UNIT I

PROBLEM SOLVING

Introduction to AI

Artificial Intelligence

Thinking humanly, AI is the Intelligent activities that we associate with human thinking, activities such as decision-making, problem solving, learning, etc. these activities when done by computer automatically using some learning Algorithm.

Thinking rationally, It is the study of Advancing the mental capabilities of computer with the help of computational algorithms and intelligent learning models.

Acting humanly, It is the study of how to make computers do things at which, at the moment, people are better"

Acting rationally, "Artificial intelligence is the study of the design of intelligent agents"

Thinking humanly: cognitive modelling

- 1960s "cognitive revolution": information-processing psychology
- Requires scientific theories of internal activities of the brain
- How to validate? Requires

  1) Predicting and testing behavior of human subjects (top-down) or

  2) Direct identification from neurological data (bottom-up)
- Both approaches (roughly, Cognitive Science and Cognitive Neuroscience)
- are now distinct from AI

  Try to understand how the mind works . how do we think?

  Two possible routes to find answers:
- by introspection . we figure it out ourselves!
- by experiment . draw upon techniques of psychology to conduct controlled experiments. (.Rat in a box.!)

Thinking rationally: laws of thought

- Aristotle: what are correct arguments/thought processes?
- Several Greek schools developed various forms of logic: notation and rules of
- derivation for thoughts; may or may not have proceeded to the idea of mechanization
- Direct line through mathematics and philosophy to modern AI
- Problems:

  –Not all intelligent behavior is mediated by logical deliberation

–What is the purpose of thinking? What thoughts should I have?

- Trying to understand how we actually think is one route to AI . but how about how we should think.
- Use logic to capture the laws of rational thought as symbols.
- Reasoning involves shifting symbols according to well-defined rules (like algebra).
- Result is idealised reasoning.

Acting humanly: Turing Test

- Turing (1950) "Computing machinery and intelligence":
- "Can machines think?"
- “Can machines behave intelligently?"
- Operational test for intelligent behavior: the Imitation Game.
- Predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes
- Anticipated all major arguments against AI in following 50 years
- Suggested major components of AI: knowledge, reasoning, language understanding, learning
- Three rooms contain a person, a computer, and an interrogator.
- The interrogator can communicate with the other two by teleprinter.
- The interrogator tries to determine which is the person and which is the machine.
- The machine tries to fool the interrogator into believing that it is the person.
- If the machine succeeds, then we conclude that the machine can think.

A program that succeeded would need to be capable of:

- natural language understanding & generation;
- knowledge representation;
- learning;
- automated reasoning.

ELIZA: A program that simulated a psychotherapist interacting with a patient and successfully passed the Turing Test.

Acting rationally: Rational agent

- Rational behavior: doing the right thing.
- The right thing: that which is expected to maximize goal achievement, given the available information

- Doesn't necessarily involve thinking − e.g., blinking reflex − but thinking should be in the service of rational action
- An agent is an entity that perceives and acts
- This course is about designing rational agents
- Abstractly, an agent is a function from percept histories to actions:

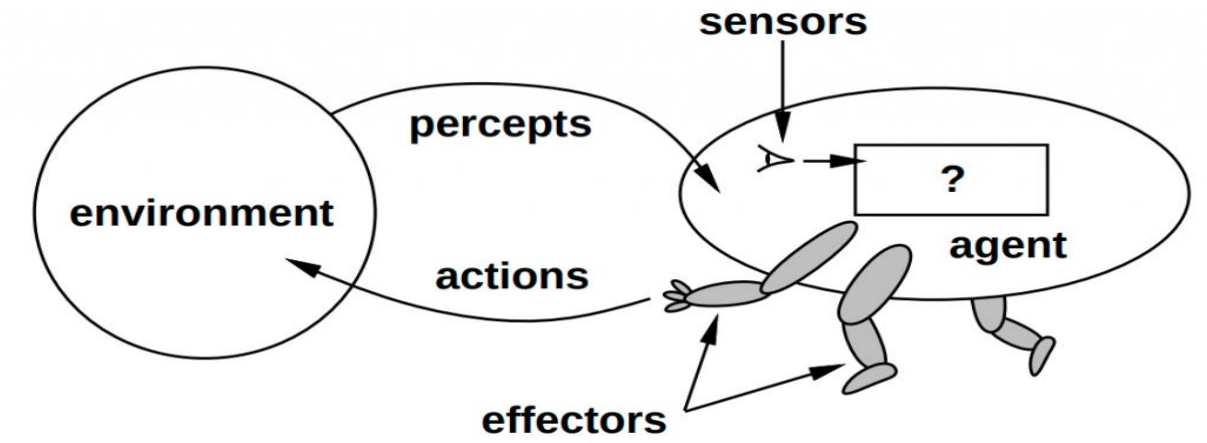
[f: P* → A]

- For any given class of environments and tasks, we seek the agent (or class of agents) with the best performance.
- Caveat: computational limitations make perfect rationality unachievable design best program for given machine resources

AI Applications

Agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators
- Human agent: eyes, ears, and other organs for sensors; hands, legs, mouth, and other body parts for actuators.
- Robotic agent: cameras and infrared range cameras are sensors; various motors, hydraulic arms and legs are actuators.



[*f*: P* → A]

•The agent program runs on the physical architecture to produce *f* agent = architecture + program

Automatic Vacuum-cleaner

•**Percepts:** location and contents, e.g., [A,Dirty]

•**Actions:** *Left*, *Right*, *Suck*, *NoOp*

A vacuum-cleaner agent

•input{tables/vacuum-agent-function-table}

Rational agents

•An agent should strive to "do the right thing", based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful

•Performance measure: An objective criterion for success of an agent's behavior

•E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

•Rational Agent: For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

•Rationality is distinct from omniscience (all-knowing with infinite knowledge)

•Agents can perform actions in order to modify future percepts so as to obtain useful information (information gathering, exploration)

•An agent is autonomous if its behavior is determined by its own experience (with ability to learn and adapt)

PEAS

•PEAS: Performance, Environment, Actuators, Sensor Must first specify

- Performance measure
- Environment
- Actuators
- Sensors

•Must first specify the setting for intelligent agent design

Agent: Self Driving Cars

•Consider, e.g., the task of designing an automated driverless car:

- **Performance measure:** Safe, fast, legal, comfortable trip, maximize profits
- **Environment:** Roads, other traffic vehicles, pedestrians, customers
- **Actuators:** Steering wheel, accelerator, brake, signal, horn

Sensors: Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard

Agent: Medical diagnosis system (Nurse bot)

- **Performance measure:** Healthy patient, minimize costs, lawsuits
- **Environment:** Patient, hospital, staff, medical instruments.
- **Actuators:** Screen display (tests recommender, disease diagnoses, treatments and prescription)
- **Sensors:** Keyboard (entry of symptoms, Blood pressure, temperature monitor, findings, patient's answers)

Agent: Part-picking robot (Assembly Arm Robot)

- **Performance measure:** Percentage of parts in correct bins
- **Environment**: Conveyor belt with parts, bins
- **Actuators**: Jointed arm and hand
- **Sensors**: Camera, joint angle sensors

Agent: Interactive English tutor (Chat bot)

- **Performance measure**: increase student's grade on test & spoken vocabulary.
- **Environment**: Set of students
- **Actuators**: Screen display (exercises, suggestions, corrections), dictionary
- **Sensors**: Keyboard, chat window input.

Environment types

Fully observable (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.

Deterministic (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is strategic)

Episodic (vs. sequential): The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.

Static (vs. dynamic): The environment is unchanged while an agent is deliberating. (The environment is semidynamic if the environment itself does not change with the passage of time but the agent's performance score does)

Discrete (vs. continuous): A limited number of distinct, clearly defined percepts and actions.

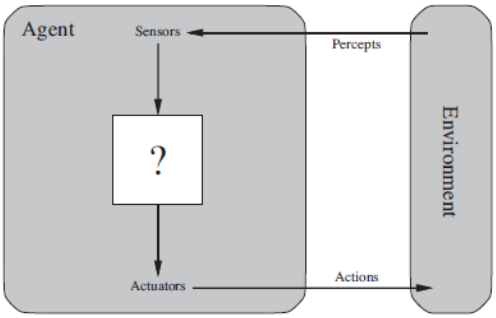
Single agent (vs. multiagent): An agent operating by itself in an environment.

Yes/No	Chess with Chess without	Taxi driving	a clock a clock
Fully / Partially observable	**Yes**	**Yes**	**No**
Deterministic / stochastic	Strategic	Strategic	**No**
Episodic/Sequential	**No**	**No**	**No**
Static/Dynamic	**Semi**	**Yes**	**No**
Discrete/Continuous	**Yes**	**Yes**	**No**
Single /Multi Agent	**No**	**No**	**No**

AGENT TYPES / STRUCTURE OF AGENTS

An **agent** is anything that perceive its **environment** through **sensors** and acting upon that environment through **actuators**.

Percept to refer to the agent's perceptual inputs at any given instant. An agent's **percept sequence** is the complete history of everything the agent has ever perceived



Rectangles: Current internal state of the agent's decision process

Ovals: Background information used in the process

Example: Agent function for vacuum agent

Program

Function REFLEX-VACUUM-AGENT*([location,status])* **returns** *an action*

If *status*=dirty **then**
return *suck*
Else if *location* = A **then**
return *right*
Else if *location* = B **then**
return *left*

The vacuum agent program is very small. But some processing is done on the visual input to establish the condition-action rule.

Percept sequence (Location, Status) (INPUT)	Action (OUTPUT)
[A, Clean]	Right
[A,Dirty]	Suck
[B,Clean]	Left
[B, Dirty]	Suck
[A,clean],[A,clean]	Right
[A,clean],[A,Dirty]	Suck
...	…
[A,clean],[A,clean], [A,clean]	Right
[A,clean],[A,clean], [A,dirty]	Suck

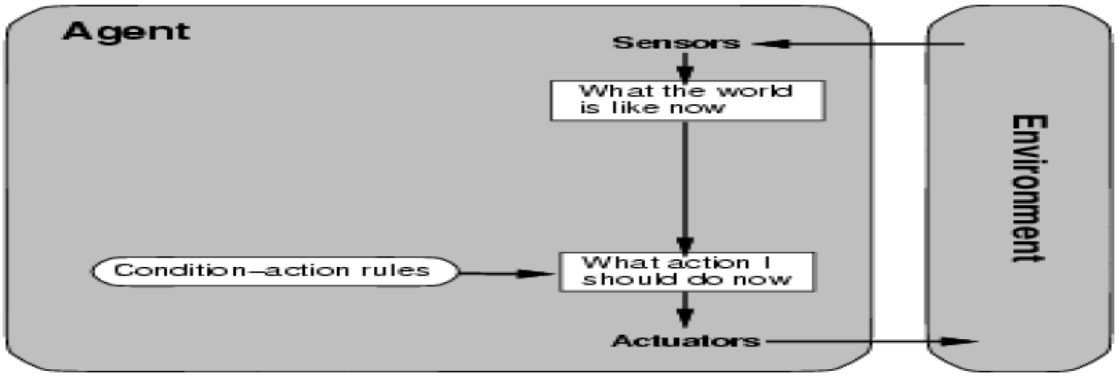
Five basic types in order of increasing generality:

- Simple reflex agents
- Model-based reflex agents
- Goal-based agents
- Utility-based agents
- Learning agents

Simple reflex agents

The simplest kind of agent is the simplex reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

The following figure shows how the condition-Action rules allow the agent to make the connection from percept to action.



Program

Function SIMPLE-REFLEX-AGENT(*percept*) **returns** an *action*

Static: rules, a set of condition-action rules

***State*← INTERPRET-INPUT (*percept*)**

***rule*← RULE-MATCH (*state*, *rule*)**

***action* ← RULE-ACTION [*rule*]**

Return *action*

State	Rule	Action
Vacuum_on(A, Clean)	Turn_left(Vacuum on)	Right
Vacuum_off(A,Dirty)	Turn_Right(Vacuum on)	Suck
Vacuum_on(B, Clean)	Turn_left(Vacuum on)	Left
Vacuum_off(B,Dirty)	Turn_Right(Vacuum on)	
	Turn_left(Vacuum off)	
	Turn_Right(Vacuum off)	

INTERPRET-INPUT: generates an abstracted description of the current state from the percept

RULE-MATCH: returns the first rule in the set of rules that matches the given state description. This agent will work only if the correct decision can be made on the basis of only the current percept. i.e. only if the environment is fully observable.

Model-based reflex agents

To handle partial observability, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

Updating this internal state information requires two kinds of knowledge to be encoded in the agent program.

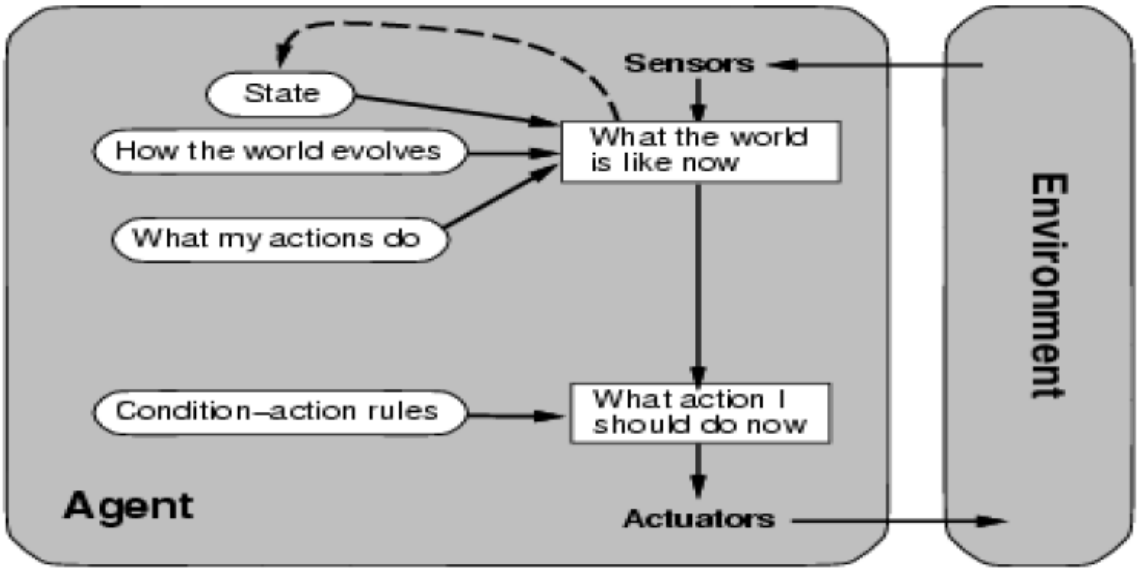
o How the world evolves independently of the agent

o How the agent's actions affect the world.

This knowledge can be implemented in simple Boolean circuits called model of the world. An agent that uses such a model is called a model-based agent.

The following figure shows the structure of the reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state.

The agent program is shown below:



Function REFLEX-AGENT-WITH-STATE(*percept*)**returns** an *action*

Static: state, a description of the current world state

Rules, a set of condition-action rules

Action, the most recent action, initially none

State <- UPDATE-STATE(*state*, *action* , *percept*)

Rule<- RULE-MATCH(*state*, *rule*)

Action<- RULE-ACTION[*rule*]

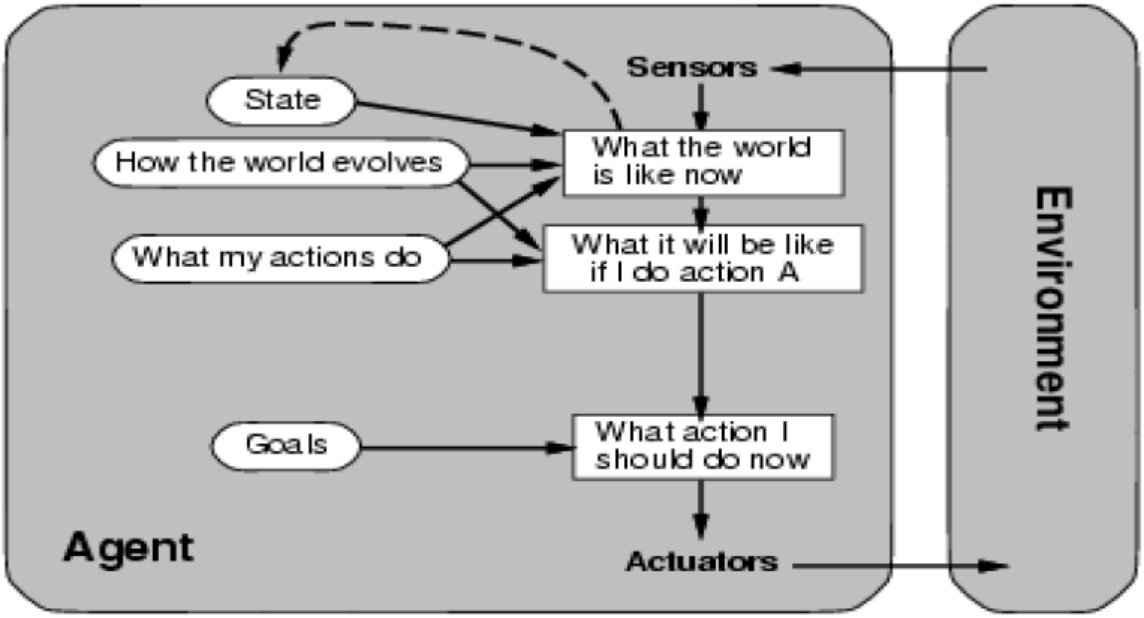
Return *action*

UPDATE-STATE: for creating the new internal state description.

State	Rule	Action
Vacuum_on(A, Clean, Vacuum_on)	Turn_left(Vacuum on)	Right
Vacuum_off(A,Dirty, Vacuum_off)	Turn_Right(Vacuum on)	Suck
Vacuum_on(B, Clean, Vacuum_off)	Turn_left(Vacuum on)	Left
Vacuum_off(B,Dirty, Vacuum_on)	Turn_Right(Vacuum on)	
	Turn_left(Vacuum off)	

Goal-based agents

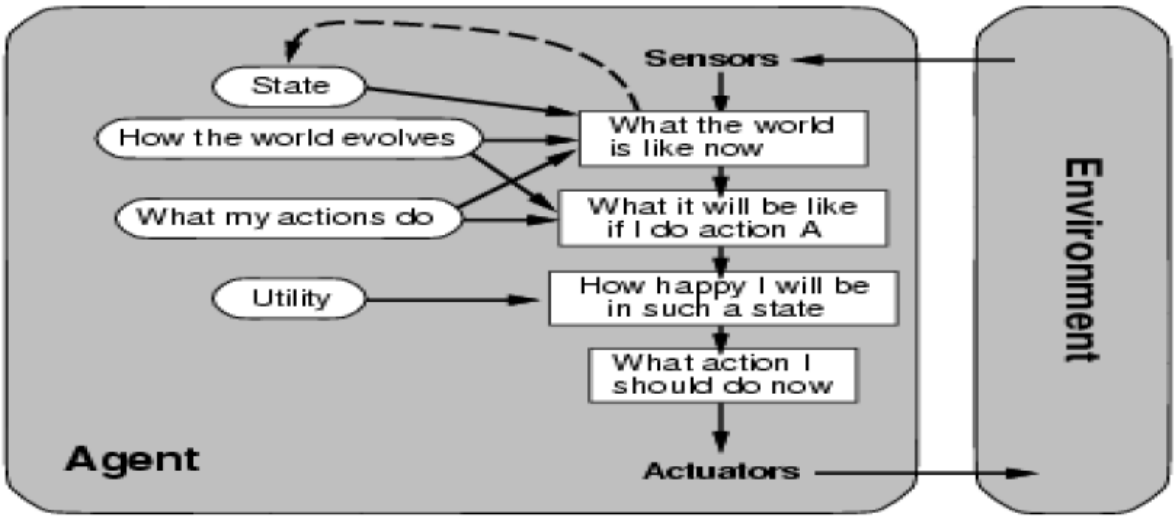
Here, along with current-state description, the agent needs some sort of goal information that describes situations that are desirable – for eg, being at the passenger's destination. Goal –based agents structure is shown below:



Utility-based agents

Goals alone are not enough to generate high-quality behaviour in most environments. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved.

A utility function maps a state onto a real number, which describes the associated degree of happiness. The utility-based agent structure appears in the following figure.

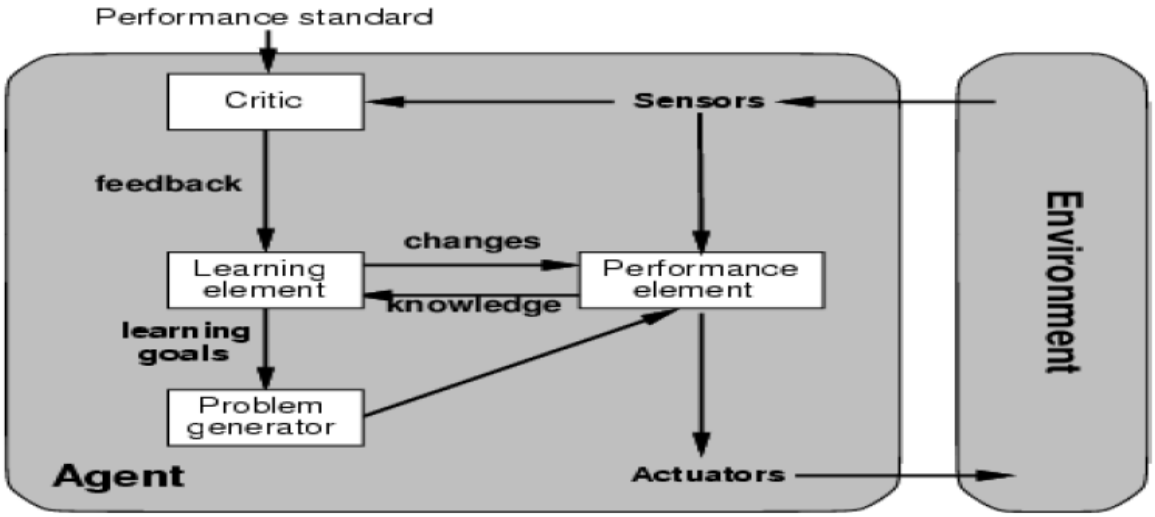


Learning agents

It allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. A learning agent can be divided into four conceptual components, as shown in figure:

Learning element: responsible for making improvement Performance element: responsible for selecting external actions The learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future.

The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent's success. The last component of the learning agent is the problem generator. It is responsible for suggesting actions that will lead to new and informative experiences.



PROBLEM-SOLVING AGENTS

Problem solving agent is a goal-based agent decides what to do by finding sequences of actions that lead to desirable states.

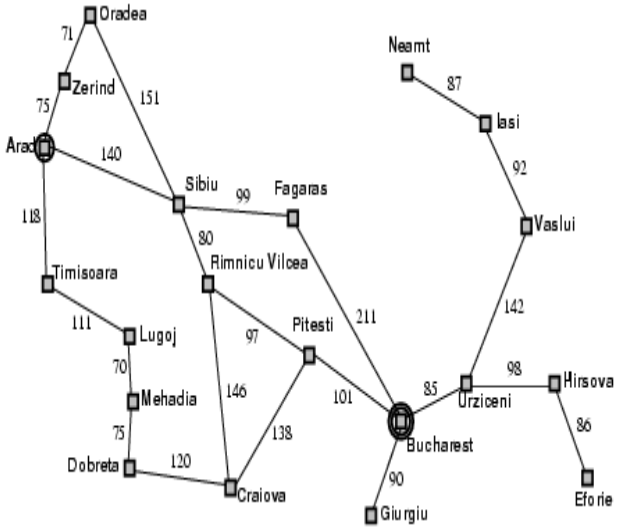
Let us take for an example, an Self Driving Car agent in the city of Arad, Romania, enjoying a touring holiday.

Goal formulation

- based on the current situation and the agent's performance measure, is the first step in problem solving.
- We will consider a goal to be a set of world states- exactly those states in which the goal is satisfied.

