \frac{\partial E}{\partial \theta_j} = -\ell Err_j,$$

but depending on sign convention of bias in the network definition you can also write the additive update as

$$\boxed{\Delta \theta_j = \ell Err_j}$$

if θ_j is considered with the opposite sign inside net_j). This matches your given $\Delta \theta_j = (\ell)Err_j$ so use the sign consistent with your network formulation.)



DERIVATION OF BACKPROPAGATION FOR A NETWORK

Case 2: Hidden-layer error signals (backpropagation)

For a hidden neuron h , it does not have a direct target. Its error signal is found by backpropagating the output errors through outgoing weights.

$$\frac{\partial E}{\partial net_h} = \sum_{j \in \text{next layer}} \frac{\partial E}{\partial net_j} \cdot \frac{\partial net_j}{\partial O_h} \cdot \frac{\partial O_h}{\partial net_h}$$

- $\frac{\partial net_j}{\partial O_h} = \frac{\partial x_{hj} w_{hj}}{\partial o_{hj}} = \frac{\partial o_j w_{hj}}{\partial o_{hj}} = w_{hj}$
- $\frac{dO_h}{dnet_h} = \frac{d\sigma(net_h)}{dnet_h} = \sigma(net_h)(1 - \sigma(net_h)) = O_h(1 - O_h)$
- $\frac{\partial E}{\partial net_j} = -\text{Err}_j$

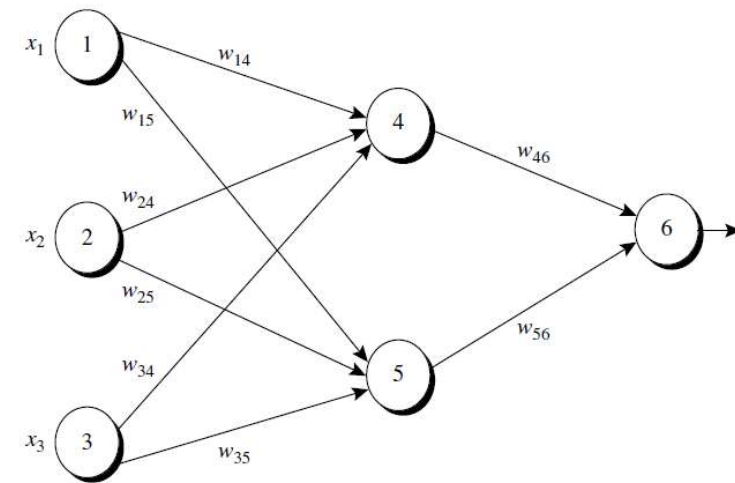
Thus (taking $\text{Err}_h = -\frac{\partial E}{\partial net_h}$ consistent with output-case sign),

$$\text{Err}_h = O_h(1 - O_h) \sum_j \text{Err}_j w_{hj}.$$

So the hidden error signal is the weighted sum of downstream error signals times the local derivative. Then weight updates for hidden-to-previous weights w_{ih} follow the same rule:

$$\Delta w_{ih} = \eta \text{Err}_h O_i.$$

Introduction to AIML



DERIVATION OF BACKPROPAGATION FOR A NETWORK

Output neuron j :

- Error signal:

$$\text{Err}_j = O_j(1 - O_j)(T_j - O_j).$$

- Weight update:

$$\Delta w_{ij} = \ell \text{Err}_j O_i.$$

- Bias update (sign depends on bias convention; using your form)

$$\Delta \theta_j = \ell \text{Err}_j.$$

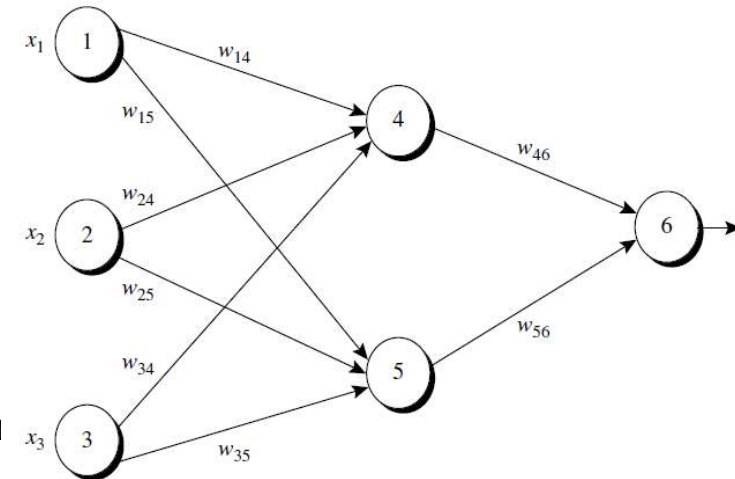
Hidden neuron h :