Problem formulation

- is the process of deciding what actions and states to consider, given a goal. Let us assume that the agent will consider actions at the level of driving from one major town to another.
- Our agent has now adopted the goal of driving to Bucharest, and is considering where to go from Arad. There are three roads out of Arad.
- The agent will not know which of its possible actions is best, because it does not know enough about the state that results from taking each action.
- If the agent has a map, it provides the agent with information about the states it might get itself into, and the actions it can take.
- An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.
- The process of looking for such a sequence is called a **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase**.
- The design for such an agent is shown in the following function:



function SIMPLE-PROBLEM-SOLVING-.AGENT (*percept*) **returns** an *action*
static: *seq*, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation
state ←UPDATE- STATE(*state*, *percept*)
if *seq* IS EMPTY **then do**
goal ←FORMULATE- GOAL(*state*)
problem ← FORMULATE-PROBLEM(*State, goal*)
seq +← SEARCH(problem)
action +← FIRST(*seq*)
seq← REST(*seq*)
return *action*

Example: Romania

- An human lives in Romania; currently in Arad.
- His Flight leaves tomorrow from Bucharest

Goal formulation

- **Formulate goal**: be in Bucharest
- **states**: various cities
- **actions**: drive between cities
- **Find solution:**sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem Formulation

A problem is defined by four items:

- **initial state** e.g., "at Arad"
- **actions** or successor function *S(x)* = set of action–state pairs
  - e.g., *S(Arad)* = {<*Arad* → *Zerind*, *Zerind*→ *oradea*,, *oradea* → *Sibiu*, *Sibu*→ *Fagams*, *Fagams*→ *Bucharest* }
- **goal test**, can be
  - explicit, e.g., *x* = "at Bucharest"
  - implicit, e.g., *Checkmate(x)*
- **path cost** (additive)

  e.g., sum of distances, number of actions executed, etc.

  c(x,a,y) is the step cost, assumed to be !A 0
- **solution** is a sequence of actions leading from the initial state to a goal state

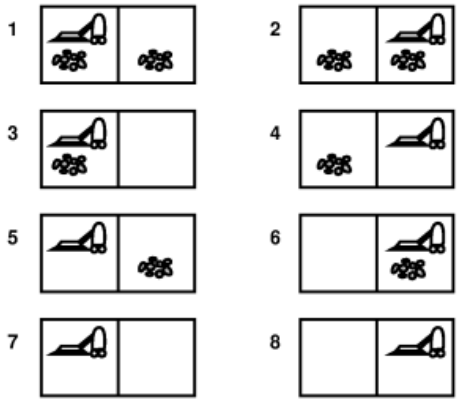
Selecting a state space

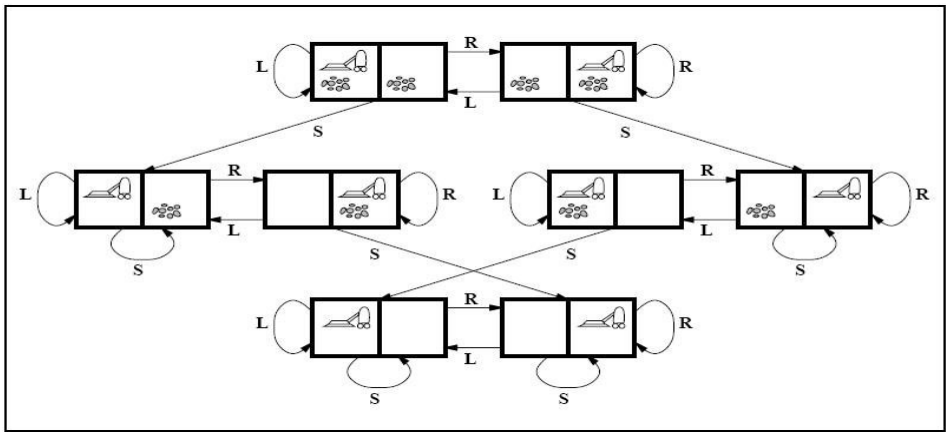
- Real world is absurdly complex
    - ♦ state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
    - ♦ e.g., "Arad →Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
    - ♦ (Abstract) solution = set of real paths that are solutions in the real world.
- Each abstract action should be "easier" than the original problem.

Types of Problem Solving Agents Based on States

Single-state problem

- Single-state problem Complete world state knowledge Complete action knowledge The agent always knows its world state
- Goal formulation → World states with certain properties
- Definition of the state space
- Definition of the actions that can change the world state.
- Definition of the problem type, which depends on the knowledge of the world states and actions states in the search space
- Specification of the search costs (search costs, offline costs) and the execution costs (path costs, online costs)





- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has n 2^n states.

• **Initial state:** Any state can be designated as the initial state.

• **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.

• **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect.

• **Goal test**: This checks whether all the squares are clean.

• **Path cost**: Each step costs 1, so the path cost is the number of steps in the pathIf the environment is completely accessible, the vacuum cleaner always knows where it is and where the dirt is.

The solution then is reduced to searching for a path from the initial state to the goal state.

If the environment is completely accessible, the vacuum cleaner always knows where it is and where the dirt is. The solution then is reduced to searching for a path from the initial state to the goal state.

Multi State Problem

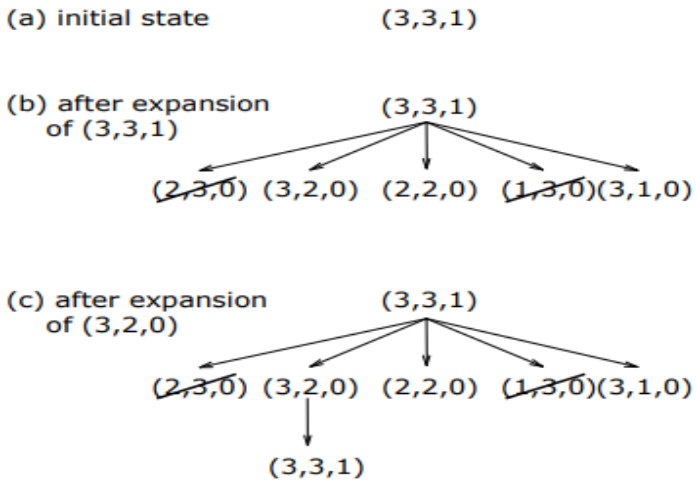
- Multiple-state problem has Incomplete world state knowledge Incomplete action knowledge
- The agent only knows which group of world states it is in.
- If the vacuum cleaner has no sensors, it doesn't know where it or the dirt is. In spite of this, it can still solve the problem.
- Here, states are knowledge states. States for the search: The power set of the world states 1-8.

- **Initial State** The state from which the agent infers that it is at the beginning
- **State Space** Set of all possible states
- **Actions** Description of possible actions and their outcome (successor function)
- **Goal Test** Tests whether the state description matches a goal state
- **Path** A sequence of actions leading from one state to another.
- **Path Costs** Cost function g over paths. Usually the sum of the costs of the actions along the path.
- **Solution Path** from an initial to a goal state
- **Search Costs** Time and storage requirements to find a solution
- **Total Costs** Search costs + path costs

Search algorithms:

Solving the formulated problem can be done by a search through the state space. One of the search technique is an explicit **search tree** that is generated by the initial state and the successor function that together define the state space.

From the initial state, produce all successive states step by step search tree.



Problem

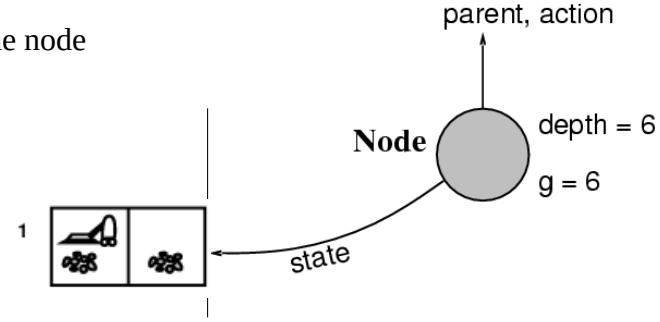
State: state in the state space

Parent-Node: Predecessor nodes

Action: The operator that generated the node

Depth: number of steps along the path from the initial state

Path Cost: Cost of the path from the initial state to the node



Operations on a queue:

- **Make-Queue**(Elements): Creates a queue
- **Empty?**(Queue): Empty test
- **First**(Queue): Returns the first element of the queue
- **Remove-First**(Queue): Returns the first element
- **Insert**(Element, Queue): Inserts new elements into the queue
- **Insert-All**(Elements, Queue): Inserts a set of elements into the queue

STATE vs NODE

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth

The node data structure is depicted in the following figure:

Tree search algorithms

- Offline, simulated exploration of state space by generating successors of already explored states (a.k.a.~expanding states)
- The following figure shows some of the expansions in the search tree for finding a route from Arad to Bucharest.
- The root of the search tree is a search node corresponding to the initial state, Arad.
- The first step is to test whether this is a goal state. If this is not the goal state, expand the current state by applying the successor function to the current state, thereby generating a new set of states.

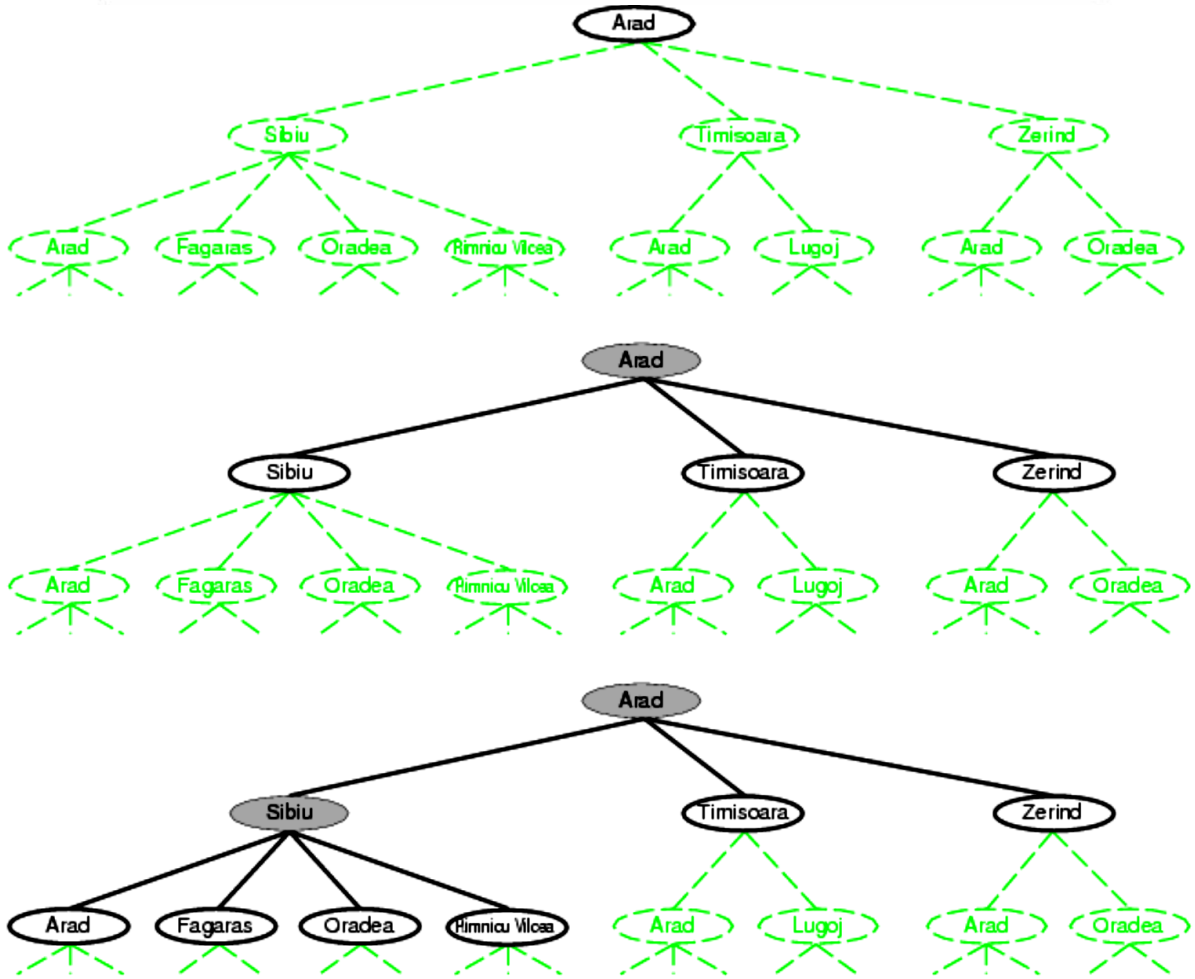
```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

---

function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    STATE[s] ← result
    PARENT-NODE[s] ← node
    ACTION[s] ← action
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```



UNINFORMED SEARCH STRATEGIES

Uninformed or **blind** search strategies use only the information available in the problem definition. Strategies that know whether one non-goal state is "more promising" than another are called **informed search or heuristic search strategies**.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Algorithm:

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop.

Otherwise,

4. Remove and expand the first element from the queue and place all the children at the end of the queue in any order.
5. Return to step 2.

- By calling TREE-SEARCH with an empty fringe
- *fringe* is a FIFO queue, i.e., new successors go at end

The following figure shows the progress of the search on a simple binary tree.

Figure: Breadth first search on a simple binary tree. At each state, the node to be expanded next is indicated by a marker.

Properties of breadth-first search

- **Complete**? Yes (if *b* is finite)

If the shallowest goal node is at some finite depth d, BFS will eventually find it after expanding all shallower nodes (b is a branching factor)

- **Time**? $1+b+b_2+b_3+\ldots+b_d+b(b_d-1) = O(b_{d+1})$

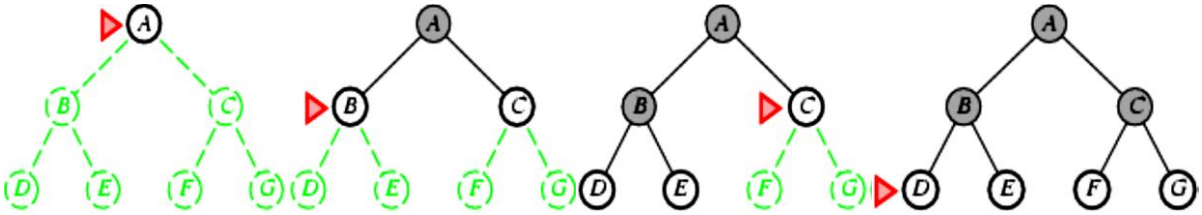
- **Space**? $O(b^{d+1})$ (keeps every node in memory)

We consider a hypothetical state space where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level, and so on. Now suppose that the solution is at depth d.

- **Optimal**? Yes (if cost = 1 per step)

BFS is optimal if the path cost is a nondecreasing funcion of the depth of the node.

- Space is the bigger problem (more than time)



Depth-first search

Always expands an unexpanded node at the greatest depth

fringe = LIFO queue, i.e., put successors at front

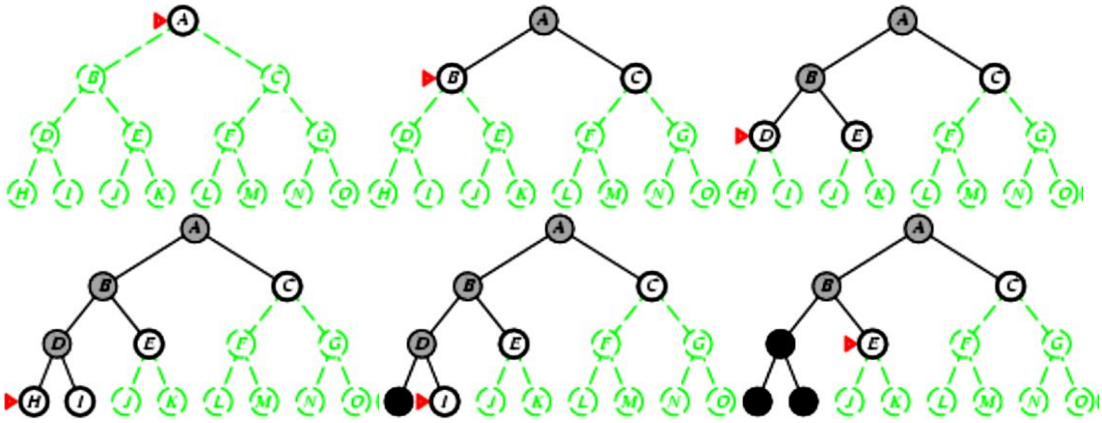
Algorithm:

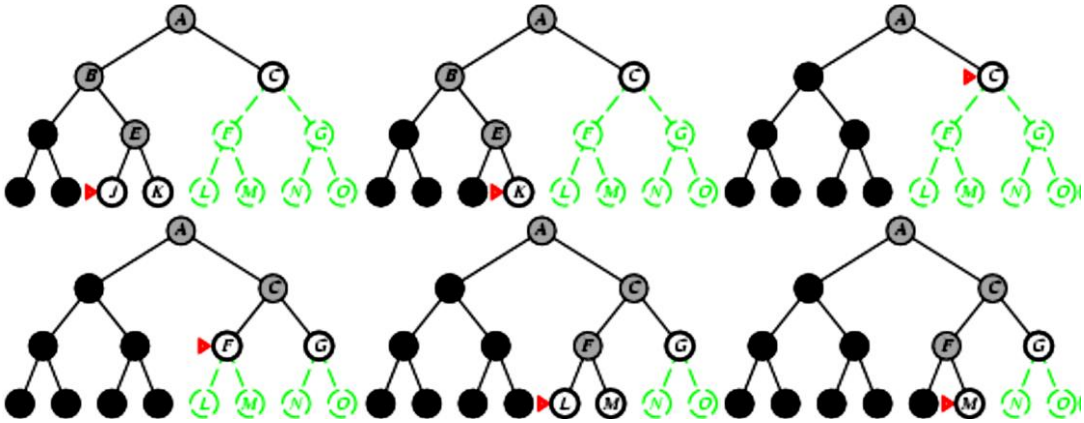
1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop.

Otherwise,

4. Remove and expand the first element , and place the children at the front of the queue (in any order).
5. Return to step 2.

The progress of the search is illustrated in the following figure:





DFS on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

Properties

Complete? No: fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path complete in finite spaces

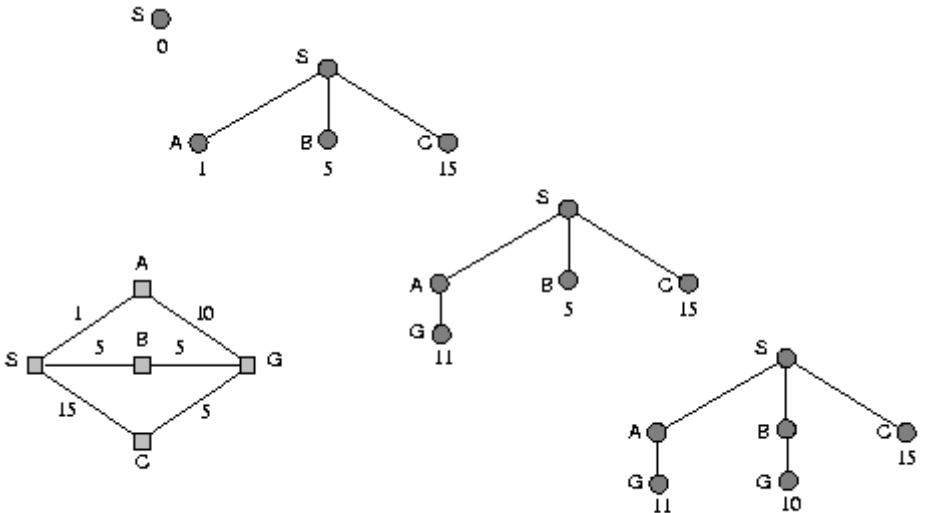
- **Time**? $O(b_m)$: terrible if m is much larger than d but if solutions are dense, It may be much faster than breadth-first Search.
- **Space**? $O(b_m)$, i.e., linear space!
- **Optimal**? No

Uniform-cost search

BFS is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. Instead of expanding the shallowest node, Uniform-cost search expands the node n with the lowest path cost.

fringe = queue ordered by path cost Equivalent to breadth-first if step costs all equal

- **Complete**? Yes, if step cost >= e
- **Time**? # of nodes with g <= cost of optimal solution, $O(b_{ceiling(C^*/e)})$ where C^* is the cost of the optimal solution
- **Space**? # of nodes with g <= cost of optimal solution, $O(b_{ceiling(C^*/e)})$
- **Optimal**? Yes – nodes expanded in increasing order of $g(n)$



Depth-limited search

The problem of unbounded trees can be alleviated by supplying DFS with a predetermined depth limit.

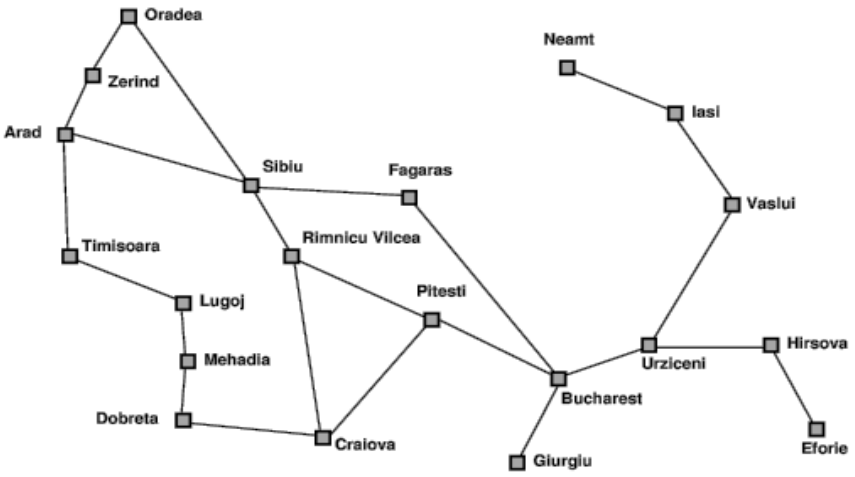
= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

- Depth-limited search will also be nonoptimal if we choose l<d. Its time complexity is $O(b^l)$ and its space complexity is O(bl).
- Depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.



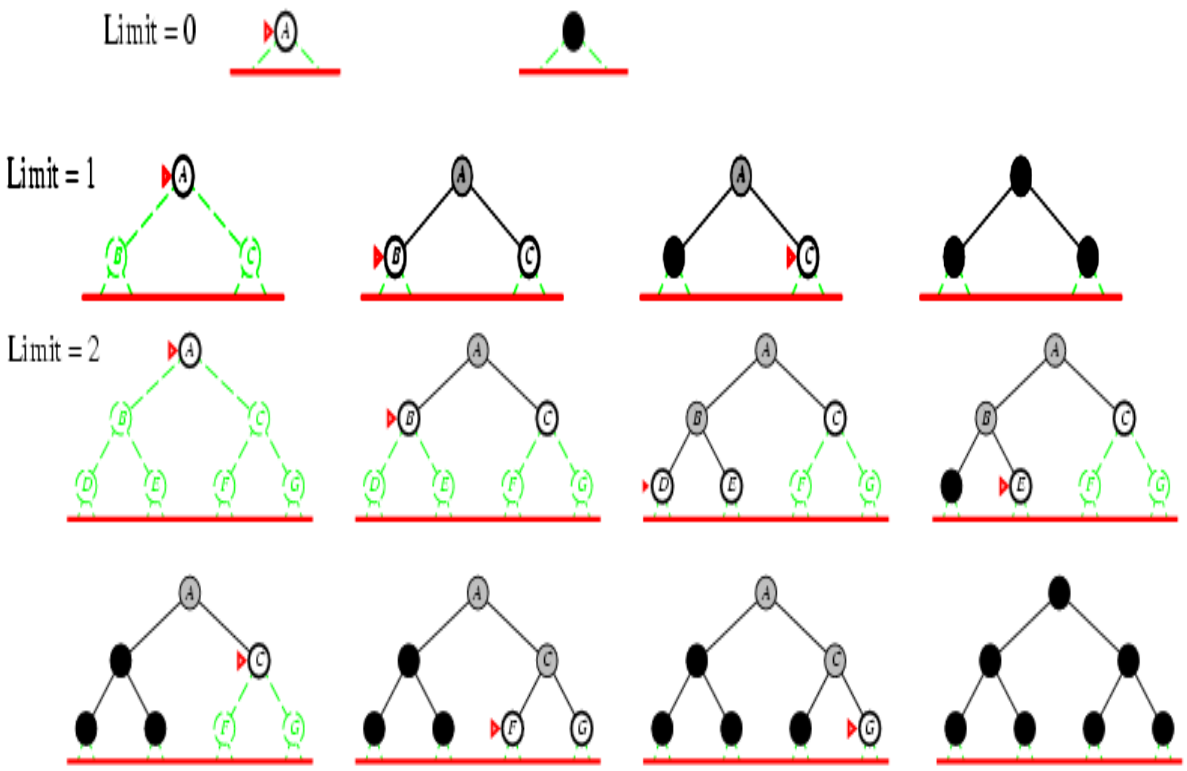
- Number of nodes generated in a depth-limited search to depth *d* with branching factor b: $NDLS = b_0 + b_1 + b_2 + \ldots + b_{d-2} + b_{d-1} + b_d$

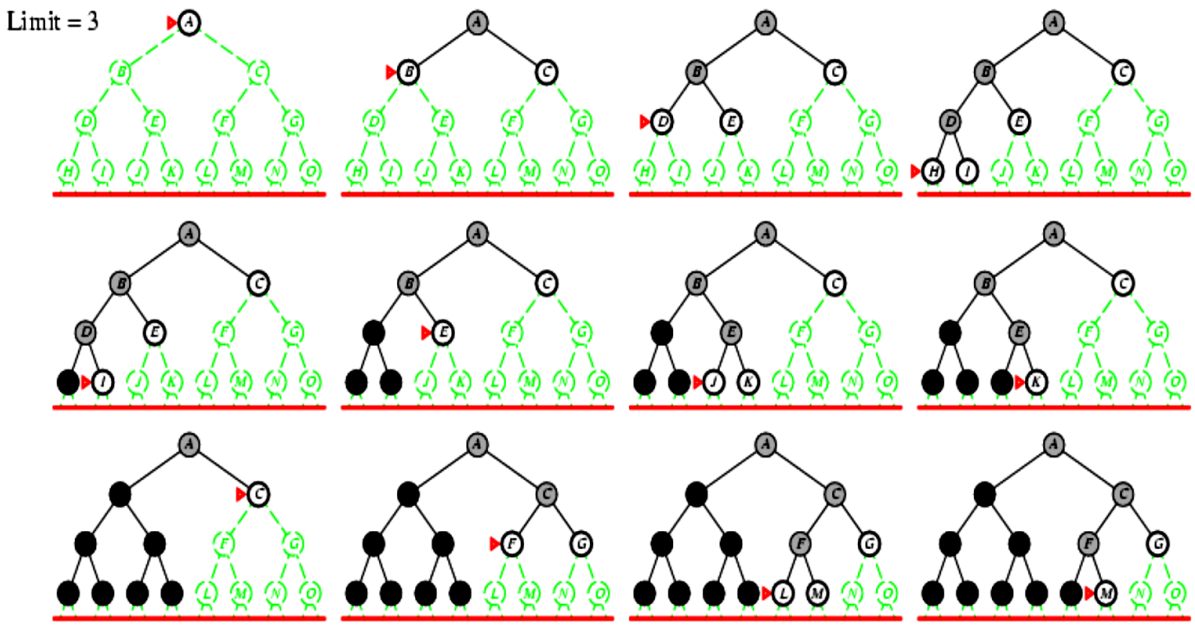
Iterative Deepening Search

Iterative deepening search is a strategy used in combination with DFS, that finds the best depth limit. It does this by gradually increasing the limit - first 0, then 1, then 2 and so on; until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. The algorithm is shown below:

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

- Iterative deepening combines the benefits of DFS and BFS. Like DFS, its memory requirements are very modest:$O(b_d)$.
- Like BFS, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.
- The following figure shows four iterations of ITERATIVE-DEEPENING SEARCH on a binary search tree, where the solution is found on the fourth iteration.





- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*: $N_{IDS} = (d+1)b_0 + d\ b^{\wedge}_1 + (d-1)b^{\wedge}_2 + \cdots + 3b_{d-2} + 2b_{d-1} + 1b_d$

Properties

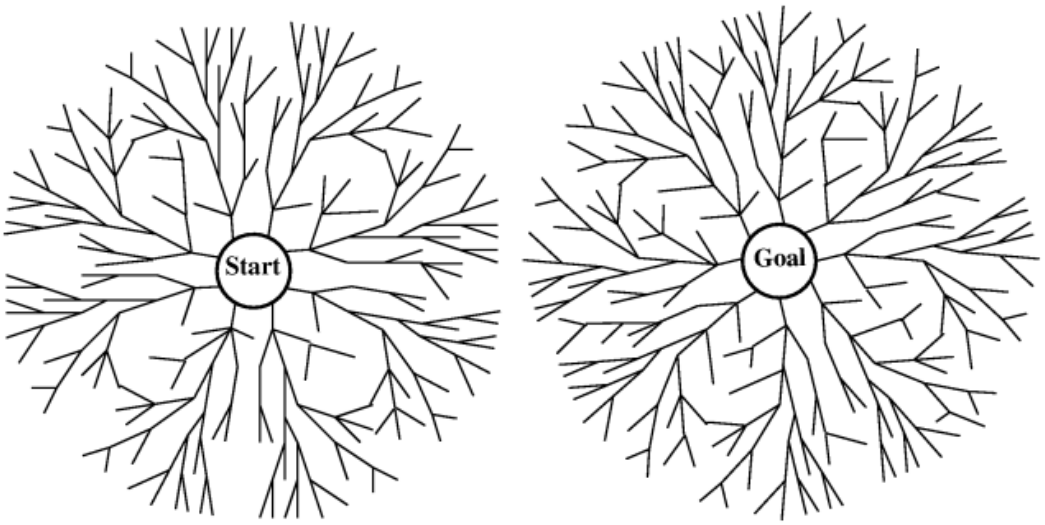
- **Complete**? Yes
- **Time**? $(d+1)b_0 + d\ b_1 + (d-1)b_2 + \ldots + b_d = O(b_d)$
- **Space**? $O(bd)$
- **Optimal**? Yes, if step cost = 1

Bidirectional Search

The idea behind bi-directional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. Checking a node for membership in the other search tree can be done in constant time with a hash table,

- so the time complexity of bi-directional search is $O(b_{d/2})$.
- Atleast one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is $O(b_{d/2})$ which is the weakness of the algorithm.
- The algorithm is complete and optimal if both searches are breadth-first;



INFORMED SEARCH ALGORITHMS

Informed search strategy is the one that uses problem-specific knowledge beyond the definition of the problem itself.

- Best-first search
- Greedy best-first search
- A* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search

Best-first search

Best first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function f(n).

- The node with the lowest evaluation is selected for expansion, because the evaluation
- measures distance to the goal. It can be implemented using a priority queue, a data structure that will maintain the fringe in ascending order of f – values.

Algorithm:

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove the first element from the queue, expand it and compute the estimated goal distances for each child. Place the children on the queue(at either end) and arrange all

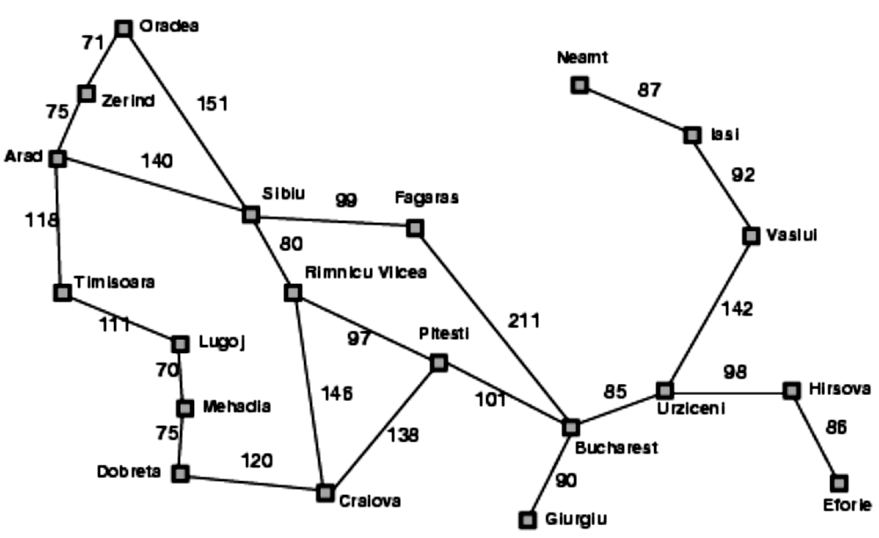
queue elements in ascending order corresponding to goal distance from the front of the queue.

5. Return to step 2. Best-first search uses different evaluation functions. A key component of these algorithms is a heuristic function, denoted h(n)

h(n)= estimated cost of the cheapest path from node n to a goal node.

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest which is shown below:

Romania with step costs in km

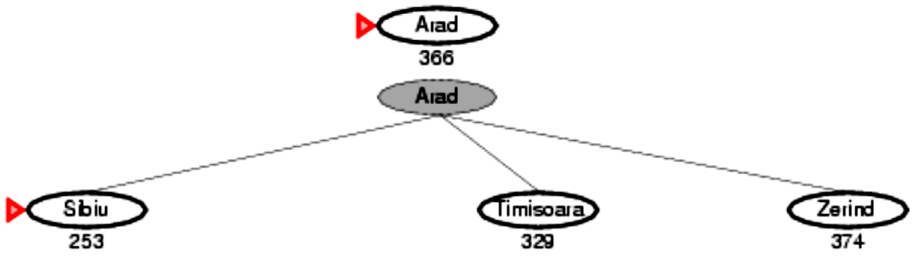


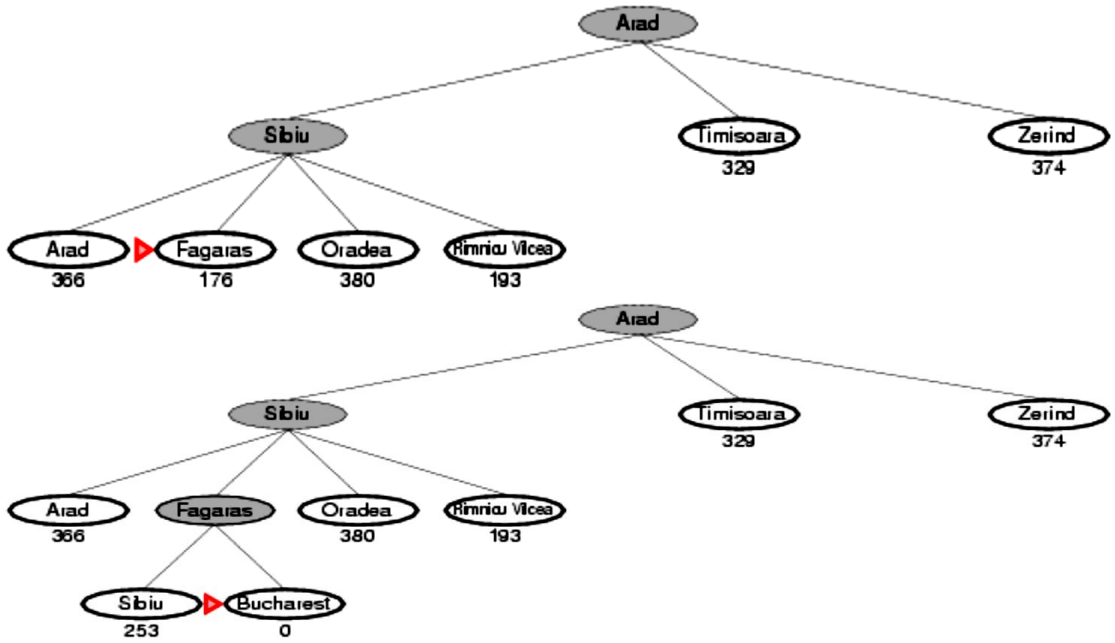
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
- = estimate of cost from *n* to *goal*
- e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

The progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest is shown in the following figure: .





The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal. Greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence its search cost is minimal.

Properties

- **Complete**? No – can get stuck in loops, e.g., Arad → Sibiu → Oradea → Arad →
- **Time**? $O(b^m)$, but a good heuristic can give dramatic improvement.
- **Space**? $O(b^m)$ -- keeps all nodes in memory
- **Optimal**? No

A* search:

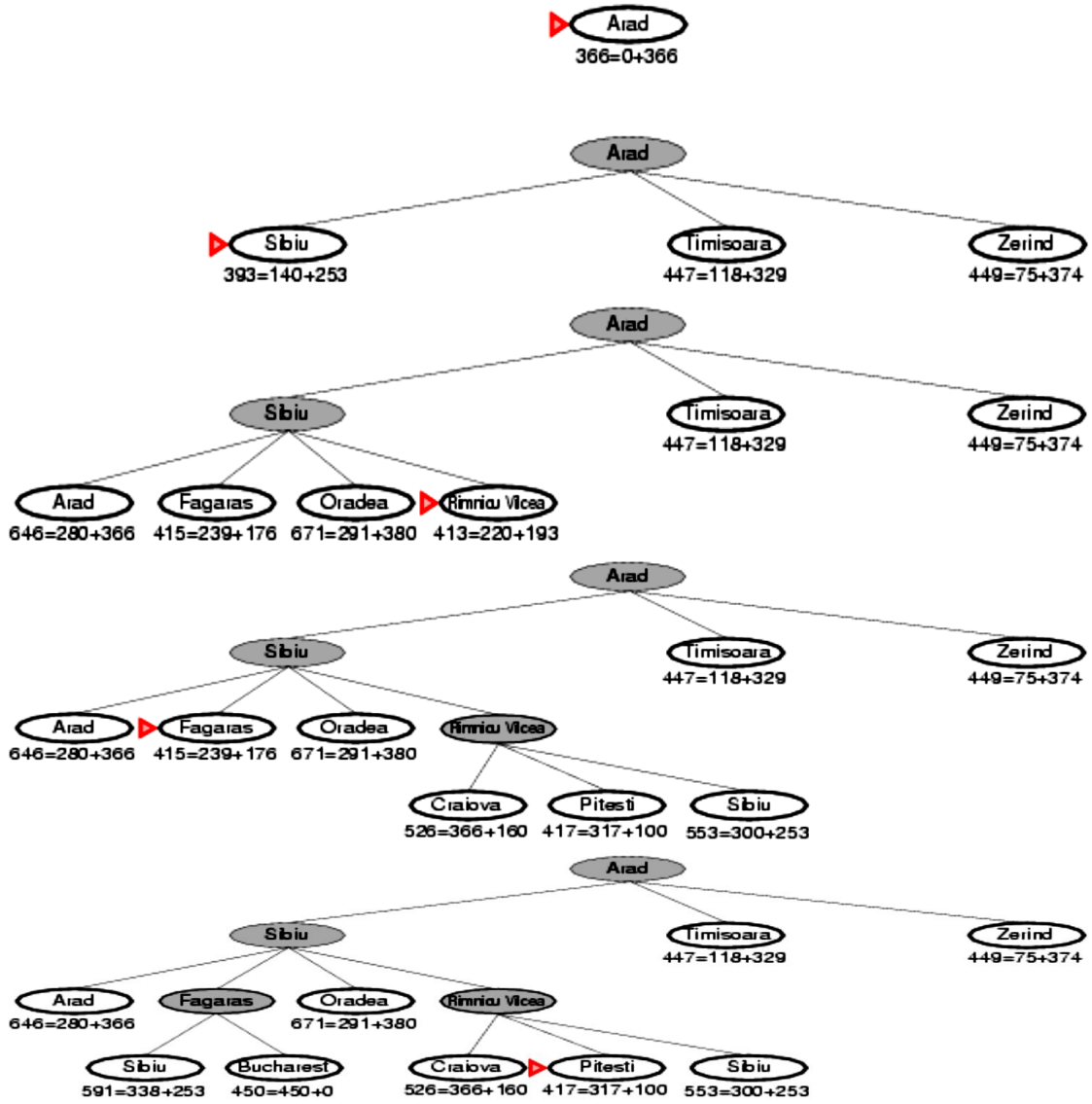
Minimizing the total estimated solution cost

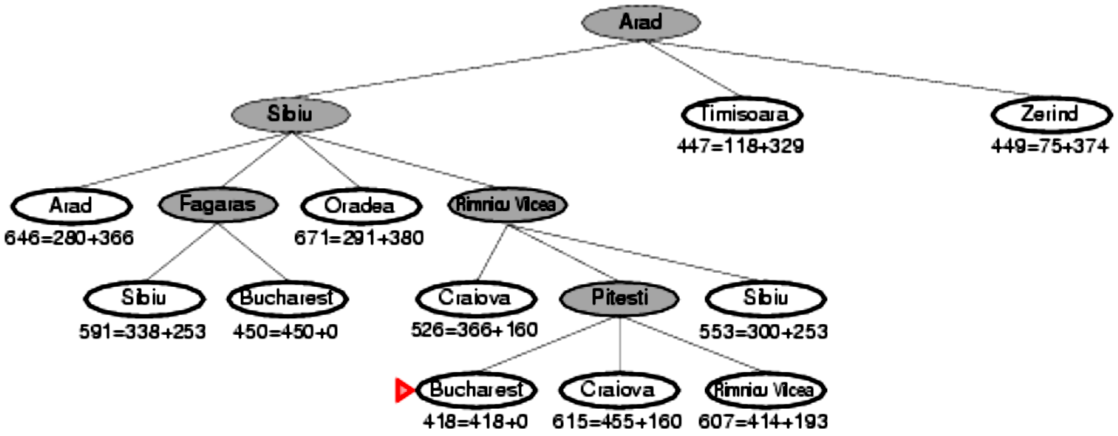
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal.

Algorithm:

1. Place the starting node s on open.
2. If open is empty, stop and return failure.
3. Remove from open the node n that has the smallest value of f*(n). If the node is a goal node, return success and stop. Otherwise,
4. Expand n, generating all of its successors n' and place n on closed. For every successor n', if n' is not already on open or closed attach a back-pointer to n, compute f*(n') and place it on open.
5. Each n' that is already on open or closed should be attached to back-pointers which reflect the lowest g*(n') path. If n' was on closed and its pointer was changed, remove it and place it on open.
6. Return to step 2.

The following figure shows an A* tree search for Bucharest.



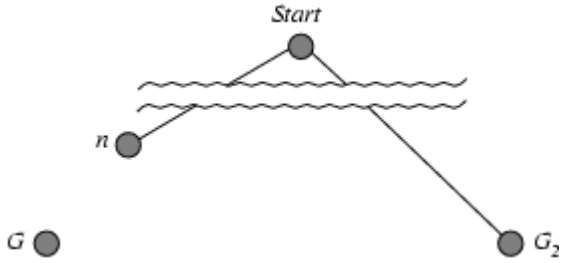


Admissible heuristics

- A heuristic *h(n)* is admissible if for every node *n*,
  h(n) <= $h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If *h(n)* is admissible, A* using TREE-SEARCH is optimal

Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe.
- Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.

Consistent heuristics

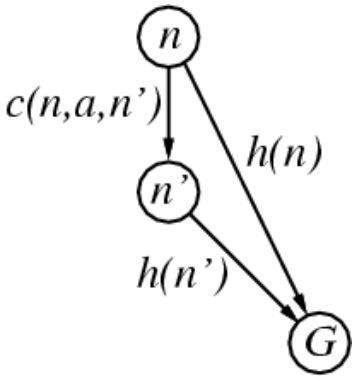
- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any
- action *a*,

$$h(n) \leq c(n,a,n') + h(n')$$

- If *h* is consistent, we have

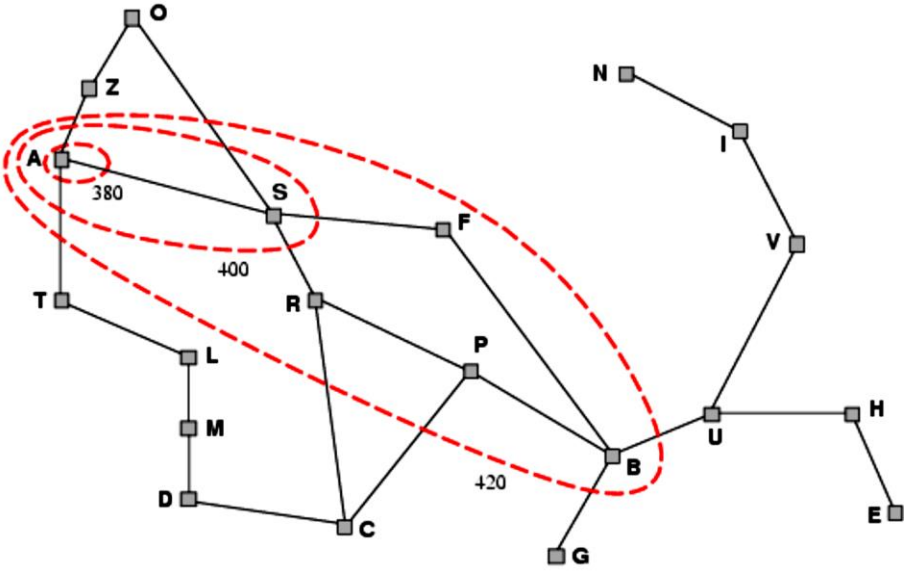
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

- i.e., *f(n)* is non-decreasing along any path.
- Theorem: If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal



Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



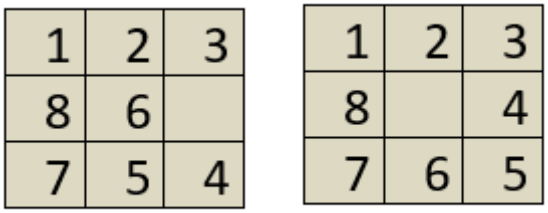
Properties

- **Complete**? Yes (unless there are infinitely many nodes with f <= *f(G)*)
- **Time**? Exponential
- **Space**? Keeps all nodes in memory
- **Optimal**? Yes

HEURISTIC FUNCTION

- *A good heuristic function is determined by its efficiency. More is the information about the problem, more is the processing time.*
- Some toy problems, such as 8-puzzle, 8-queen, tic-tac-toe, etc., can be solved more efficiently with the help of a heuristic function.
- Consider the following 8-puzzle problem where we have a start state and a goal state.
- Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state.
- There can be four moves either **left, right, up, or down**. There can be several ways to convert the current/start state to the goal state, but, we can use a heuristic function h(n) to solve the problem more efficiently.

E.g., for the **8-puzzle:**



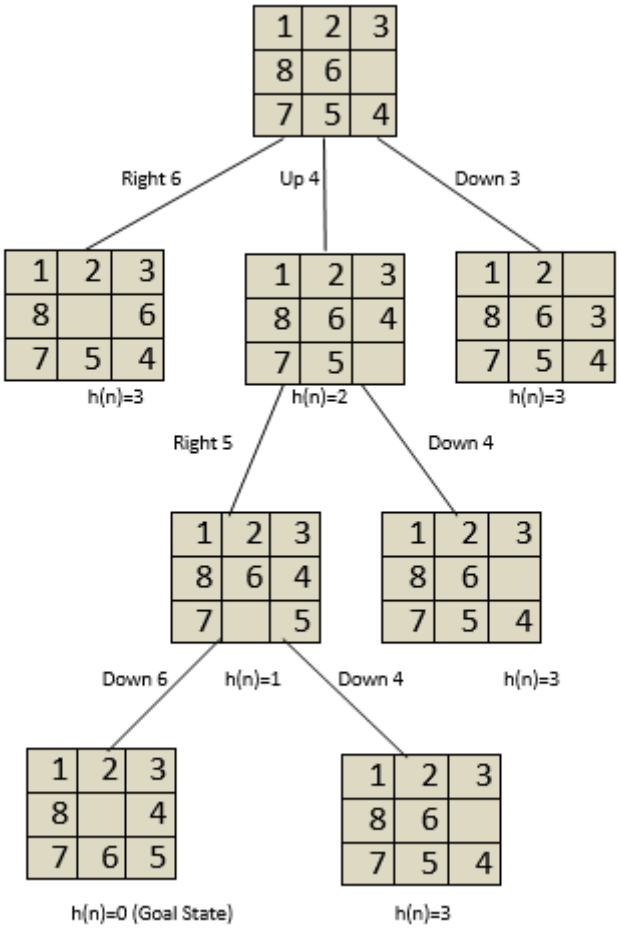
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions. This is sometimes called the city block distance or Manhattan distance

(i.e., no. of squares from desired location of each tile)

- $h_1(S)$ = ? 3, 3 tiles are out of position, so the start state would have $h_1=3$.

h_1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

- $h_2(S)$ = ? 0+0+0+1+1+1+0+0 = 3 . h_2 is also admissible, because all any move can do is move one tile one step closer to the goal.



A heuristic function for the 8-puzzle problem is defined below:

h(n)=Number of tiles out of position.

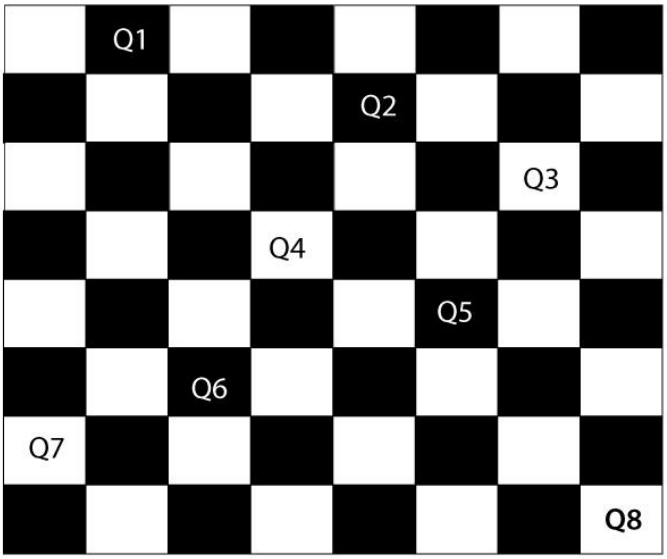
- So, there is total of three tiles out of position i.e., 6,5 and 4. Do not count the empty tile present in the goal state). i.e. h(n)=3.
- Now, we require to minimize the value of **h(n) =0.**
- We can construct a state-space tree to minimize the h(n) value to 0, as shown below:
- It is seen from the above state space tree that the goal state is minimized from h(n)=3 to h(n)=0.
- However, we can create and use several heuristic functions as per the reqirement.

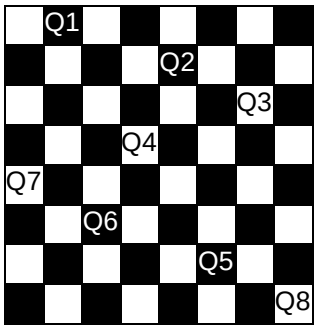
- It is also clear from the above example that a heuristic function h(n) can be defined as the information required to solve a given problem more efficiently.
- The information can be related to the **nature of the state, cost of transforming from one state to another, goal node characterstics,** etc., which is expressed as a heuristic function.

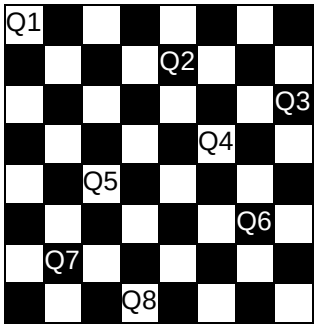
8-queens problem

It The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.**

From the following figure, we can understand the problem as well as its correct solution







.

It is noticed from the above figure that each queen is set into the chessboard in a position where no other queen is placed diagonally, in same row or column. Therefore, it is one right approach to the 8-queens problem.
For this problem, there are two main kinds of formulation:

Incremental formulation: It starts from an empty state where the operator augments a queen at each step.

- **States:** Arrangement of any 0 to 8 queens on the chessboard.
- **Initial State:** An empty chessboard
- **Actions:** Add a queen to any empty box.
- **Transition model:** Returns the chessboard with the queen added in a box.
- **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.
- **Path cost:** There is no need for path cost because only final states are counted.

In this formulation, there is approximately **1.8×10^{14}** possible sequence to investigate.

Complete-state formulation: It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.

This formulation is better than the incremental formulation as it reduces the state space from **1.8×10^{14}** to **2057**, and it is easy to find the solutions.
Some Real-world problems.

Properties

- **Admissible Condition:** An algorithm is said to be admissible, if it returns an optimal solution.
- **Completeness:** An algorithm is said to be complete, if it terminates with a solution (if the solution exists).
- **Dominance Property:** If there are two admissible heuristic algorithms **A1** and **A2** having **h1** and **h2** heuristic functions, then **A1** is said to dominate **A2** if **h1** is better than **h2** for all the values of node **n.**
- **Optimality Property:** If an algorithm is **complete, admissible**, and **dominating** other algorithms, it will be the best one and will definitely give an optimal solution.

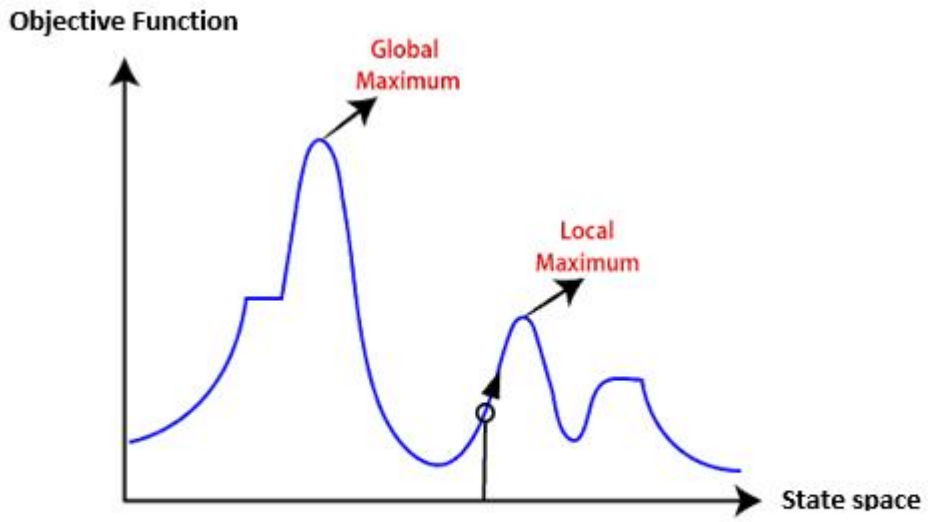
LOCAL SEARCH ALGORITHMS

local search algorithms" where the path cost does not matters, and only focus on solution-state needed to reach the goal node.

A local search algorithm completes its task by traversing on a single current node rather than multiple paths and following the neighbors of that node generally.

.Let's understand the working of a local search algorithm with the help of an example: Consider the below state-space landscape having both:

- **Location:** It is defined by the state.
- **Elevation:** It is defined by the value of the objective function or heuristic cost function.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

The LSA explores the above landscape by finding the following two points:

- **Global Minimum:** If the elevation corresponds to the cost, then the task is to find the lowest valley, which is known as **Global Minimum.**
- **Global Maxima:** If the elevation corresponds to an objective function, then it finds the highest peak which is called as **Global Maxima**. It is the highest point in the valley.

We will understand the working of these points better in Hill-climbing search.

Below are some different types of local searches:

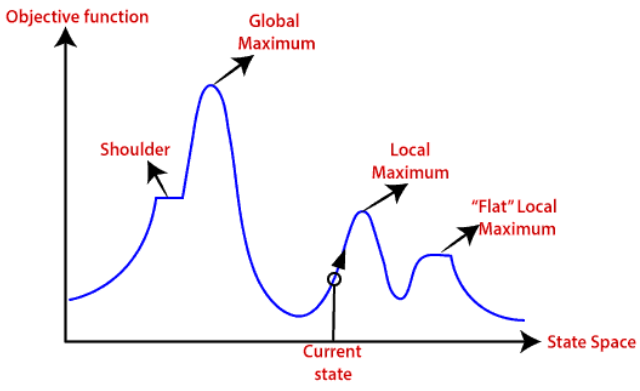
- Hill-climbing Search
- Simulated Annealing
- Local Beam Search

Hill Climbing Algorithm:

Hill climbing search is a local search problem. The purpose of the hill climbing search is to climb a hill and reach the topmost peak/ point of that hill. It is based on the **heuristic search technique** where the person who is climbing up on the hill estimates the direction which will lead him to the highest peak.

To understand the concept of hill climbing algorithm, consider the below landscape representing the **goal state/peak** and the **current state** of the climber. The topographical regions shown in the figure can be defined as:

- **Global Maximum:** It is the highest point on the hill, which is the goal state.
- **Local Maximum:** It is the peak higher than all other peaks but lower than the global maximum.
- **Flat local maximum:** It is the flat area over the hill where it has no uphill or downhill. It is a saturated point of the hill.
- **Shoulder:** It is also a flat area where the summit is possible.
- **Current state:** It is the current position of the person.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

Types of Hill climbing search algorithm

There are following types of hill-climbing search:

- Simple hill climbing
- Steepest-ascent hill climbing
- Stochastic hill climbing

Simple hill climbing search

Simple hill climbing is the simplest technique to climb a hill. The task is to reach the highest peak of the mountain. Here, the movement of the climber depends on his move/steps. If he finds his next step better than the previous one, he continues to move else remain in the same state. This search focus only on his previous and next step.

1. Create a **CURRENT** node, **NEIGHBOUR** node, and a **GOAL** node.
2. If the **CURRENT node=GOAL node**, return **GOAL** and terminate the search.
3. Else **CURRENT node<= NEIGHBOUR node,** move ahead.
4. Loop until the goal is not reached or a point is not found.

Steepest-ascent hill climbing

Steepest-ascent hill climbing is different from simple hill climbing search. Unlike simple hill climbing search, It considers all the successive nodes, compares them, and choose the node which is closest to the solution. Steepest hill climbing search is similar to **best-first search** because it focuses on each node instead of one.

Note: Both simple, as well as steepest-ascent hill climbing search, fails when there is no closer node.

1. Create a **CURRENT** node and a **GOAL** node.
2. If the **CURRENT node=GOAL** node, return **GOAL** and terminate the search.
3. Loop until a better node is not found to reach the solution.
4. If there is any better successor node present, expand it.
5. When the **GOAL** is attained, return **GOAL** and terminate.

Stochastic hill climbing

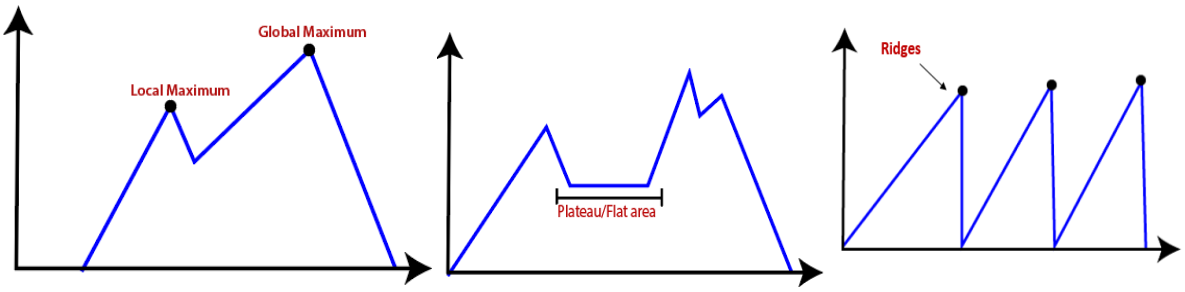
Stochastic hill climbing does not focus on all the nodes. It selects one node at random and decides whether it should be expanded or search for a better one.

Advantages

- Local search algorithms use a very little or constant amount of memory as they operate only on a single path.
- Most often, they find a reasonable solution in large or infinite state spaces where the classical or systematic algorithms do not work.

Drawbacks

- **Local Maxima:** It is that peak of the mountain which is highest than all its neighboring states but lower than the global maxima. It is not the goal peak because there is another peak higher than it.



- **Plateau:** It is a flat surface area where no uphill exists. It becomes difficult for the climber to decide that in which direction he should move to reach the goal point. Sometimes, the person gets lost in the flat area.

- **Ridges:** It is a challenging problem where the person finds two or more local maxima of the same height commonly. It becomes difficult for the person to navigate the right point and stuck to that point itself.

Simulated Annealing

Simulated annealing is similar to the hill climbing algorithm. It works on the current situation. It picks **a random move** instead of picking **the best move**. If the move leads to the improvement of the current situation, it is always accepted as a step

towards the solution state, else it accepts the move having **a probability less than 1**. It is also applied for factory scheduling and other large optimization tasks.

Local Beam Search

Local beam search is quite different from random-restart search. It keeps track of **k** states instead of just one. It selects **k** randomly generated states, and expand them at each step. If any state is a goal state, the search stops with success. Else it selects the best **k** successors from the complete list and repeats the same process. In random-restart search where each search process runs independently, but in local beam search, the necessary information is shared between the parallel search processes.

OPTIMIZATION PROBLEMS

Genetic Algorithms

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. **They are commonly used to generate high-quality solutions for optimization problems and search problems.**

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. **Each generation consist of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

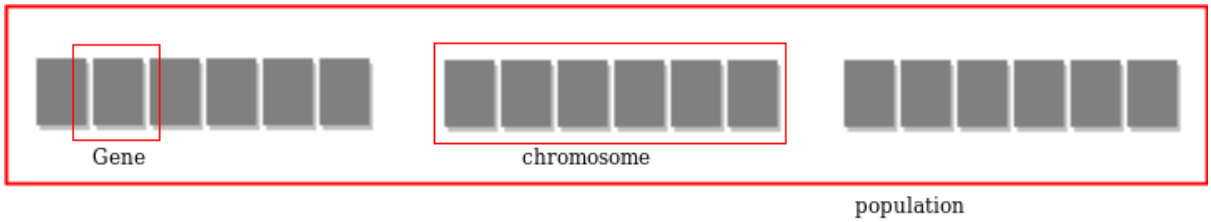
Foundation of Genetic Algorithms

Genetic algorithms are based on an analogy with genetic structure and behaviour of chromosomes of the population. Following is the foundation of GAs based on this analogy –

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from “fittest” parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

Search space

The population of individuals are maintained within search space. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).



Fitness Score

A Fitness Score is given to each individual which **shows the ability of an individual to “compete”**. The individual having optimal fitness score (or near optimal) are sought.

The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce **better offspring** by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

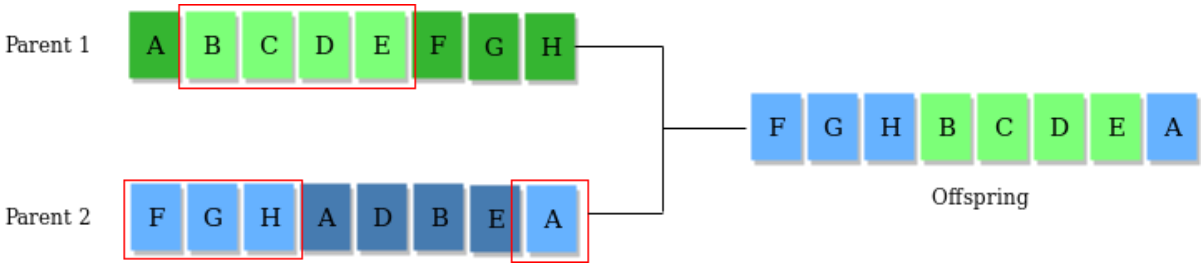
Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better **"partial solutions"** than previous generations. Once the offspring produced having no significant difference from offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.

Operators of Genetic Algorithms

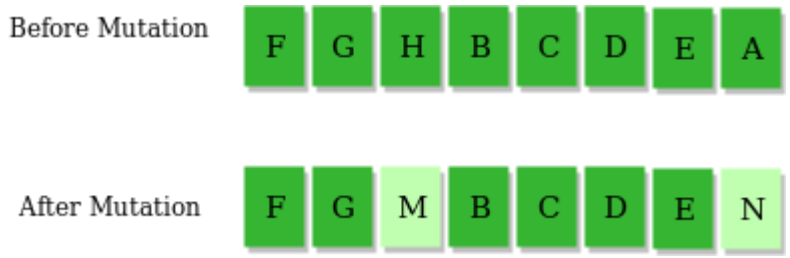
Once the initial generation is created, the algorithm evolves the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).



3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence



The whole algorithm can be summarized as –

1) Randomly initialize populations p

2) Determine fitness of population

3) Until convergence repeat:

a) Select parents from population

b) Crossover and generate new population

c) Perform mutation on new population

d) Calculate fitness for new population

Example problem and solution using Genetic Algorithms

Given a target string, the goal is to produce target string starting from a random string of the same length. In the following implementation, following analogies are made.

- Characters A-Z, a-z, 0-9, and other special symbols are considered as genes.
- A string generated by these characters is considered as chromosome/solution/Individual.

Fitness score is the number of characters which differ from characters in target string at a particular index. So individual having lower fitness value is given more preference.

Adversarial Search

Competitive environments, in which the agents' goals are in conflict, give rise to adversarial search problems- often known as games. In AI, "games" are usually of a rather specialized kind – in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite.

A game can be formally defined as a kind of search problem with the following components:

• The **initial state**, which includes the board position and identifies the player to move.

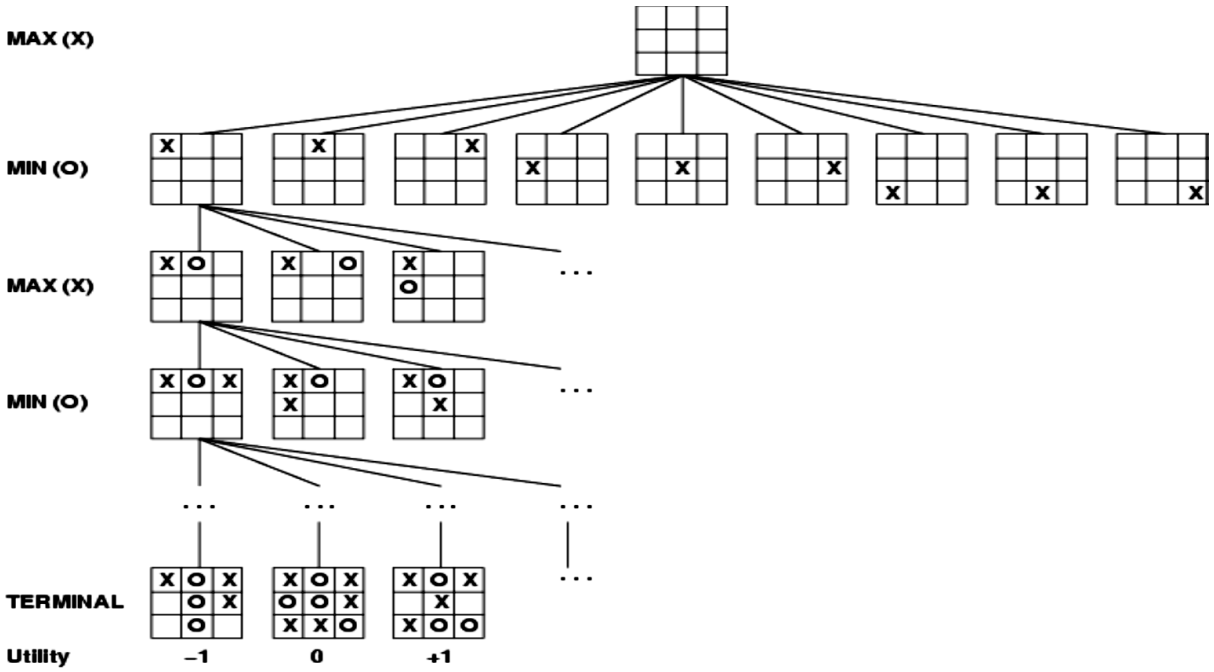
•A **successor function**, which returns a list of (move, state)pairs, each indicating a legal move and the resulting state.

•A **terminal test**, which determines when the game is over. States where the game has ended are called terminal states.

•A **utility function**, which gives a numeric value for the terminal states.

The initial state and the legal moves for each side define the **game tree** for the game. The following figure shows part of the game tree for tic-tac-toe. From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled.

Game tree (2-player, deterministic, turns)



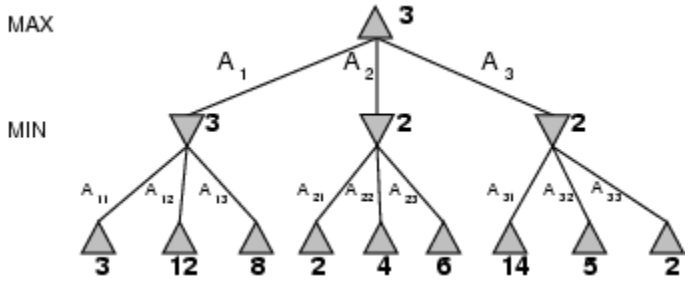
MINIMAX

Given a game tree, the optimal strategy can be determined by examining the minimax value of each node, which we write as MINIMAX-VALUE(n). The minimax value of a node is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game.

•Perfect play for deterministic games

•Idea: choose move to position with highest minimax value = best achievable payoff against best play

E.g., 2-ply game:




```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Properties

• **Complete**? Yes (if tree is finite)

• **Optimal**? Yes (against an optimal opponent)

• **Time complexity**? $O(b_m)$:the maximum depth of the tree is m, and there are b legal moves at each point

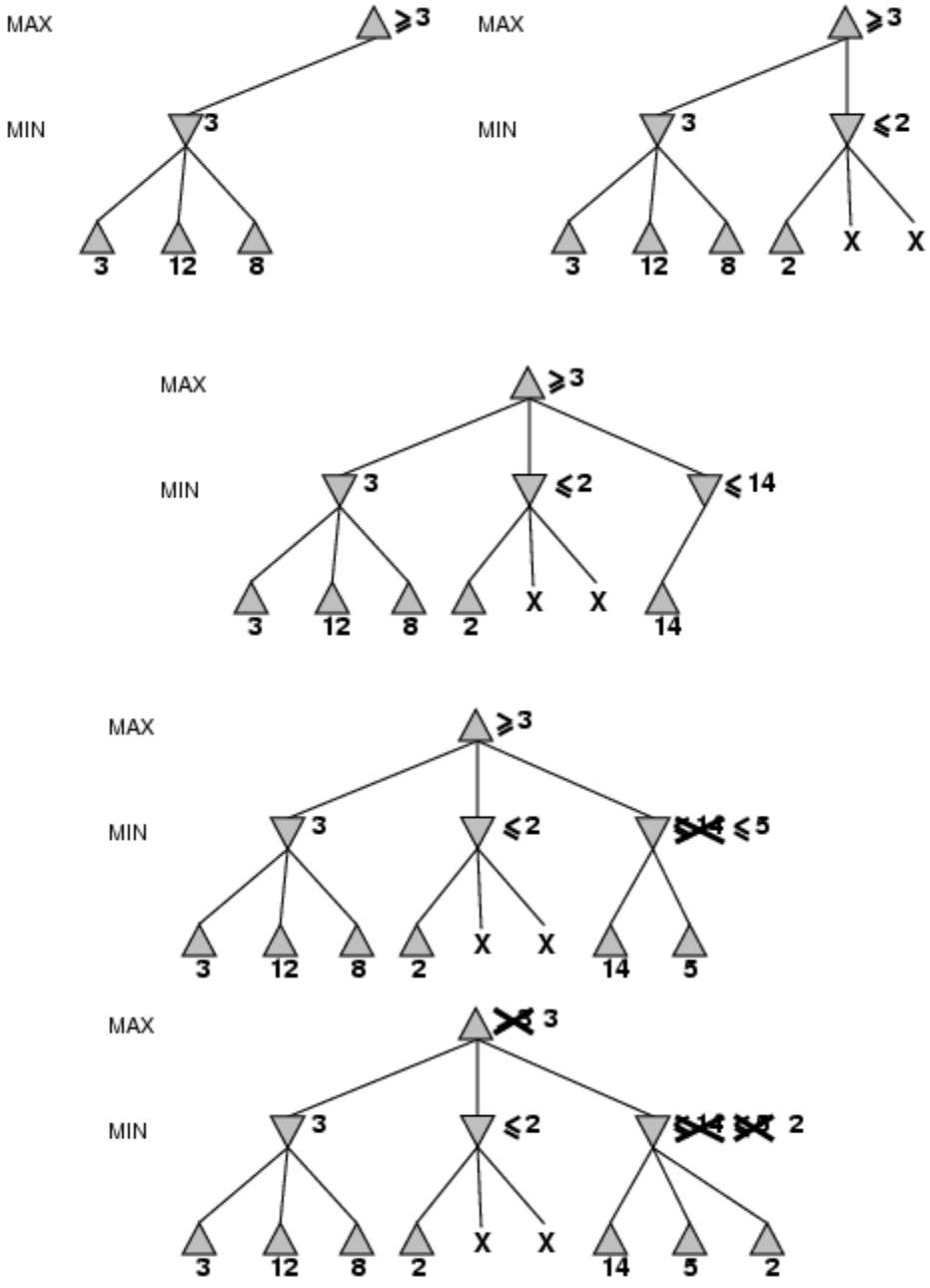
• **Space complexity**? O(bm) (depth-first •With "perfect ordering," time complexity = $O(b_{m/2})$

ALPHA-BETA PRUNING

The problem with minimax procedure is that the number of game states it has to examine is exponential in the number of moves. We can cut it in half using the technique called alpha-beta pruning. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider again the two-ply game tree. The steps are explained in the following figure. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

α-β pruning example



The value of the root node is given by

MINIMAX-VALUE(root) = max(min(3,12,8), min(2,x,y), min(14,5,2))

= max(3,min(2,x,y),2)

=max(3,z,2) where z<=2

=3

x and y: two unevaluated successors

z: minimum of x and y

Properties

•Pruning does not affect final result

•Good move ordering improves effectiveness of pruning

•A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

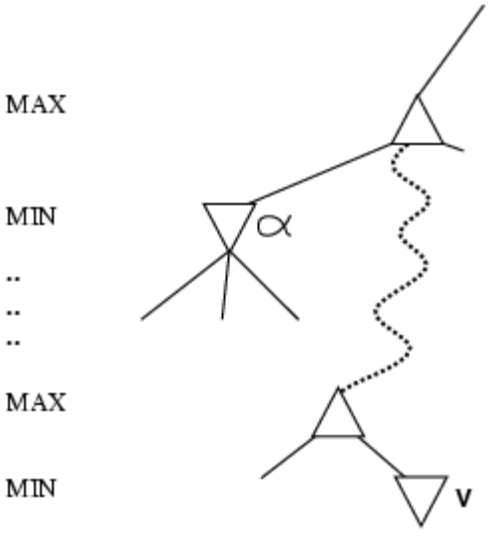
Why is it called α-β?