- Error signal:

$$\text{Err}_h = O_h(1 - O_h) \sum_j \text{Err}_j w_{hj}.$$

- Weight update (for weight from neuron i to hidden h):

$$\Delta w_{ih} = \ell \text{Err}_h O_i.$$



RADIAL BASIS FUNCTION NEURAL NETWORKS

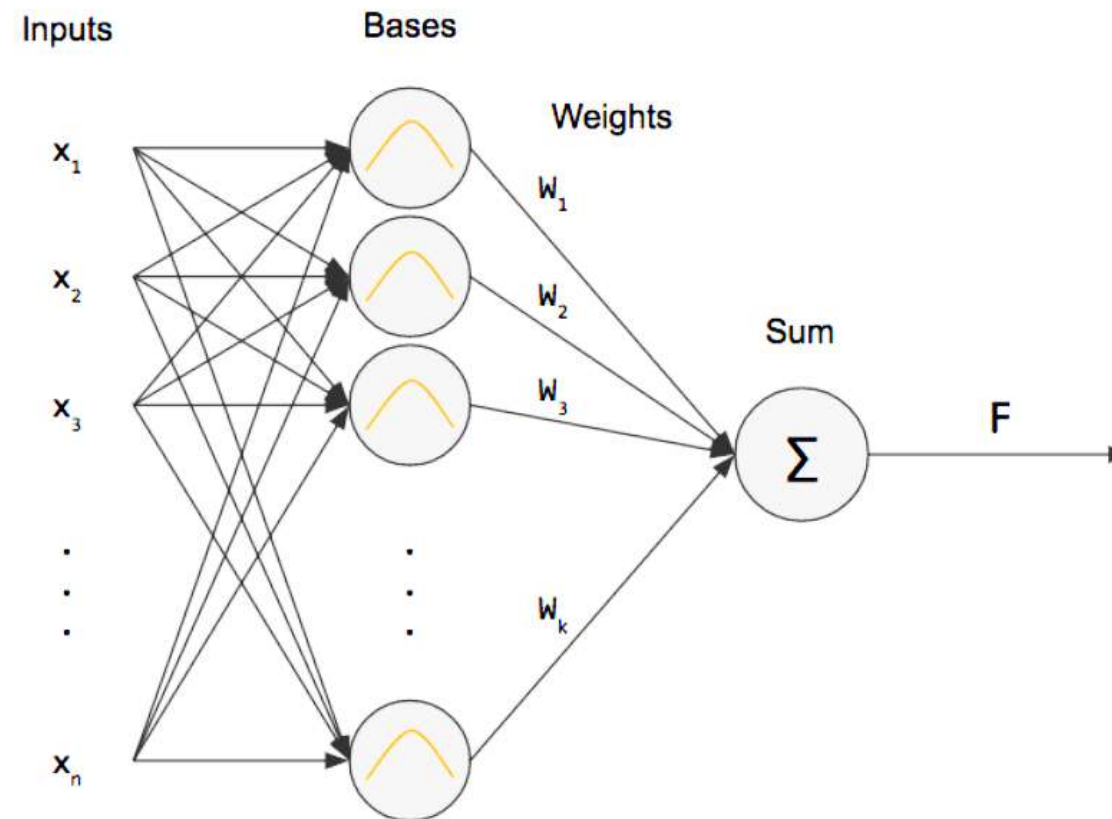
Radial Basis Function (RBF) Neural Networks are used for function approximation tasks. They are a special category of feed-forward neural networks comprising of three layers. Due to this distinct three-layer architecture and universal approximation capabilities they offer faster learning speeds and efficient performance in classification and regression problems.

How Do RBF Networks Work?

RBF Networks are conceptually similar to K-Nearest Neighbor (k-NN) models though their implementation is distinct. The fundamental idea is that nearby items with similar predictor variable values influence an item's predicted target value. Here's how RBF Networks operate:

- 1. Input Vector:** The network receives an n-dimensional input vector that needs classification or regression.
- 2. RBF Neurons:** Each neuron in the hidden layer represents a prototype vector from the training set. The network computes the Euclidean distance between the input vector and each neuron's center.
- 3. Activation Function:** The Euclidean distance is transformed using a Radial Basis Function (typically a Gaussian function) to compute the neuron's activation value. This value decreases exponentially as the distance increases.