•α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *max*

•β is the value of the best (i.e., lowest-value) choice found so far at any choice point along the path for *min*

•If *v* is worse than α, *max* will avoid it

- prune that branch




```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- branches at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN respectively.
- The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined.

UNIT II

PROBABILISTIC REASONING

ACTING UNDER UNCERTAINTY

Uncertainty

Let action At = leave for airport t minutes before flight

Will At get me there on time?

Problems:

1) partial observability (road state, other drivers' plans, etc.)

2) noisy sensors (KCBS track reports)

3) uncertainty in action outcomes (at tire, etc.)

4) immense complexity of modelling and predicting track

Hence a purely logical approach either

1) risks falsehood: A25 will get me there on time"

or 2) leads to conclusions that are too weak for decision making:

“A25 will get me there on time if there's no accident on the bridge and it doesn't rain and my tires remain intact etc."

Methods for handling uncertainty

Default or nonmonotonic logic:

Assume my car does not have a at tire

Assume A25 works unless contradicted by evidence

Issues: What assumptions are reasonable? How to handle contradiction?

Rules with fudge factors:

$$A_{25} \mapsto_{0.3} AtAirportOnTime$$
$$Sprinkler \mapsto_{0.99} WetGrass$$
$$WetGrass \mapsto_{0.7} Rain$$

Issues: Problems with combination, e.g., Sprinkler causes Rain??

Probability

Given the available evidence, A25 will get me there on time with probability 0:04

(Fuzzy logic handles degree of truth NOT uncertainty e.g., WetGrass is true to degree 0:2)

Probabilistic assertions summarize effects of

laziness: failure to enumerate exceptions, qualifications, etc.

ignorance: lack of relevant facts, initial conditions, etc.

Subjective or Bayesian probability:

Probabilities relate propositions to one's own state of knowledge

e.g., P(A25/no reported accidents) = 0:06

These are not claims of a \probabilistic tendency" in the current situation (but might be learned from past experience of similar situations)

Probabilities of propositions change with new evidence:

e.g., P(A25jno reported accidents; 5 a.m.) = 0:15

(Analogous to logical entailment status KB j= _, not truth.)

Making decisions under uncertainty

Suppose I believe the following:

P(A25 gets me there on timej : : :) = 0:04

P(A90 gets me there on timej : : :) = 0:70

P(A120 gets me there on timej : : :) = 0:95

P(A1440 gets me there on timej : : :) = 0:9999

Which action to choose?

Depends on my preferences for missing ight vs. airport cuisine, etc.

Utility theory is used to represent and infer preferences

Decision theory = utility theory + probability theory

Probability basics

A probability space or probability model is a sample space with an assignment $P(\omega)$ for every $\omega \in \Omega$ s.t.

$0 \leq P(\omega) \leq 1$

$\Sigma_\omega P(\omega) = 1$

e.g., $P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6.$

An event A is any subset of Ω

$P(A) = \Sigma_{\{\omega \in A\}} P(\omega)$

E.g., $P(\text{die roll} < 4) = P(1) + P(2) + P(3) = 1/6 + 1/6 + 1/6 = 1/2$

Begin with a set W - the sample space

e.g., 6 possible rolls of a die.

w Wis a sample point/possible world/atomic event

Random variables

A random variable is a function from sample points to some range, e.g., the reals or Booleans

e.g., Odd(1)=true.

P induces a probability distribution for any r.v. X:

$$P(X = x_i) = \Sigma_{\{\omega: X(\omega) = x_i\}} P(\omega)$$

e.g., $P(Odd = true) = P(1) + P(3) + P(5) = 1/6 + 1/6 + 1/6 = 1/2$

Propositions

Think of a proposition as the event (set of sample points) where the proposition is true

Given Boolean random variables A and B:

event a = set of sample points where $A(\omega) = true$

event $\neg a$ = set of sample points where $A(\omega) = false$

event $a \wedge b$ = points where $A(\omega) = true$ and $B(\omega) = true$

Often in AI applications, the sample points are defined by the values of a set of random variables, i.e., the sample space is the Cartesian product of the ranges of the variables With Boolean variables, sample point = propositional logic model

e.g., $A = true$, $B = false$, or $a \wedge \neg b$.

Proposition = disjunction of atomic events in which it is true

e.g., $(a \vee b) \equiv (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$

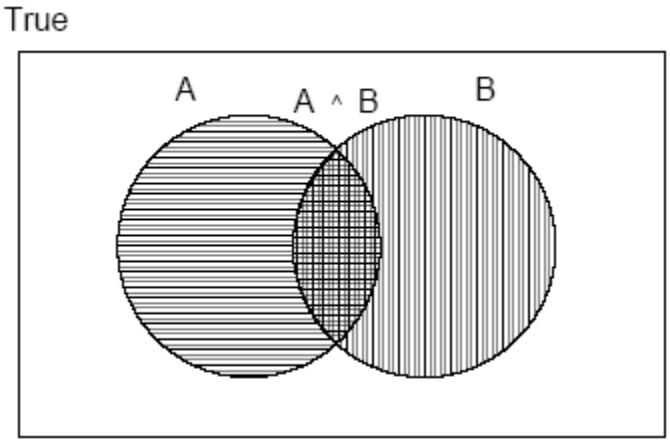
$\Rightarrow P(a \vee b) = P(\neg a \wedge b) + P(a \wedge \neg b) + P(a \wedge b)$

Why use probability?

The definitions imply that certain logically related events must have related probabilities

E.g., $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$

De Finetti (1931): an agent who bets according to probabilities that violate these axioms can be forced to bet so as to lose money regardless of outcome.



Syntax for propositions

Propositional or Boolean random variables

e.g., Cavity (do I have a cavity?)

Cavity =true is a proposition, also written cavity

Discrete random variables (finite or infinite)

e.g., Weather is one of á(sunny, rain, cloudy, snow ñ

Weather =rain is a proposition

Values must be exhaustive and mutually exclusive

Continuous random variables (bounded or unbounded)

e.g., Temp=21:6; also allow, e.g., Temp < 22:0.

Arbitrary Boolean combinations of basic propositions

Prior probability

Prior or unconditional probabilities of propositions

e.g., P(Cavity =true) = 0:1 and P(Weather =sunny) = 0:72

correspond to belief prior to arrival of any (new) evidence

Probability distribution gives values for all possible assignments:

P(Weather) = á0:72; 0:1, 0:08, 0:1ñ (normalized, i.e., sums to 1)

Joint probability distribution for a set of r.v.s gives the

probability of every atomic event on those r.v.s (i.e., every sample point)

P(Weather,Cavity) = a 4 X 2 matrix of values:

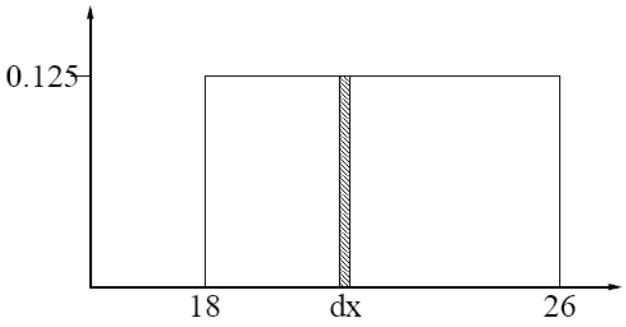
$Weather =$	$sunny$	$rain$	$cloudy$	$snow$
$Cavity = true$	0.144	0.02	0.016	0.02
$Cavity = false$	0.576	0.08	0.064	0.08

Every question about a domain can be answered by the joint distribution because every event is a sum of sample points

Probability for continuous variables

Express distribution as a parameterized function of value:

P(X =x) = U[18; 26](x) = uniform density between 18 and 26



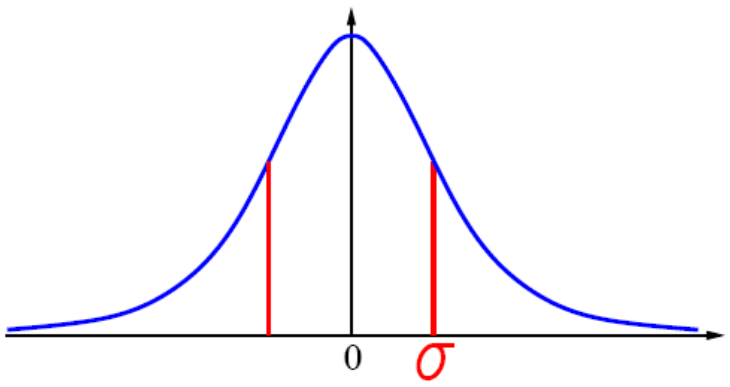
Here P is a density; integrates to 1.

P(X =20:5) = 0:125 really means

$$\lim_{dx \to 0} P(20.5 \leq X \leq 20.5 + dx)/dx = 0.125$$

Gaussian density

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



Conditional probability

Conditional or posterior probabilities

e.g., P(cavity/toothache) = 0:8

i.e., given that toothache is all I know

NOT "if toothache then 80% chance of cavity"

P(Cavity/Toothache) = 2-element vector of 2-element vectors)

If we know more, e.g., cavity is also given, then we have

P(cavity/toothache, cavity) = 1

Note: the less specific belief remains valid after more evidence arrives,

but is not always useful

New evidence may be irrelevant, allowing simplification,

e.g., P(cavity/toothache; 49ersWin) = P(cavity/toothache) = 0:8

This kind of inference, sanctioned by domain knowledge, is crucial

Definition of conditional probability:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \text{ if } P(b) \neq 0$$

Product rule gives an alternative formulation:

P(a ^ b) = P(a/b)P(b) = P(b/a)P(a)

A general version holds for whole distributions, e.g.,

P(Weather,Cavity) = P(Weather/Cavity)P(Cavity)

(View as a 4 _ 2 set of equations, not matrix mult.)

Chain rule is derived by successive application of product rule:

$$\begin{aligned}\mathbf{P}(X_1, \ldots, X_n) &= \mathbf{P}(X_1, \ldots, X_{n-1})\ \mathbf{P}(X_n|X_1, \ldots, X_{n-1}) \\ &= \mathbf{P}(X_1, \ldots, X_{n-2})\ \mathbf{P}(X_{n_1}|X_1, \ldots, X_{n-2})\ \mathbf{P}(X_n|X_1, \ldots, X_{n-1}) \\ &= \ldots \\ &= \prod_{i=1}^{n} \mathbf{P}(X_i|X_1, \ldots, X_{i-1})\end{aligned}$$

Bayesian Inference by enumeration

Start with the joint distribution:

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

For any proposition Æ, sum the atomic events where it is true:

$$P(\phi) = \sum_{\omega:\omega \models \phi} P(\omega)$$

Start with the joint distribution:

	toothache		¬ toothache	
	catch	¬ catch	catch	¬ catch
cavity	.108	.012	.072	.008
¬ cavity	.016	.064	.144	.576

For any proposition Æ, sum the atomic events where it is true:

$$P(\phi) = \Sigma_{\omega:\omega \models \phi} P(\omega)$$

$$P(toothache) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2$$

Start with the joint distribution:

	toothache		¬ toothache	
	catch	¬ catch	catch	¬ catch
cavity	.108	.012	.072	.008
¬ cavity	.016	.064	.144	.576

For any proposition Æ, sum the atomic events where it is true:

$$P(\phi) = \Sigma_{\omega:\omega \models \phi} P(\omega)$$

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$$

Start with the joint distribution:

	toothache		¬ toothache	
	catch	¬ catch	catch	¬ catch
cavity	.108	.012	.072	.008
¬ cavity	.016	.064	.144	.576

Can also compute conditional probabilities:

$$P(\neg cavity|toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$

Normalization

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

Denominator can be viewed as a normalization constant α

$$\mathbf{P}(Cavity|toothache) = \alpha\,\mathbf{P}(Cavity, toothache)$$
$$= \alpha\,[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)]$$
$$= \alpha\,[\langle 0.108, 0.016\rangle + \langle 0.012, 0.064\rangle]$$
$$= \alpha\,\langle 0.12, 0.08\rangle = \langle 0.6, 0.4\rangle$$

General idea: compute distribution on query variable

by fixing evidence variables and summing over hidden variables

Let X be all the variables. Typically, we want the posterior joint distribution of the query variables Y given specific values e for the evidence variables E

Let the hidden variables be H = X - Y - E

Then the required summation of joint entries is done by summing out the hidden variables:

$$\mathbf{P(Y|E = e)} = \alpha\mathbf{P(Y, E = e)} = \alpha\Sigma_{\mathbf{h}}\mathbf{P(Y, E = e, H = h)}$$

The terms in the summation are joint entries because Y, E, and H together exhaust the set of random variables

Obvious problems:

1) Worst-case time complexity $O(d^n)$ where d is the largest arity

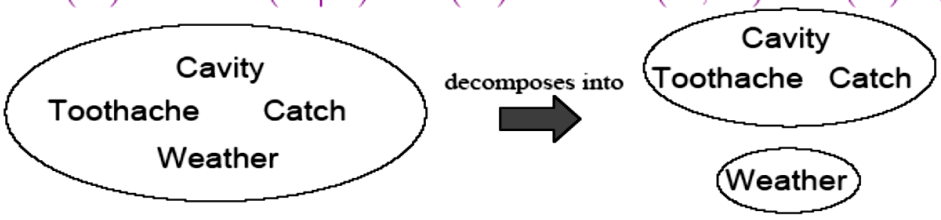
2) Space complexity $O(d^n)$ to store the joint distribution

3) How to find the numbers for $O(d^n)$ entries???

Independence

A and B are independent iff

$$\mathbf{P}(A|B) = \mathbf{P}(A) \quad \text{or} \quad \mathbf{P}(B|A) = \mathbf{P}(B) \quad \text{or} \quad \mathbf{P}(A, B) = \mathbf{P}(A)\mathbf{P}(B)$$



$$\mathbf{P}(Toothache, Catch, Cavity, Weather) = \mathbf{P}(Toothache, Catch, Cavity)\mathbf{P}(Weather)$$

32 entries reduced to 12; for n independent biased coins, 2n ® n

Absolute independence powerful but rare

Dentistry is a large field with hundreds of variables,

none of which are independent. What to do?

Conditional independence

$\mathbf{P}(Toothache, Cavity, Catch)$ has $2^3 - 1 = 7$ independent entries

If I have a cavity, the probability that the probe catches in it doesn't depend on whether I have a toothache:

(1) $P(catch|toothache, cavity) = P(catch|cavity)$

The same independence holds if I haven't got a cavity:

(2) $P(catch|toothache, \neg cavity) = P(catch|\neg cavity)$

$Catch$ is conditionally independent of $Toothache$ given $Cavity$:

$\mathbf{P}(Catch|Toothache, Cavity) = \mathbf{P}(Catch|Cavity)$

Equivalent statements:

$\mathbf{P}(Toothache|Catch, Cavity) = \mathbf{P}(Toothache|Cavity)$

$\mathbf{P}(Toothache, Catch|Cavity) = \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)$

Write out full joint distribution using chain rule:

$$\begin{aligned}
&\mathbf{P}(Toothache, Catch, Cavity) \\
&= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch, Cavity) \\
&= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity) \\
&= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)
\end{aligned}$$

I.e., 2 + 2 + 1 = 5 independent numbers (equations 1 and 2 remove 2)

In most cases, the use of conditional independence reduces the size of the representation of the joint distribution from exponential in n to linear in n.

Conditional independence is our most basic and robust form of knowledge about uncertain environments.

Bayes' Rule

Product rule P(a ^ b) = P(a|b)P(b) = P(b|a)P(a)

The Probablity of a condition b is as same as the product of probability of b condition a and probability of a to the probability of b

$$\text{Bayes rule } P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

or in distribution form

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} = \alpha P(X|Y)P(Y)$$

Useful for assessing diagnostic probability from causal probability:

$$P(\text{Caust}|\text{Effect}) = \frac{P(\text{Effect}|\text{Cause})P(\text{Cause})}{P(\text{Effect})}$$

E.g., let M be meningitis, S be stiff neck:

$$P(m|s) = \frac{P(S|M)P(M)}{P(S)} = (0.8*0.0001)/0.1$$

$$=0.0008$$

Bayes' Rule and conditional independence

$$\begin{aligned}&\mathbf{P}(Cavity|toothache \wedge catch)\\ &= \alpha\,\mathbf{P}(toothache \wedge catch|Cavity)\mathbf{P}(Cavity)\\ &= \alpha\,\mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)\mathbf{P}(Cavity)\end{aligned}$$

This is an example of a naive Bayes model:

$$\mathbf{P}(Cause, Effect_1, \ldots, Effect_n) = \mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i|Cause)$$

Total number of parameters is **linear** in n

Naïve Bayes' Classifier

Naive Bayes classifiers are a collection of classification algorithms based on **Bayes' Theorem**. i.e. every pair of features being classified is independent of each other. To start with, let us consider a dataset.

Consider a fictional dataset that describes the weather conditions for playing a game of golf. Given the weather conditions, each tuple classifies the conditions as fit("Yes") or unfit("No") for playing golf.

	Outlook	Temperature	Humidity	Windy	Play Golf
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No

The dataset is divided into two parts, namely, **feature matrix** and the **response vector**.

- Feature matrix contains all the vectors(rows) of dataset in which each vector consists of the value of **dependent features**. In above dataset, features are 'Outlook', 'Temperature', 'Humidity' and 'Windy'.
- Response vector contains the value of **class variable**(prediction or output) for each row of feature matrix. In above dataset, the class variable name is 'Play golf'.

Assumption:

The fundamental Naive Bayes assumption is that each feature makes an:

- independent
- equal

contribution to the outcome.

With relation to our dataset, this concept can be understood as:

- We assume that no pair of features are dependent. For example, the temperature being 'Hot' has nothing to do with the humidity or the outlook being 'Rainy' has no effect on the winds. Hence, the features are assumed to be **independent**.
- Secondly, each feature is given the same weight(or importance). For example, knowing only temperature and humidity alone can't predict the outcome accurately. None of the attributes is irrelevant and assumed to be contributing **equally** to the outcome.

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$\text{Bayes rule } P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

where A and B are events and $P(B) \neq 0$.

- Basically, we are trying to find probability of event A, given the event B is true. Event B is also termed as **evidence**.
- P(A) is the **priori** of A (the prior probability, i.e. Probability of event before evidence is seen). The evidence is an attribute value of an unknown instance(here, it is event B).
- P(A|B) is a posteriori probability of B, i.e. probability of event after evidence is seen.

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}$$

where, y is class variable and X is a dependent feature vector (of size *n*)

where:$X=x_1,x_2,\ldots..X_n$

Just to clear, an example of a feature vector and corresponding class variable can be: (refer 1st row of dataset)

X = (Rainy, Hot, High, False)

y = No

So basically, P(y|X) here means, the probability of "Not playing golf" given that the weather conditions are "Rainy outlook", "Temperature is hot", "high humidity" and "no wind".

Naive assumption

Now, its time to put a naive assumption to the Bayes' theorem, which is, **independence** among the features. So now, we split **evidence** into the independent parts.

Now, if any two events A and B are independent, then,

P(A,B) = P(A)P(B)

Hence, we reach to the result:

$$P(y|x_1, \ldots, x_n) = \frac{P(x_1|y)P(x_2|y)\ldots P(x_n|y)P(y)}{P(x_1)P(x_2)\ldots P(x_n)}$$

which can be expressed as:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1)P(x_2)\ldots P(x_n)}$$

Now, as the denominator remains constant for a given input, we can remove that term:

$$P(y|x_1, \ldots, x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y)$$

Now, we need to create a classifier model. For this, we find the probability of given set of inputs for all possible values of the class variable *y* and pick up the output with maximum probability. This can be expressed mathematically as:

$$y = argmax_y P(y) \prod_{i=1}^{n} P(x_i|y)$$

So, finally, we are left with the task of calculating P(y) and P(x_i | y).

Please note that P(y) is also called **class probability** and P(x_i | y) is called **conditional probability**.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of P(x_i | y).

Let us try to apply the above formula manually on our weather dataset. For this, we need to do some precomputations on our dataset.

We need to find P(x_i | y_j) for each x_i in X and y_j in y. All these calculations have been demonstrated in the tables below:

Outlook

	Yes	No	P(yes)	P(no)
Sunny	2	3	2/9	3/5
Overcast	4	0	4/9	0/5
Rainy	3	2	3/9	2/5
Total	9	5	100%	100%

Temperature

	Yes	No	P(yes)	P(no)
Hot	2	2	2/9	2/5
Mild	4	2	4/9	2/5
Cool	3	1	3/9	1/5
Total	9	5	100%	100%

Humidity

	Yes	No	P(yes)	P(no)
High	3	4	3/9	4/5
Normal	6	1	6/9	1/5
Total	9	5	100%	100%

Wind

	Yes	No	P(yes)	P(no)
False	6	2	6/9	2/5
True	3	3	3/9	3/5
Total	9	5	100%	100%

Play		P(Yes)/P(No)
Yes	9	9/14
No	5	5/14
Total	14	100%

So, in the figure above, we have calculated P(x_i | y_j) for each x_i in X and y_j in y manually in the tables 1-4. For example, probability of playing golf given that the temperature is cool, i.e P(temp. = cool | play golf = Yes) = 3/9.

Also, we need to find class probabilities (P(y)) which has been calculated in the table 5. For example, P(play golf = Yes) = 9/14.

So now, we are done with our pre-computations and the classifier is ready!

Let us test it on a new set of features (let us call it today):

today = (Sunny, Hot, Normal, False)

So, probability of playing golf is given by:

$$P(Yes|today) = \frac{P(SunnyOutlook|Yes)P(HotTemperature|Yes)P(NormalHumidity|Yes)P(NoWind|Yes)P(Yes)}{P(today)}$$

and probability to not play golf is given by:

$$P(No|today) = \frac{P(SunnyOutlook|No)P(HotTemperature|No)P(NormalHumidity|No)P(NoWind|No)P(No)}{P(today)}$$

Since, P(today) is common in both probabilities, we can ignore P(today) and find proportional probabilities as:

$$P(Yes|today) \propto \frac{2}{9}.\frac{2}{9}.\frac{6}{9}.\frac{6}{9}.\frac{9}{14} \approx 0.0141$$

and

$$P(No|today) \propto \frac{3}{5}.\frac{2}{5}.\frac{1}{5}.\frac{2}{5}.\frac{5}{14} \approx 0.0068$$

Now, since

$$P(Yes|today) + P(No|today) = 1$$

These numbers can be converted into a probability by making the sum equal to 1 (normalization):

$$P(Yes|today) = \frac{0.0141}{0.0141+0.0068} = 0.67$$

and

$$P(No|today) = \frac{0.0068}{0.0141+0.0068} = 0.33$$

Since

$$P(Yes|today) > P(No|today)$$

So, prediction that golf would be played is ‘Yes’.

Bayesian networks

A simple, graphical notation for conditional independence assertions and hence for compact specification of full joint distributions

Syntax:

a set of nodes, one per variable

a directed, acyclic graph (link _ \directly influences")

a conditional distribution for each node given its parents:

$$\mathbf{P}(X_i|Parents(X_i))$$

In the simplest case, conditional distribution represented as a **conditional probability table** (CPT) giving the distribution over Xi for each combination of parent values

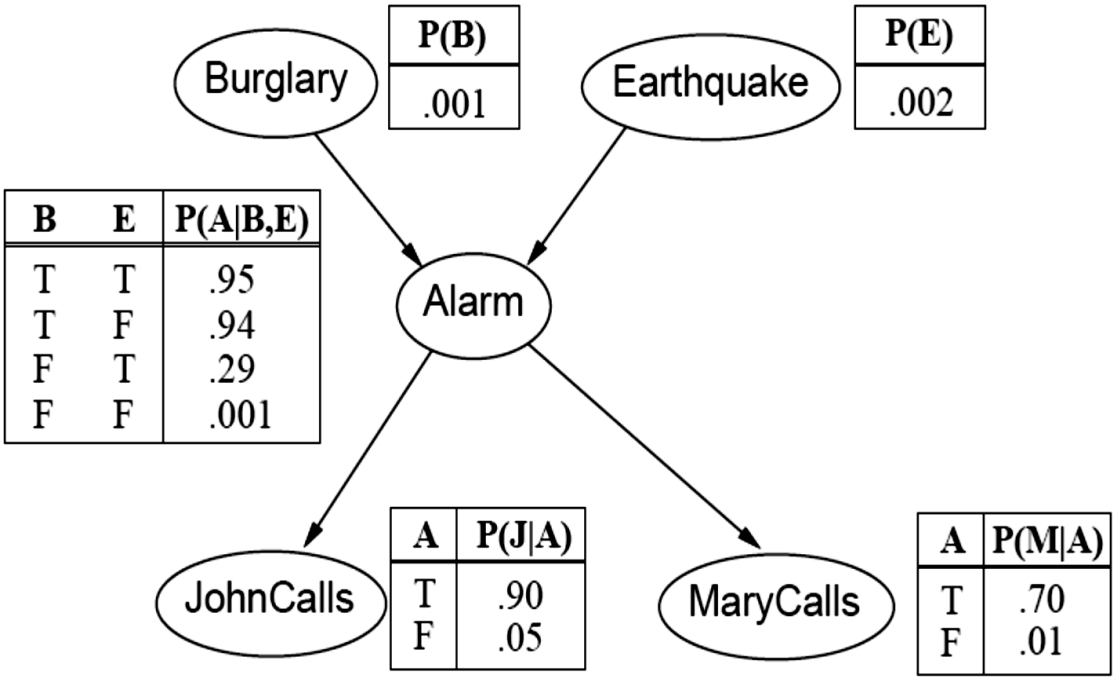
Example

I'm at work, neighbor John calls to say my alarm is ringing, but neighbor Mary doesn't call. Sometimes it's set of by minor earthquakes. Is there a burglar?

Variables: Burglar, Earthquake, Alarm, JohnCalls, MaryCalls

Network topology reacts \causal" knowledge:

- A burglar can set the alarm off
- An earthquake can set the alarm off
- The alarm can cause Mary to call
- The alarm can cause John to call



Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors John and Mary, who have taken a responsibility to inform Harry at work when they hear the alarm. John always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Mary likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and John and Mary both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Mary calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.
- The conditional distributions for each node are given as conditional probabilities table or CPT.
- Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
- In CPT, a boolean variable with k boolean parents contains 2^K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

- **Burglary (B)**
- **Earthquake(E)**
- **Alarm(A)**
- **John Calls(J)**
- **Mary calls(M)**

We can write the events of problem statement in the form of probability: **P[J, M, A, B, E]**, can rewrite the above probability statement using joint probability distribution:

P[J, M, A, B, E]= P[J | A]. P[M | A]. P[A| B, E]. P[B]. P[E]

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

P(M, J, A, ¬B, ¬E) = P (M|A) *P (J|A)*P (A|¬B ^ ¬E) *P (¬B) *P (¬E).

= 0.75* 0.91* 0.001* 0.998*0.999

= 0.00068045.

Hence, a Bayesian network can answer any query about the domain by using Joint distribution.

The semantics of Bayesian Network:

There are two ways to understand the semantics of the Bayesian network, which is given below:

1. To understand the network as the representation of the Joint probability distribution.

It is helpful to understand how to construct the network.

2. To understand the network as an encoding of a collection of conditional independence statements.

It is helpful in designing inference procedure.

Exact Inference in Bayesian Networks

The basic task for any probabilistic inference system is to compute the posterior probability EVENT distribution for a set of query variables, given some observed event—that is, some assignment of values to a set of evidence variables. To simplify the presentation, we will consider only one query variable at a time; the algorithms can easily be extended to queries with multiple variables. X denotes the query variable; E denotes the set of evidence variables E1,...,Em, and e is a particular observed event; Y will HIDDEN VARIABLE denotes the nonevidence, nonquery variables Y1,...,Yl. Thus, the complete set of variables is X = {X} ∪ E ∪ Y. A typical query asks for the posterior probability distribution P(X | e).

- **Enumeration**
- **Variable elimination**
- **Clustering Method**

Simple queries: compute posterior marginal $\mathbf{P}(X_i|\mathbf{E}=\mathbf{e})$
e.g., $P(NoGas|Gauge=empty, Lights=on, Starts=false)$

Conjunctive queries: $\mathbf{P}(X_i, X_j|\mathbf{E}=\mathbf{e}) = \mathbf{P}(X_i|\mathbf{E}=\mathbf{e})\mathbf{P}(X_j|X_i, \mathbf{E}=\mathbf{e})$

Optimal decisions: decision networks include utility information;
probabilistic inference required for $P(outcome|action, evidence)$

Value of information: which evidence to seek next?

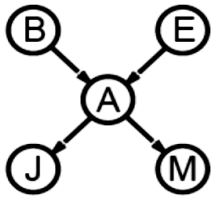
Sensitivity analysis: which probability values are most critical?

Explanation: why do I need a new starter motor?

Inference by enumeration

Simple query on the burglary network:

$$\begin{aligned}&\mathbf{P}(B|j,m)\\&=\mathbf{P}(B,j,m)/P(j,m)\\&=\alpha\mathbf{P}(B,j,m)\\&=\alpha\ \Sigma_e\ \Sigma_a\ \mathbf{P}(B,e,a,j,m)\end{aligned}$$



Rewrite full joint entries using product of CPT entries:

$$\begin{aligned}&\mathbf{P}(B|j,m)\\&=\alpha\ \Sigma_e\ \Sigma_a\ \mathbf{P}(B)P(e)\mathbf{P}(a|B,e)P(j|a)P(m|a)\\&=\alpha\mathbf{P}(B)\ \Sigma_e\ P(e)\ \Sigma_a\ \mathbf{P}(a|B,e)P(j|a)P(m|a)\end{aligned}$$

Recursive depth-first enumeration: $O(n)$ space, $O(d^n)$ time

Slightly intelligent way to sum out variables from the joint without actually constructing its explicit representation

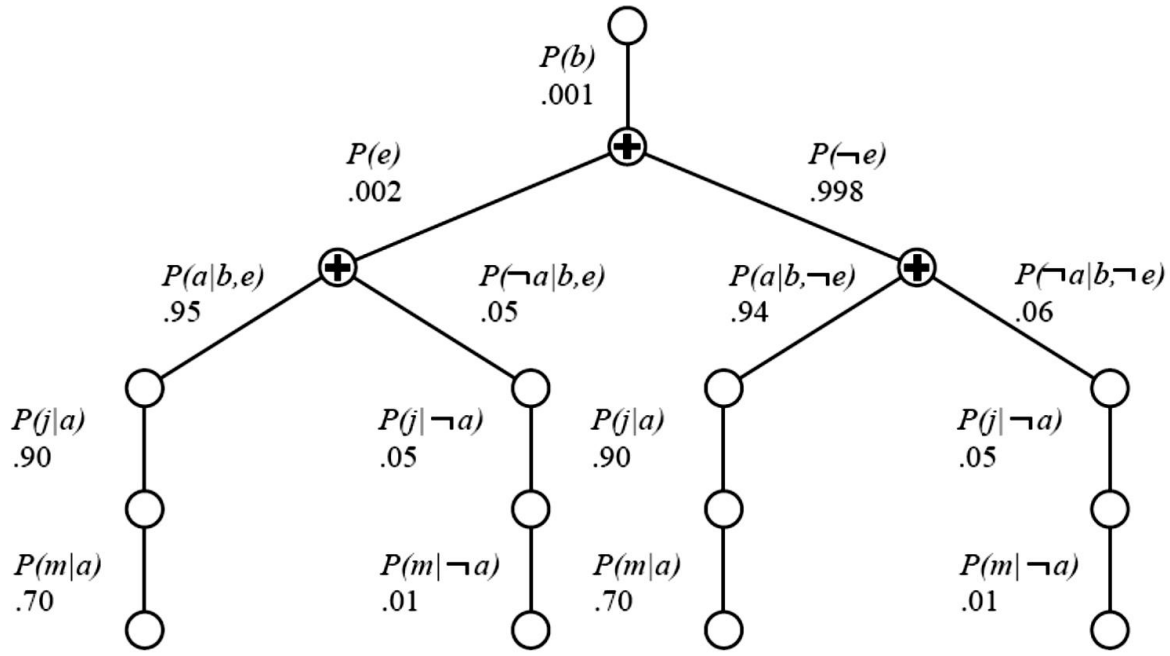
Enumeration algorithm

```
function ENUMERATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network with variables {X} ∪ E ∪ Y

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
      extend e with value x_i for X
      Q(x_i) ← ENUMERATE-ALL(VARS[bn], e)
  return NORMALIZE(Q(X))

function ENUMERATE-ALL(vars, e) returns a real number
  if EMPTY?(vars) then return 1.0
  Y ← FIRST(vars)
  if Y has value y in e
      then return P(y | Pa(Y)) × ENUMERATE-ALL(REST(vars), e)
      else return Σ_y P(y | Pa(Y)) × ENUMERATE-ALL(REST(vars), e_y)
          where e_y is e extended with Y = y
```

Evaluation tree



Enumeration is inefficient: repeated computation
e.g., computes $P(j|a)P(m|a)$ for each value of e

Variable elimination: carry out summations right-to-left, storing intermediate results (factors) to avoid recomputation

$$
\begin{aligned}
&\mathbf{P}(B|j,m) \\
&\quad = \alpha \underbrace{\mathbf{P}(B)}_{B} \Sigma_e \underbrace{P(e)}_{E} \Sigma_a \underbrace{\mathbf{P}(a|B,e)}_{A} \underbrace{P(j|a)}_{J} \underbrace{P(m|a)}_{M} \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a \mathbf{P}(a|B,e) P(j|a) f_M(a) \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a \mathbf{P}(a|B,e) f_J(a) f_M(a) \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a f_A(a,b,e) f_J(a) f_M(a) \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) f_{\bar{A}JM}(b,e) \text{ (sum out } A\text{)} \\
&\quad = \alpha \mathbf{P}(B) f_{\bar{E}\bar{A}JM}(b) \text{ (sum out } E\text{)} \\
&\quad = \alpha f_B(b) \times f_{\bar{E}\bar{A}JM}(b)
\end{aligned}
$$

Variable elimination: Basic operations

Summing out a variable from a product of factors: move any constant factors outside the summation add up submatrices in pointwise product of remaining factors

$$\Sigma_x f_1 \times \cdots \times f_k = f_1 \times \cdots \times f_i \Sigma_x f_{i+1} \times \cdots \times f_k = f_1 \times \cdots \times f_i \times f_{\bar{X}}$$

assuming $f_1, \ldots, f_i$ do not depend on X

Pointwise product of factors f_1 and f_2:

$$f_1(x_1, \ldots, x_j, y_1, \ldots, y_k) \times f_2(y_1, \ldots, y_k, z_1, \ldots, z_l) = f(x_1, \ldots, x_j, y_1, \ldots, y_k, z_1, \ldots, z_l)$$

E.g., $f_1(a,b) \times f_2(b,c) = f(a,b,c)$

Variable elimination algorithm

function ELIMINATION-ASK(X, e, bn) **returns** a distribution over X
  inputs: X, the query variable
    e, evidence specified as an event
    bn, a belief network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

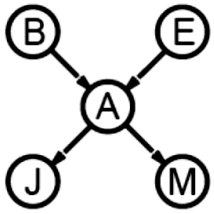
  factors ← []; *vars* ← REVERSE(VARS[bn])
  for each *var* **in** *vars* **do**
    factors ← [MAKE-FACTOR(*var*, e)|*factors*]
    if *var* is a hidden variable **then** *factors* ← SUM-OUT(*var*, *factors*)
  return NORMALIZE(POINTWISE-PRODUCT(*factors*))

Irrelevant variables

Consider the query $P(JohnCalls|Burglary=true)$

$$P(J|b) = \alpha P(b) \sum_e P(e) \sum_a P(a|b,e) P(J|a) \sum_m P(m|a)$$

Sum over m is identically 1; M is **irrelevant** to the query



Thm 1: Y is irrelevant unless $Y \in Ancestors(\{X\} \cup \mathbf{E})$

Here, $X = JohnCalls$, $\mathbf{E} = \{Burglary\}$, and
$Ancestors(\{X\} \cup \mathbf{E}) = \{Alarm, Earthquake\}$
so $MaryCalls$ is irrelevant

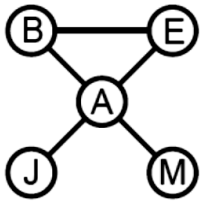
(Compare this to backward chaining from the query in Horn clause KBs)

Defn: moral graph of Bayes net: marry all parents and drop arrows

Defn: $\mathbf{A}$ is m-separated from $\mathbf{B}$ by $\mathbf{C}$ iff separated by $\mathbf{C}$ in the moral graph

Thm 2: Y is irrelevant if m-separated from X by $\mathbf{E}$

For $P(JohnCalls|Alarm=true)$, both
$Burglary$ and $Earthquake$ are irrelevant

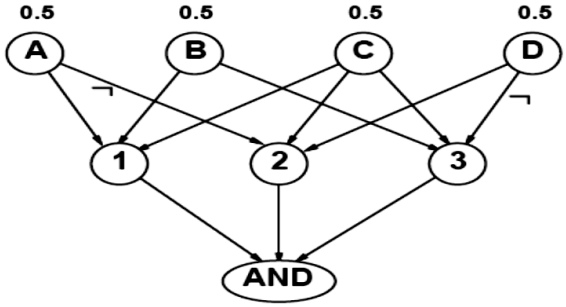


Complexity of exact inference

Singly connected networks (or polytrees):
- any two nodes are connected by at most one (undirected) path
- time and space cost of variable elimination are $O(d^k n)$

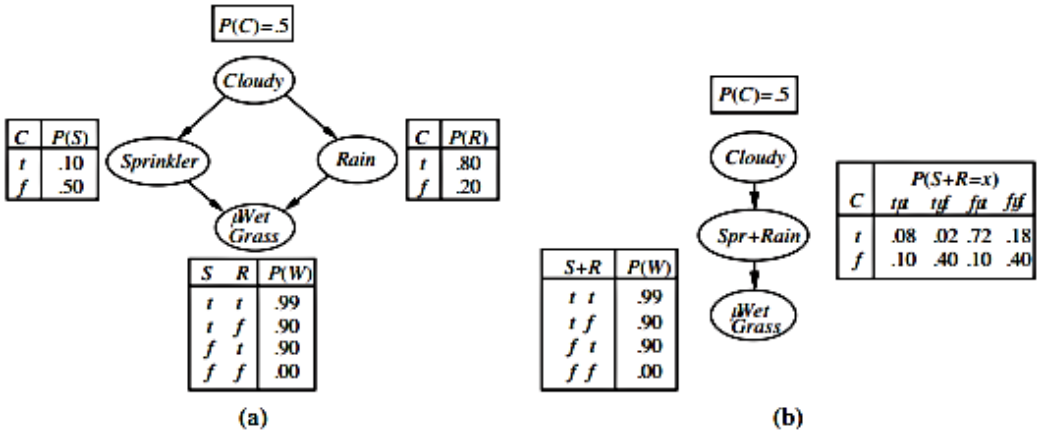
Multiply connected networks:
- can reduce 3SAT to exact inference $\Rightarrow$ NP-hard
- equivalent to **counting** 3SAT models $\Rightarrow$ #P-complete

1. A v B v C
2. C v D v ¬A
3. B v C v ¬D



Clustering algorithms

- If we want to compute posterior probabilities for all the variables in a network, however, it can be less efficient.
- For example, in a polytree network, one would need to issue O(n) queries costing O(n) each, for a total of O(n2 CLUSTERING) time.
- Using clustering algorithms (also known as JOIN TREE join tree algorithms), the time can be reduced to O(n).
- For this reason, these algorithms are widely used in commercial Bayesian network tools. The basic idea of clustering is to join individual nodes of the network to form cluster nodes in such a way that the resulting network is a polytree.
- For example, the multiply connected network, can be converted into a polytree by combining the Sprinkler and Rain node into a cluster node called Sprinkler+Rain.
- The two Boolean nodes are replaced by a "meganode" that takes on four possible values: tt, tf, f t, and ff.
- The meganode has only one parent, the Boolean variable Cloudy, so there are two conditioning cases.
- Although this example doesn't show it, the process of clustering often produces meganodes that share some variables.
- Probabilistic Reasoning Once the network is in polytree form, a special-purpose inference algorithm is required, because ordinary inference methods cannot handle meganodes that share variables with each other. Essentially, the algorithm is a form of constraint propagation.



PROBABILISTIC REASONING

- The topology of the network—the set of nodes and links—specifies the conditional independence relationships that hold in the domain, in a way that will be made precise shortly.
- The intuitive meaning of an arrow is typically that X has a direct influence on Y, which suggests that causes should be parents of effects. It is usually easy for a domain expert to decide what direct influences exist in the domain—much easier, in fact, than actually specifying the probabilities themselves.
- Once the topology of the Bayesian network is laid out, we need only specify a conditional probability distribution for each variable, given its parents.
- We will see that the combination of the topology and the conditional distributions suffices to specify (implicitly) the full joint distribution for all the variables.
- Consisting of the variables Toothache, Cavity, Catch, and Weather . We argued that Weather is independent of the other variables; furthermore, we argued that Toothache and Catch are conditionally independent, given Cavity. These relationships are represented by the Bayesian network structure shown.
- Formally, the conditional independence of Toothache and Catch, given Cavity, is indicated by the absence of a link between Toothache and Catch. Intuitively, the network represents the fact that Cavity is a direct cause of Toothache and Catch, whereas no direct causal relationship exists between Toothache and Catch.
- Now consider the following example, which is just a little more complex. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. (This example is due to Judea Pearl, a resident of Los Angeles—hence the acute interest in earthquakes.) You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm.
- John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with Weather Cavity Toothache Catch the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.
- The network structure shows that burglary and earthquakes directly affect the probability of the alarm's going off, but whether John and Mary call depends only on

the alarm. The network thus represents our assumptions that they do not perceive burglaries directly, they do not notice minor earthquakes, and they do not confer before calling.

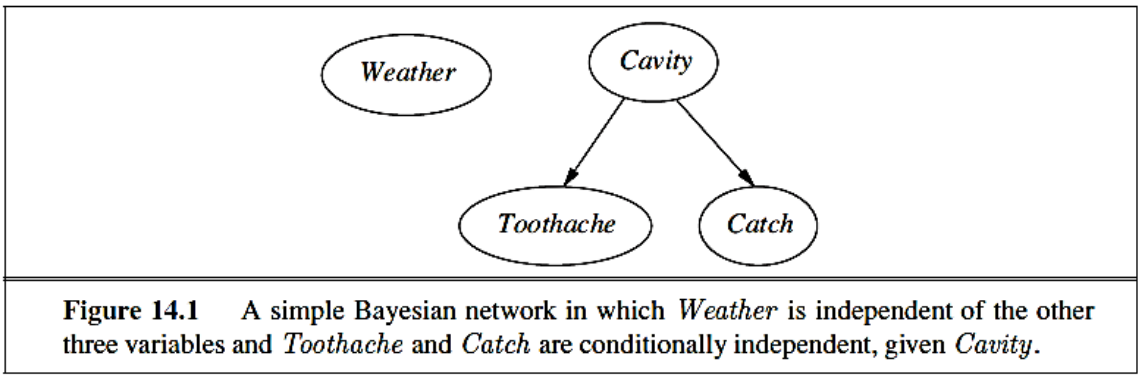


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

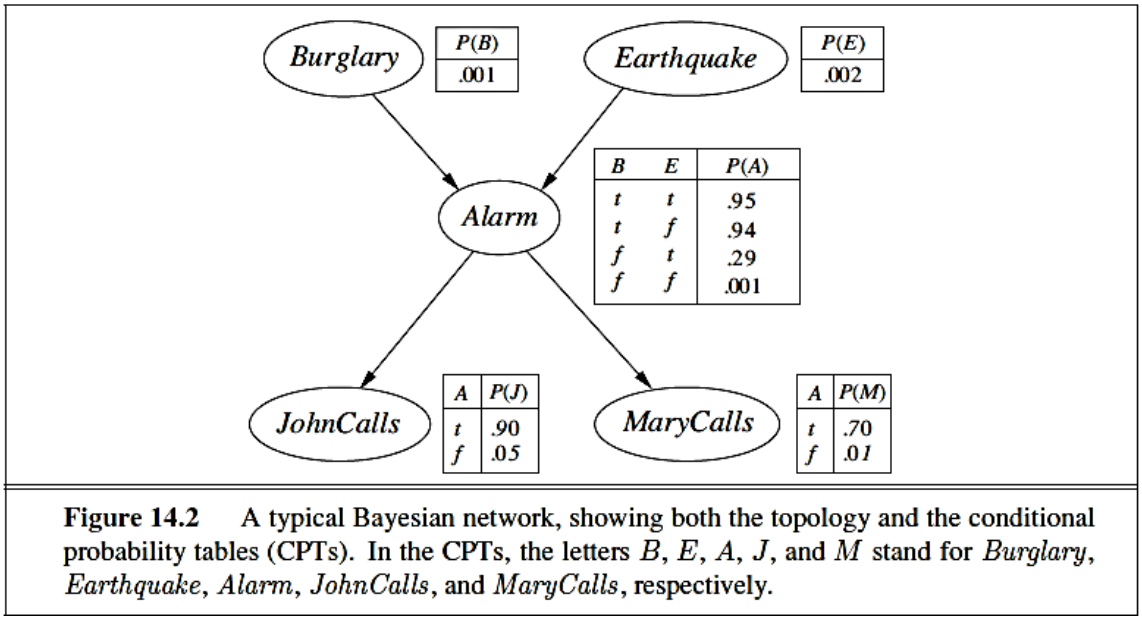


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

- The conditional distributions in Figure are shown as a conditional probability table, or CPT. Each row in a CPT contains the conditional probability of each node value for a conditioning case.
- A conditioning case is just a possible combination of values for the parent nodes—a miniature possible world, if you like. Each row must sum to 1, because the entries represent an exhaustive set of cases for the variable.
- For Boolean variables, once you know that the probability of a true value is p, the probability of false must be 1 – p, so we often omit the second number, as in a table for

a Boolean variable with k Boolean parents contains 2k independently specifiable probabilities.

- A node with no parents has only one row, representing the prior probabilities of each possible value of the variable. Notice that the network does not have nodes corresponding to Mary's currently listening to loud music or to the telephone ringing and confusing John.
- These factors are summarized in the uncertainty associated with the links from Alarm to JohnCalls and MaryCalls. shows both laziness and ignorance in operation: it would be a lot of work to find out why those factors would be more or less likely in any particular case, and we have no reasonable way to obtain the relevant information anyway.
- The probabilities actually summarize a potentially infinite set of circumstances in which the alarm might fail to go off (high humidity, power failure, dead battery, cut wires, a dead mouse stuck inside the bell, etc.) or John or Mary might fail to call and report it (out to lunch, on vacation, temporarily deaf, passing helicopter, etc.).
- In this way, a small agent can cope with a very large world, at least approximately. The degree of approximation can be improved if we introduce additional relevant information.

Approximate Inference using BN

- **Rejection sampling**
- **Likelihood weighting**
- **Markov chain Monte Carlo**
- **Direct Sampling**

Rejection sampling

- Used to compute conditional probabilities P(X|e)
- Generate samples as before
- Reject samples that do not match evidence
- Estimate by counting the how often event X is in the resulting samples

$\hat{\mathbf{P}}(X|\mathbf{e})$ estimated from samples agreeing with $\mathbf{e}$