ARCHITECTURE OF RBF NETWORKS



The architecture of an RBF Network typically consists of three layers:

ARCHITECTURE OF RBF NETWORKS

- **Function:** After receiving the input features the input layer sends them straight to the hidden layer.
- **Components:** It is made up of the same number of neurons as the characteristics in the input data. One feature of the input vector corresponds to each neuron in the input layer.

➤ Hidden Layer

- **Function:** This layer uses radial basis functions (RBFs) to conduct the non-linear transformation of the input data.
- **Components:** Neurons in the buried layer apply the RBF to the incoming data. The Gaussian function is the RBF that is most frequently utilized.
- **RBF Neurons:** Every neuron in the hidden layer has a spread parameter (σ) and a center which are also referred to as prototype vectors. The spread parameter modulates the distance between the center of an RBF neuron and the input vector which in turn determines the neuron's output.

➤ Output Layer

- **Function:** The output layer uses weighted sums to integrate the hidden layer neurons outputs to create the network's final output.
- **Components:** It is made up of neurons that combine the outputs of the hidden layer in a linear fashion. To reduce the error between the network's predictions and the actual target values, the weights of these combinations are adjusted during training.



ARCHITECTURE OF RBF NETWORKS

The input vector is $X = [x_1, x_2, \dots, x_n]^T$, and the network has m hidden neurons, then the output $Y = [y_1, y_2, \dots, y_n]^T$, is computed as:

$$y_j = \sum_{i=1}^m w_{ij} \phi_i(X)$$

where:

- w_{ij} : weight connecting the i^{th} hidden neuron to the j^{th} output neuron
- $\phi_i(X)$: activation of the i^{th} RBF neuron

Hidden Layer Activation Functions

Each hidden neuron computes its activation based on the **distance between the input X and its center C_i** :

$$r_i = (X - C_i)^2$$

The **RBF activation** depends only on this distance r_i .

The output of an RBFNN for an input vector X is:

$$y(X) = \sum_{i=1}^m w_i \phi_i(X)$$

where:

$$\phi_i(X) = \phi(X - C_i)^2$$

So, the full expression becomes:

$$y(X) = \sum_{i=1}^m w_i \phi(X - C_i)^2$$



TRAINING (LEARNING ALGORITHM)

The RBFNN training is typically **two-stage**:

Stage 1: Determine Hidden Layer Parameters

Find the **centers** C_i and **spreads** σ_i of the RBF neurons.

Methods:

1. **K-Means clustering** – find centers C_i

2. Compute spreads σ_i as:

$$\sigma_i = \frac{d_{max}}{\sqrt{2m}}$$

where d_{max} is the maximum distance between any two centers.



TRAINING (LEARNING ALGORITHM)

Stage 2: Determine Output Weights

After fixing centers and spreads, compute the RBF activations for all training samples and find the output weights W by solving:

$$Y = \Phi W$$

where:

Y : target output matrix (size $N \times k$)

Φ : activation matrix (size $N \times m$),

with $\Phi_{ji} = \phi(\|X_j - C_i\|)$

W : weights (size $m \times k$)

Solve for W using **least squares**:

$$W = (\Phi^T \Phi)^{-1} \Phi^T Y$$



ALGORITHM: RBFNN TRAINING

1. Initialize Parameters

1. Choose number of hidden neurons m
2. Select RBF type (e.g., Gaussian)
3. Initialize centers C_i (e.g., via K-Means)
4. Compute spreads σ_i

2. Compute Hidden Layer Activations

For each input X_j and hidden neuron i :

$$\Phi_{ji} = \phi(\|X_j - C_i\|)$$

3. Estimate Output Weights

Solve:

$$W = (\Phi^T \Phi)^{-1} \Phi^T Y$$

4. Prediction

For new input X :

$$\hat{Y}(X) = \sum_{i=1}^m w_i \phi(\|X - C_i\|)$$



ARCHITECTURE OF RBF NETWORKS

Type	Formula	Behavior
Gaussian RBF	$\phi(r) = e^{-\frac{r}{2\sigma^2}}$	Monotonically decreases with distance
Multiquadric RBF	$\phi(r) = \sqrt{r^2 + \sigma^2}$	Monotonically increases with distance
Inverse Multiquadric RBF	$\phi(r) = \frac{1}{\sqrt{r^2 + \sigma^2}}$	Monotonically decreases with distance
Linear RBF	$\phi(r) = r$	Linear growth with distance