```
function REJECTION-SAMPLING(X, e, bn, N) returns an estimate of P(X|e)
  local variables: N, a vector of counts over X, initially zero
  for j = 1 to N do
      x ← PRIOR-SAMPLE(bn)
      if x is consistent with e then
          N[x] ← N[x]+1 where x is the value of X in x
  return NORMALIZE(N[X])
```

E.g., estimate $\mathbf{P}(Rain|Sprinkler = true)$ using 100 samples
27 samples have $Sprinkler = true$
Of these, 8 have $Rain = true$ and 19 have $Rain = false$.

$\hat{\mathbf{P}}(Rain|Sprinkler = true) = \text{NORMALIZE}(\langle 8, 19\rangle) = \langle 0.296, 0.704\rangle$

Similar to a basic real-world empirical estimation procedure

Analysis of rejection sampling

$$\begin{aligned}
\hat{\mathbf{P}}(X|\mathbf{e}) &= \alpha \mathbf{N}_{PS}(X, \mathbf{e}) && \text{(algorithm defn.)} \\
&= \mathbf{N}_{PS}(X, \mathbf{e})/N_{PS}(\mathbf{e}) && \text{(normalized by } N_{PS}(\mathbf{e})\text{)} \\
&\approx \mathbf{P}(X, \mathbf{e})/P(\mathbf{e}) && \text{(property of PRIORSAMPLE)} \\
&= \mathbf{P}(X|\mathbf{e}) && \text{(defn. of conditional probability)}
\end{aligned}$$

Hence rejection sampling returns consistent posterior estimates

Problem: hopelessly expensive if $P(\mathbf{e})$ is small

$P(\mathbf{e})$ drops off exponentially with number of evidence variables!

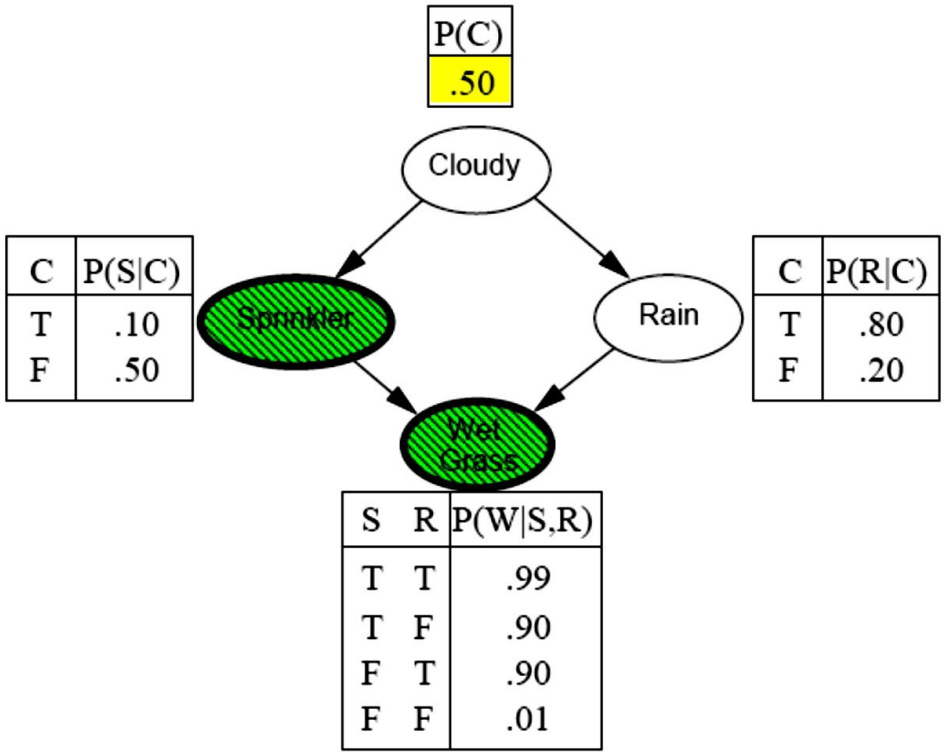
Likelihood weighting

- fix Values for evidence variables E,
- sample only Remaining variables X and Y
- This guarantees that each event generated is consistent with evidence.
- Before tallying the count in the distribution for the query variable.
- Each event is weighted by the likelihood that the event accords to the evidence.
- As measured by the product of Conditional probabilities for each evidence variables.

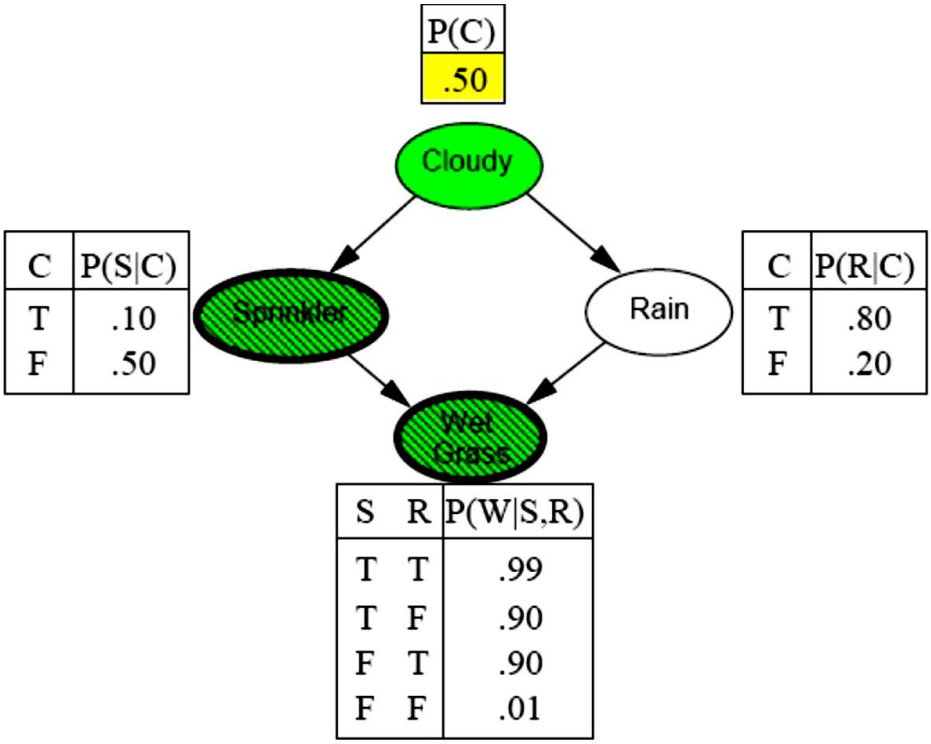
```
function LIKELIHOOD-WEIGHTING(X, e, bn, N) returns an estimate of P(X|e)
  local variables: W, a vector of weighted counts over X, initially zero
  for j = 1 to N do
      x, w ← WEIGHTED-SAMPLE(bn)
      W[x] ← W[x] + w where x is the value of X in x
  return NORMALIZE(W[X])

function WEIGHTED-SAMPLE(bn, e) returns an event and a weight
  x ← an event with n elements; w ← 1
  for i = 1 to n do
      if X_i has a value x_i in e
          then w ← w × P(X_i = x_i | parents(X_i))
          else x_i ← a random sample from P(X_i | parents(X_i))
  return x, w
```

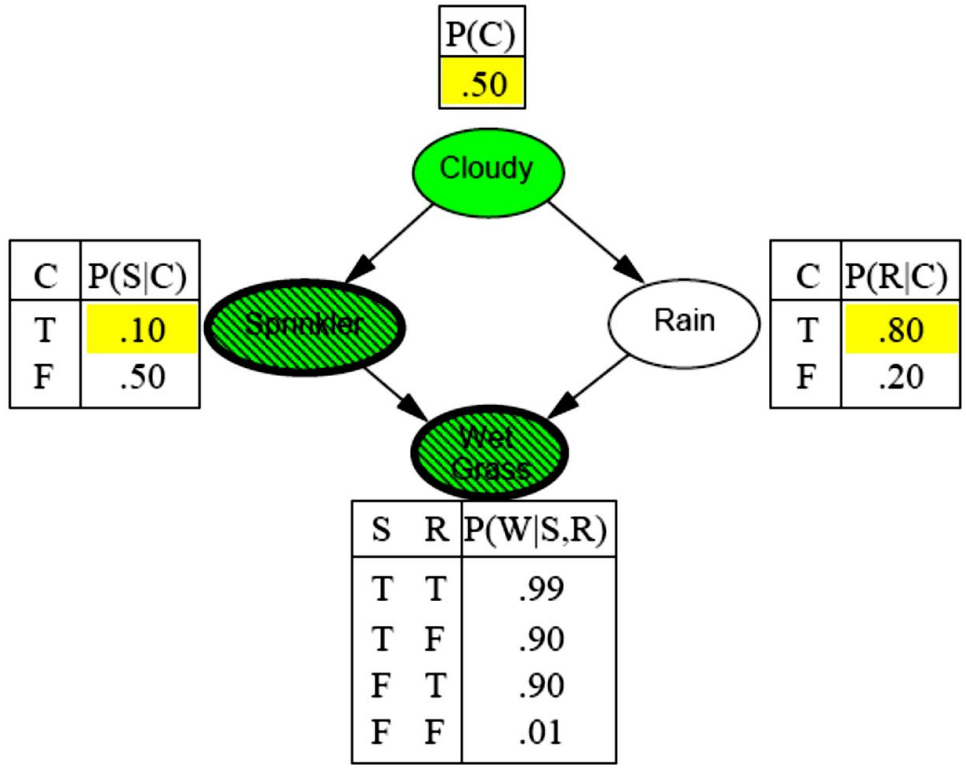
Likelihood weighting example



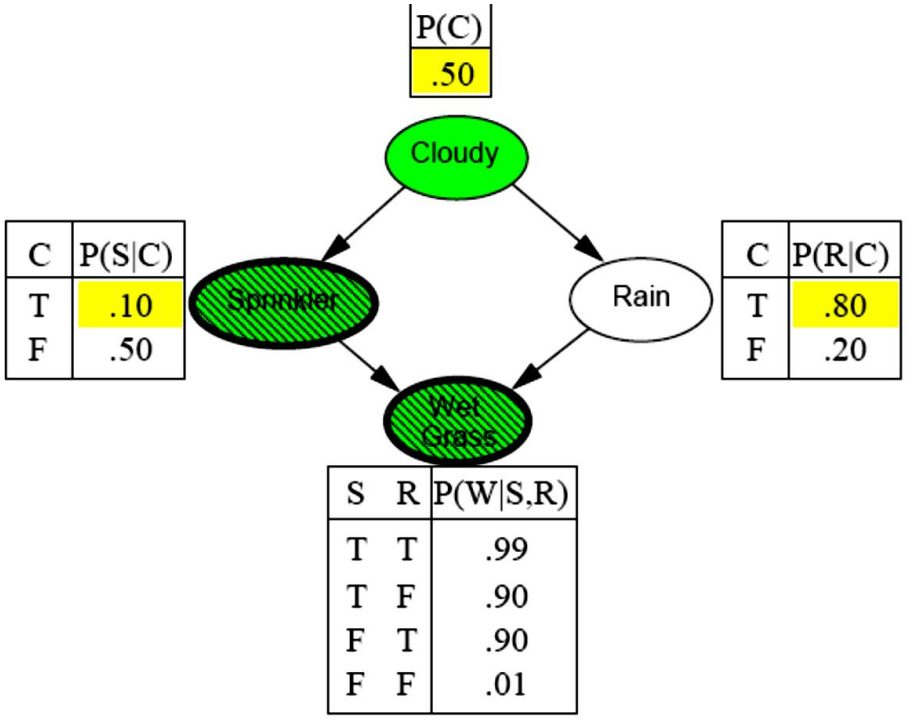
$w = 1.0$


P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01

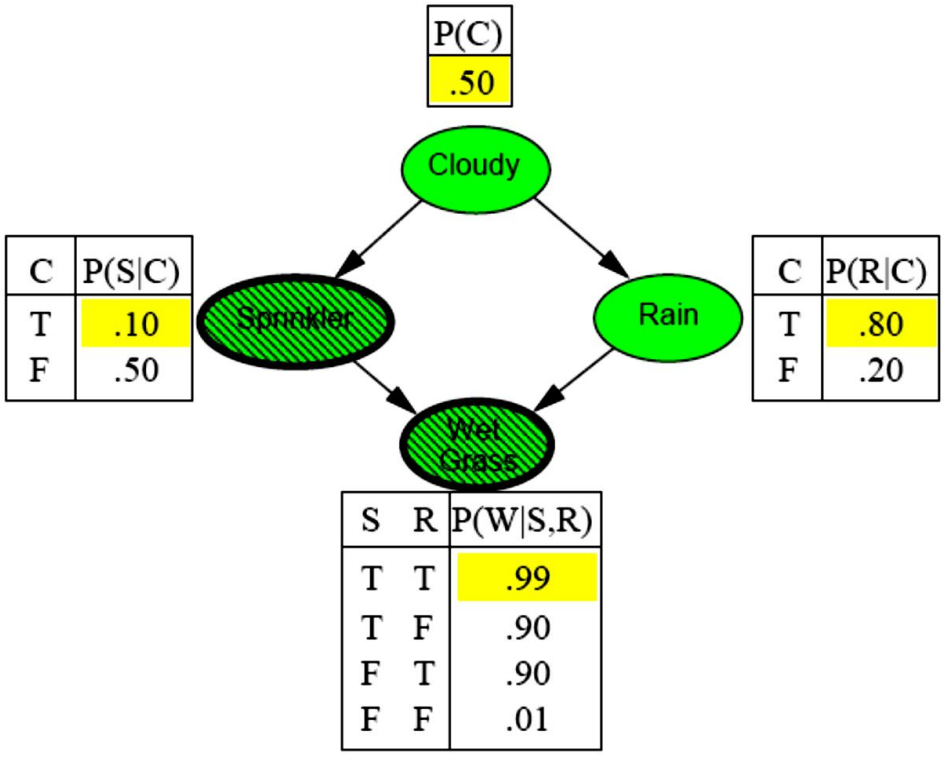
$w = 1.0$


P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01

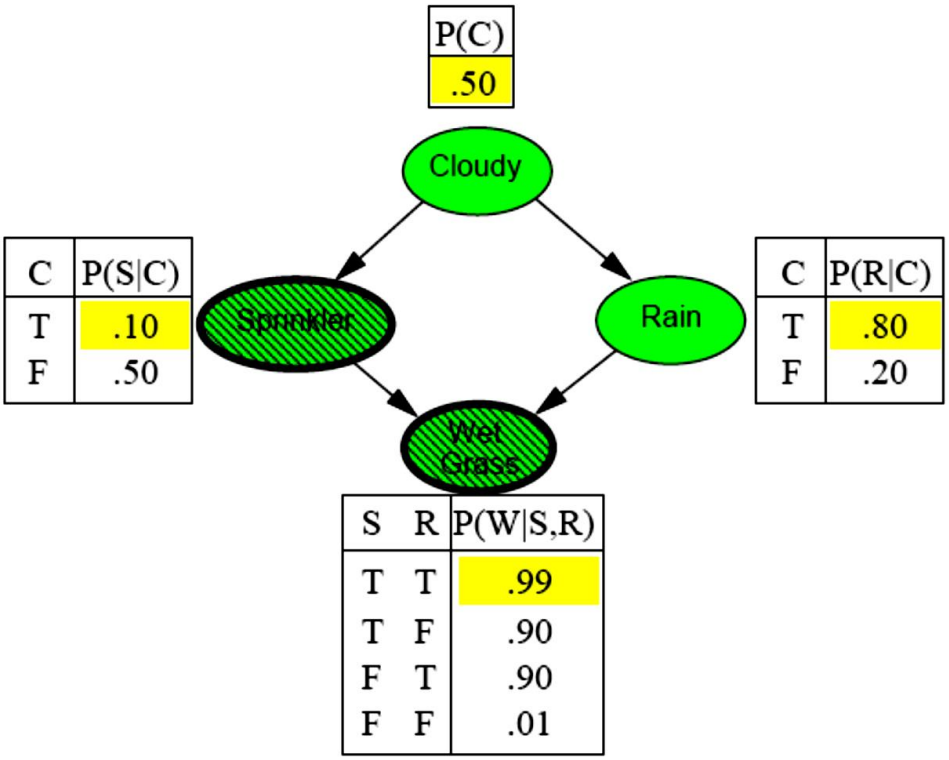
$w = 1.0$



$w = 1.0 \times 0.1$



$w = 1.0 \times 0.1$



$w = 1.0 \times 0.1 \times 0.99 = 0.099$

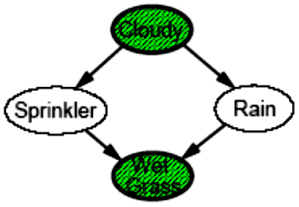
Likelihood weighting analysis

Sampling probability for WEIGHTEDSAMPLE is

$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{l} P(z_i | parents(Z_i))$

Note: pays attention to evidence in **ancestors** only

$\Rightarrow$ somewhere "in between" prior and posterior distribution



Weight for a given sample $\mathbf{z}, \mathbf{e}$ is

$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{m} P(e_i | parents(E_i))$

Weighted sampling probability is

$S_{WS}(\mathbf{z}, \mathbf{e}) w(\mathbf{z}, \mathbf{e})$

$= \prod_{i=1}^{l} P(z_i | parents(Z_i)) \prod_{i=1}^{m} P(e_i | parents(E_i))$

$= P(\mathbf{z}, \mathbf{e})$ (by standard global semantics of network)

Markov chain Monte Carlo (MCMC):

- It utilizes the “State" of network to assign values to all variables.
- Generate next state by sampling next variable along with previous state.

.

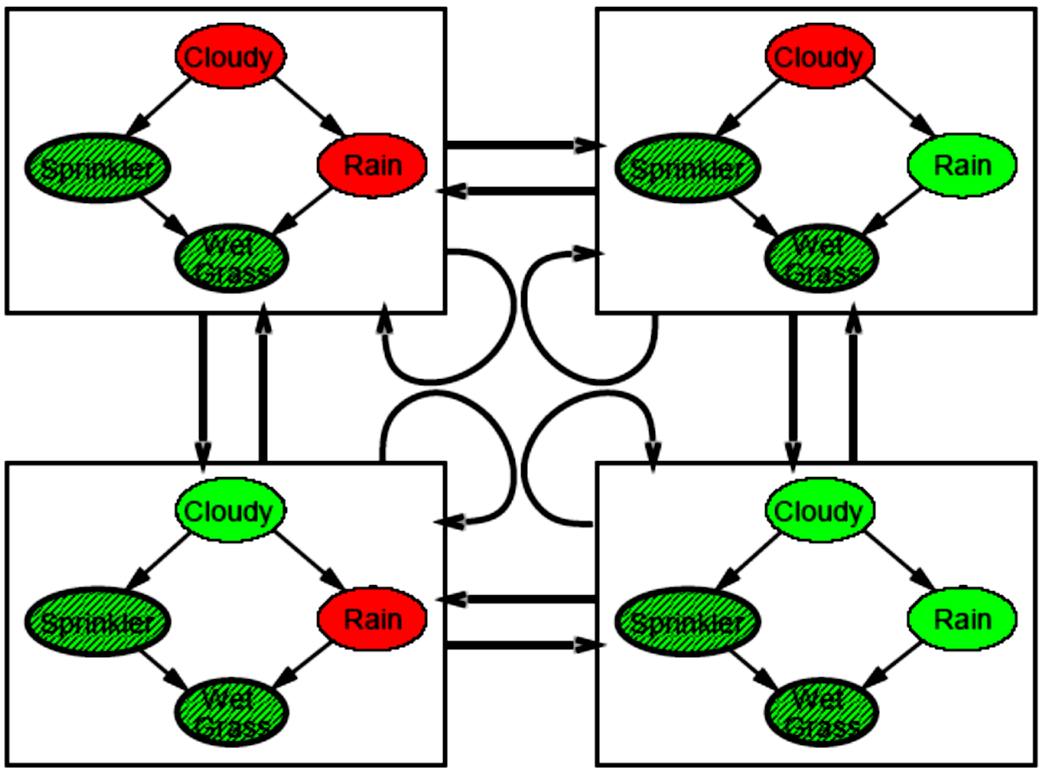
```
function MCMC-Ask(X, e, bn, N) returns an estimate of P(X|e)
  local variables: N[X], a vector of counts over X, initially zero
                   Z, the nonevidence variables in bn
                   x, the current state of the network, initially copied from e

  initialize x with random values for the variables in Y
  for j = 1 to N do
      for each Z_i in Z do
          sample the value of Z_i in x from P(Z_i|mb(Z_i))
              given the values of MB(Z_i) in x
          N[x] ← N[x] + 1 where x is the value of X in x
  return Normalize(N[X])
```

Can also choose a variable to sample at random each time

The Markov chain

- Generate new event by making random change to previous event.(current state:- specify values for each variable.
- The next state is generated by randomly sampling a value for one of the non evidence variables X_i
- Conditioned on current values of the variables in the Markov Blanket of X_i
- **P**(Rain | Sprinkler =true, WetGrass =true)
- With Cloudy; Rain as Non evidence there are four states.
    - Cloudy =true; Rain =true
    - Cloudy =true; Rain =False
    - Cloudy =false; Rain =True
    - Cloudy =false; Rain =False
- It randoms around the Space state (the space of Possible complete assignments) Thus flipping one variable at a time.



Estimate $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$

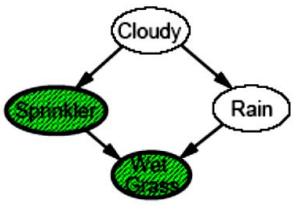
Sample $Cloudy$ or $Rain$ given its Markov blanket, repeat.
Count number of times $Rain$ is true and false in the samples.

E.g., visit 100 states
31 have $Rain = true$, 69 have $Rain = false$

$\hat{\mathbf{P}}(Rain|Sprinkler = true, WetGrass = true)$
$= \textsc{Normalize}(\langle 31, 69 \rangle) = \langle 0.31, 0.69 \rangle$

Markov blanket sampling

Markov blanket of $Cloudy$ is
$Sprinkler$ and $Rain$
Markov blanket of $Rain$ is
$Cloudy$, $Sprinkler$, and $WetGrass$



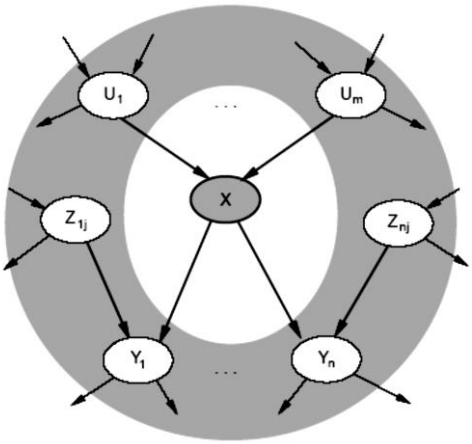
Probability given the Markov blanket is calculated as follows:
$P(x_i'|mb(X_i)) = P(x_i'|parents(X_i))\Pi_{Z_j \in Children(X_i)} P(z_j|parents(Z_j))$

Easily implemented in message-passing parallel systems, brains

Main computational problems:

1) Difficult to tell if convergence has been achieved
2) Can be wasteful if Markov blanket is large:
   $P(X_i|mb(X_i))$ won't change much (law of large numbers)

- sample from a stochastic process whose stationary distribution is the true posterior.
- Generate events by making a random change to the preceding event.
- This change is made using the Markov Blanket of the variable to be changed Markov Blanket = parents, children, children's parents



Direct Sampling

- The simplest kind of random sampling process for Bayesian networks generates events from a network that has no evidence associated with it.
- The idea is to sample each variable in turn, in topological order.
- The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents.

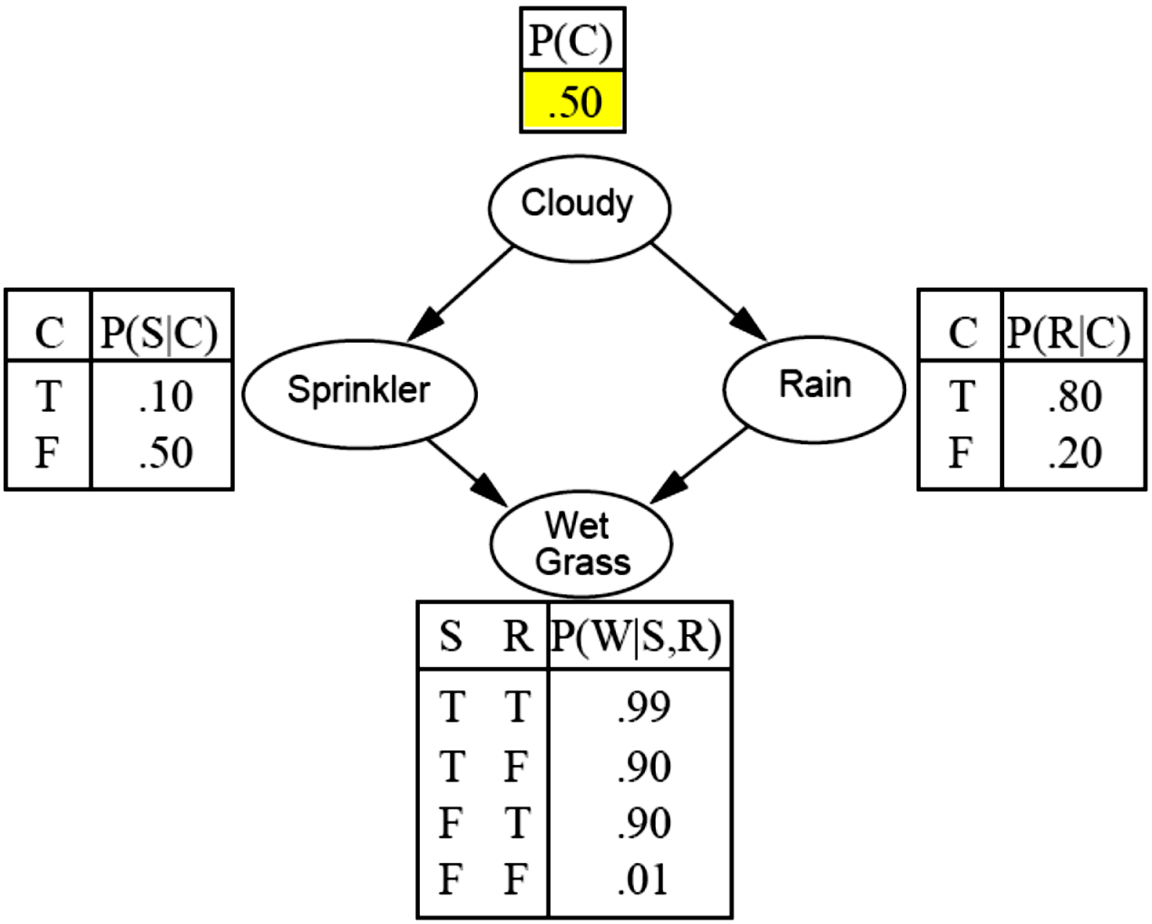
Example

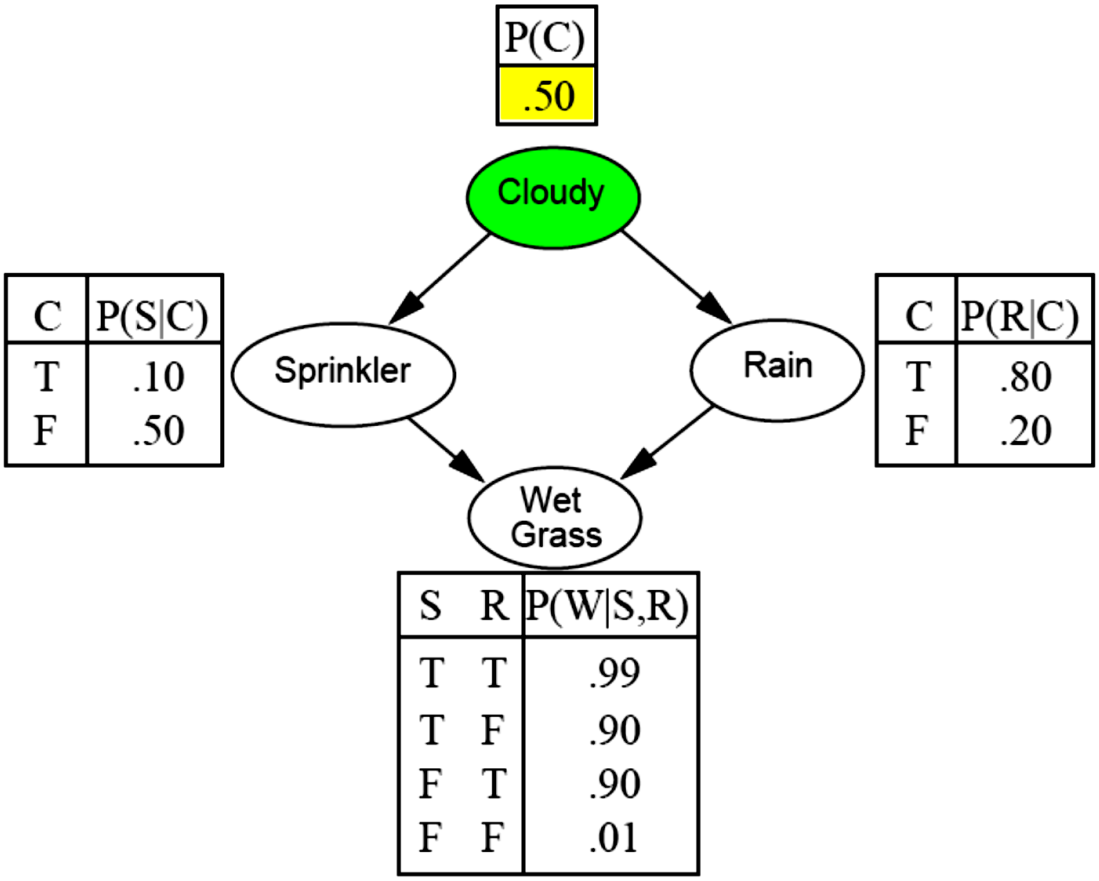
1. Sample from P(Cloudy) = 0.5, 0.5, value is true.
2. Sample from P(Sprinkler |Cloudy =true) = 0.1, 0.5, value is false.
3. Sample from P(Rain |Cloudy =true) = 0.8, 0.2, value is true.
4. Sample from P(WetGrass | Sprinkler =false, Rain =true) = 0.9, 0.1, value is true.

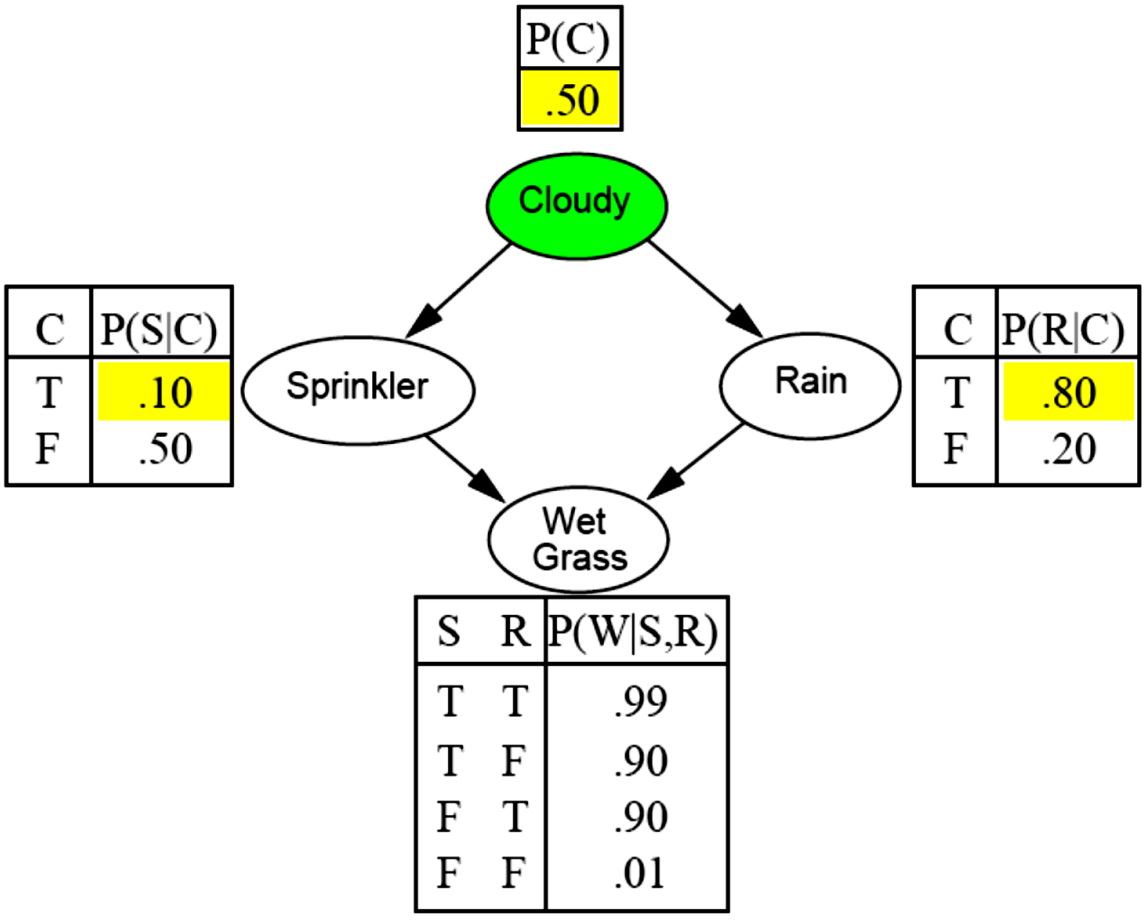
S_{PS}(true, false, true, true)
[Cloudy, Sprinkler, Rain, Wetgrass]

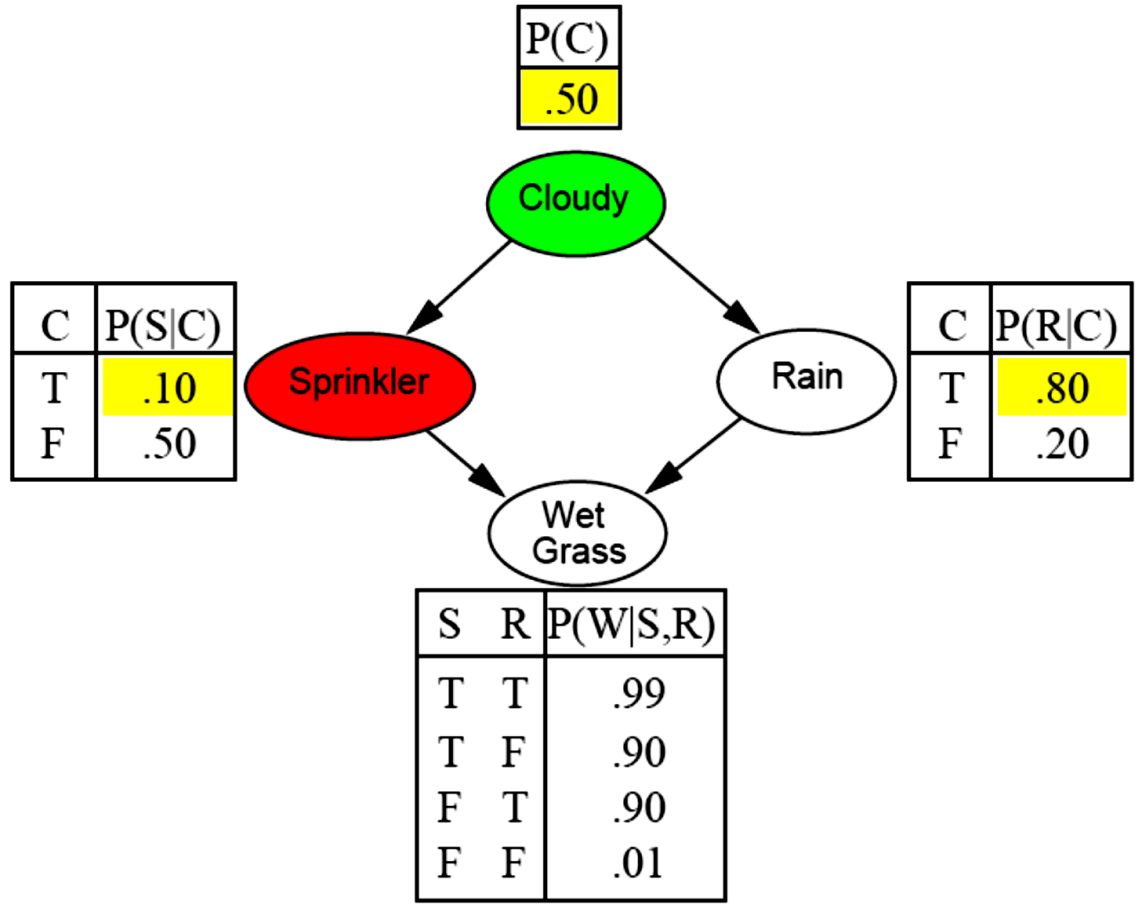
function PRIOR-SAMPLE(bn) **returns** an event sampled from bn
  inputs: bn, a belief network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

  $\mathbf{x} \leftarrow$ an event with n elements
  for $i = 1$ **to** n **do**
    $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
      given the values of $Parents(X_i)$ in $\mathbf{x}$
  return $\mathbf{x}$

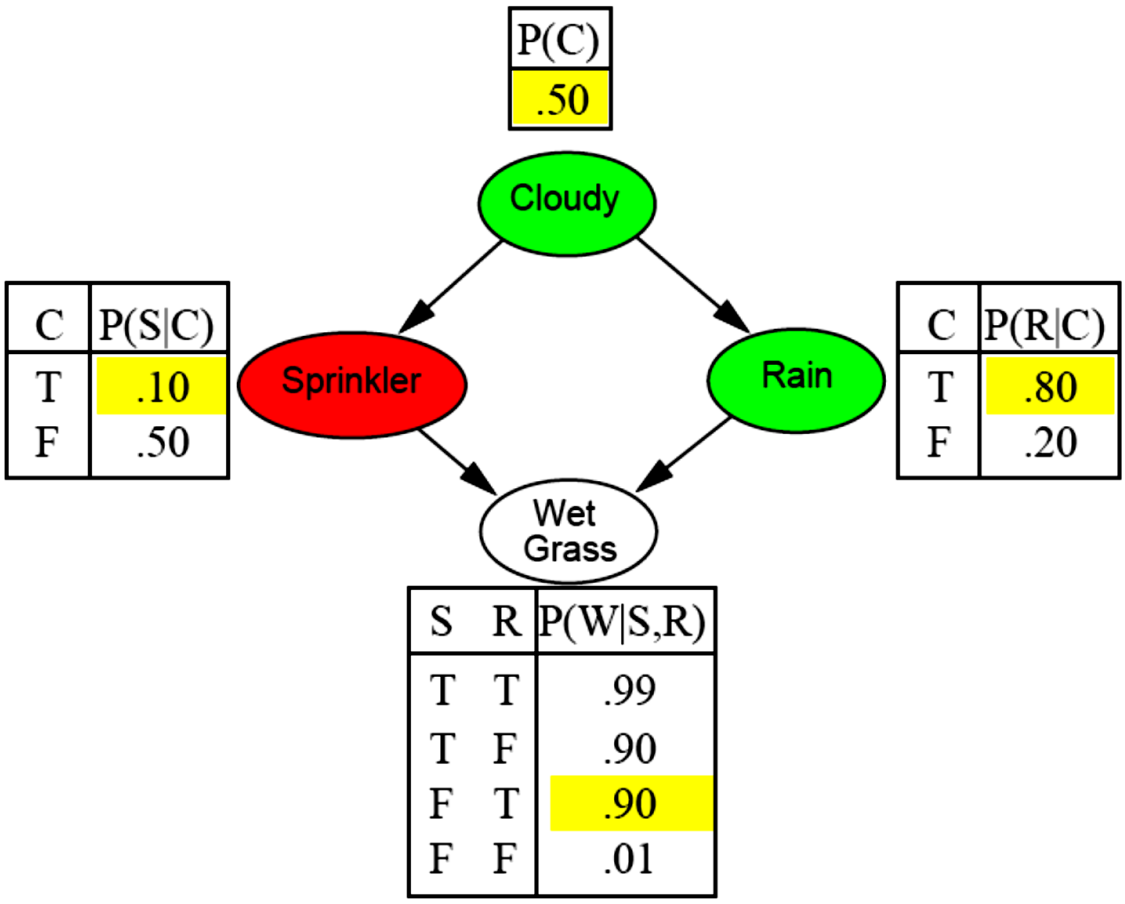


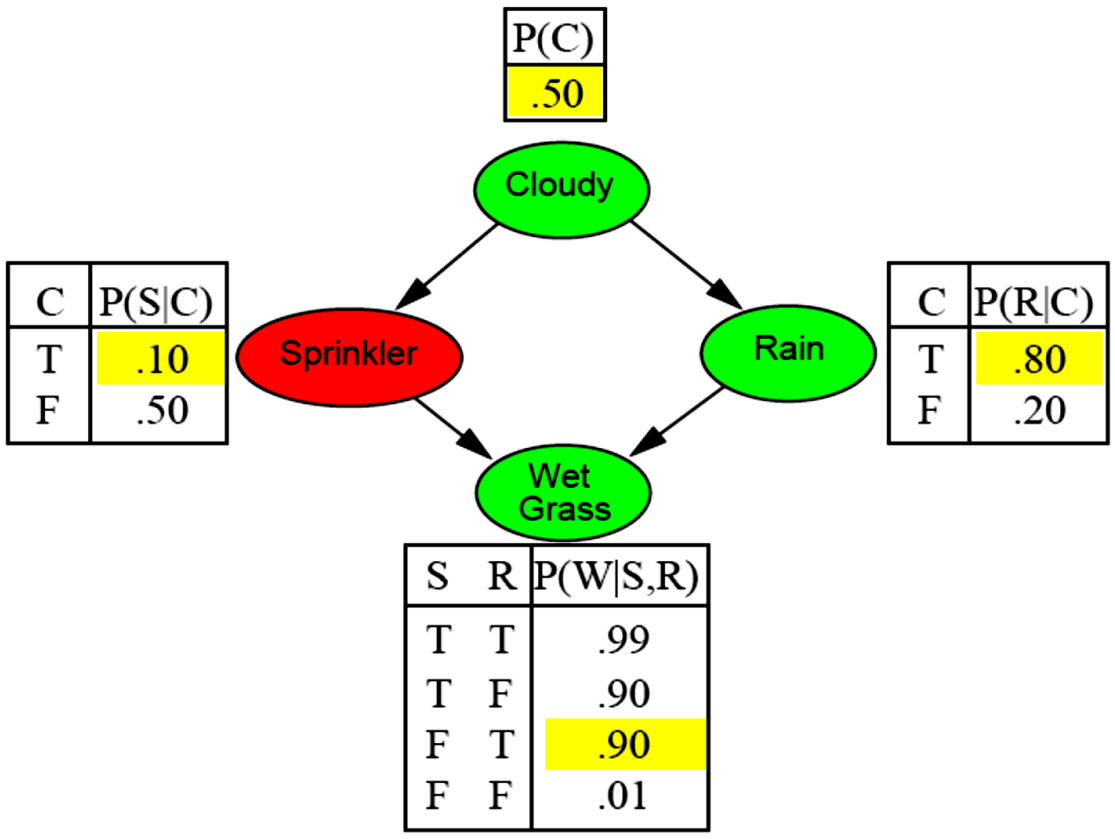

P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01


P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01



P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01


P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01


P(C)
.50
Cloudy
C P(S|C)
T .10
F .50
Sprinkler
Rain
C P(R|C)
T .80
F .20
Wet Grass
S R P(W|S,R)
T T .99
T F .90
F T .90
F F .01

Probability that PRIORSAMPLE generates a particular event

$$S_{PS}(x_1 \ldots x_n) = \Pi_{i=1}^{n} P(x_i | parents(X_i)) = P(x_1 \ldots x_n)$$

i.e., the true prior probability

E.g., $S_{PS}(t, f, t, t) = 0.5 \times 0.9 \times 0.8 \times 0.9 = 0.324 = P(t, f, t, t)$

Let $N_{PS}(x_1 \ldots x_n)$ be the number of samples generated for event $x_1, \ldots, x_n$

Then we have

$$\begin{aligned} \lim_{N \to \infty} \hat{P}(x_1, \ldots, x_n) &= \lim_{N \to \infty} N_{PS}(x_1, \ldots, x_n)/N \\ &= S_{PS}(x_1, \ldots, x_n) \\ &= P(x_1 \ldots x_n) \end{aligned}$$

That is, estimates derived from PRIORSAMPLE are consistent

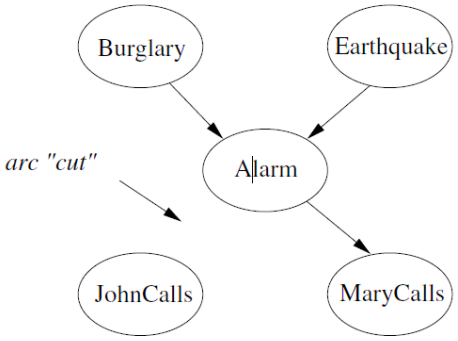
Shorthand: $\hat{P}(x_1, \ldots, x_n) \approx P(x_1 \ldots x_n)$

Causal inference

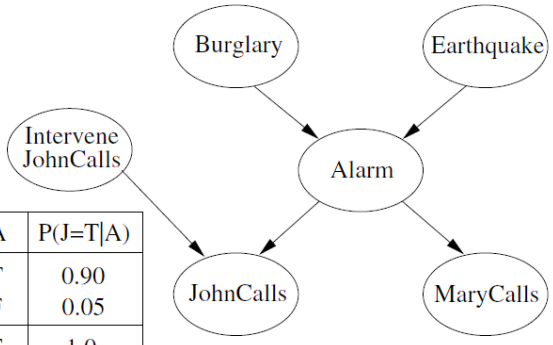
- The majority opinion is that there is nothing special about a causal interpretation, that is, one which asserts that corresponding to each (non-redundant) direct arc in the network not only is there a probabilistic dependency but also a causal dependency. after all, by reordering the variables and applying the network construction algorithm we can get the arcs turned around! Yet, clearly, both networks cannot be causal.
- That causal structure is what underlies all useful Bayesian networks. Certainly not all Bayesian networks are causal, but if they represent a real-world probability distribution, then some causal model is their source. Regardless of how that debate falls out, however, it is important to consider how to do inferences with Bayesian networks that are causal.
- If we have a causal model, then we can perform inferences which are not available with a non-causal BN. This ability is important, for there is a large range of potential applications for particularly causal inferences, such as process control, manufacturing and decision support for medical intervention.
- For Example, Consider again Pearl's earthquake network. That network is intended to represent a causal structure: each link makes a specific causal claim. Since it is a Bayesian network (causal or not), if we observe that JohnCalls is true, then this will raise the probability of MaryCalls being true, as we know. However, if we intervene, somehow forcing John to call, this probability raising inference will no longer be valid. Why?

Because the reason an observation raises the probability of Mary calling is that there is a common cause for both, the Alarm; so one provides evidence for the other.

- However, under intervention we have effectively cut off the connection between the Alarm and John's calling. The belief propagation (message passing) from JohnCalls to Alarm and then down to MaryCalls is all wrong under suggests that we understand the "effectively cut off" above quite literally, and model causal intervention in a variable simply by (temporarily) cutting all arcs from.
- If you do that with the earthquake example then, of course, you will find that forcing John to call will tell us nothing about earthquakes, burglaries, the Alarm or Mary — which is quite correct. This is the simplest way to model causal interventions and often will do the job.



(a)



I	A	P(J=T|A)
F	T	0.90
F	F	0.05
T	T	1.0
T	F	1.0

(b)

UNIT III

INTRODUCTION TO MACHINE LEARNING

Every time we buy a product, every time we rent a movie, visit a web page, write a blog, or post on the social media, even when we just walk or drive around, many data are generated.

Each of us is not only a generator but also a consumer of data. One also want to have products and services specialized for us. Each of our needs has to be understood and interests to be predicted.

for example, of a supermarket chain that is selling thousands of goods to millions of customers either at hundreds of brick-and-mortar stores all over a country or through a virtual store over the web. The details of each transaction are stored: date, customer id, goods bought and their amount, total money spent, and so forth. This typically amounts to a lot of data every day. What the supermarket chain wants is to be able to predict which customer is likely to buy which product, to maximize sales and profit. Similarly each customer wants to find the set of products best matching his/her needs.

Applications of ML

- Mine databases to obtain Important Information.
- To Reduce the Feature space of Large Databases.
- To optimize the Learning Principle by Repetitive learning to improve performance.
- To predict and classify type one Normal data from the type two abnormal data in dataset by using Learning Models on dataset.
- Application of machine learning methods to large databases is called data mining

Applications of ML in Real world App

1. **In retail Firms**- To analyse and buying pattern of goods purchased together.
2. **In credit card applications**- It is used to detect fraud credit cards by analysing its credentials,
3. **In finance banks**- It is used to analyse their past data to build models for to see, the predictiveness on loan approvals, based many loan payment details.
4. **In the stock market**- It is used to predict recommended Firm's Shares to be purchased at correct time based on previous share up and down price.

5. **In manufacturing-** learning models are used for optimization, control, and troubleshooting.
6. **In medicine**, learning programs are used for medical diagnosis to classify Healthy patients from abnormal patients.
7. **In telecommunications**, call patterns are analyzed for network optimization and maximizing the quality of service.
8. **In science**, large amounts of data in physics, astronomy, and biology can only be analyzed fast enough by computers.
9. **The World Wide Web** is huge; it is constantly growing, and searching for relevant information cannot be done manually.

Data Mining- The analogy is that a large volume of earth and raw material is extracted from a mine, which when processed leads to a small amount of very precious material; similarly, in data mining, a large volume of data is processed to extract simple valuable information.

Artificial Intelligence in ML

Machine learning is also a part of artificial intelligence. that it has the ability to learn in a changing environment. If the system can learn and adapt to such changes, the system designer need not foresee and provide solutions for all possible situations. Machine learning also helps us to find solutions for many problems in Computer vision, speech recognition, and robotics.

EXAMPLES OF MACHINE LEARNING APPLICATIONS

Associations Learning

In the case of retail—for example, a supermarket chain—one application of machine learning is *basket analysis*, which is finding associations between products bought by customers: If people who buy X typically also buy Y, and if there is a customer who buys X and does not buy Y, he or she is a potential Y customer. Once we find such customers, we can target them for cross-selling.

association rule

In order to find an *association rule*, we are interested in learning a conditional probability of the form $P(Y|X)$ where Y is the product we would like to condition on X, which is the product or the set of products which we know that the customer has already purchased.

Let us say, going over our data, we calculate that $P(\text{Bread}|\text{Milk}) = 0.7$. Then, we can define the rule: 70 percent of customers who buy Milk also buy Bread. We may want to make

a distinction among customers and toward this, estimate $P(Y|X,D)$ where D is the set of customer attributes, for example,

gender, age, marital status, and so on, assuming that we have access to this information. If this is a bookseller instead of a supermarket, products can be books or authors. In the case of a web portal, items correspond to links to web pages, and we can estimate the links a user is likely to click and use this information to download such pages in advance for faster access.

Classification

A credit is an amount of money loaned by a financial institution, for example, a bank, to be paid back with interest, generally in installments. It is important for the bank to be able to predict in advance the risk associated with a loan, which is the probability that the customer will default and not pay the whole amount back. This is both to make sure that the bank will make a profit and also to not inconvenience a customer with a loan over his or her financial capacity.

In *credit scoring*, the bank calculates the risk given the amount of credit and the information about the customer. The information about the customer includes data we have access to and is relevant in calculating his or her financial capacity—namely, income, savings, collaterals, profession, age, past financial history, and so forth. The bank has a record of past loans containing such customer data and whether the loan was paid back or not. From this data of particular applications, the aim is to infer a general rule coding the association between a customer's attributes and his risk. That is, the machine learning system fits a model to the past data to be able to calculate the risk for a new application and then decides to accept or refuse it accordingly.

classification

In this example of *classification*, where there are two classes: low-risk and high-risk customers. The information about a customer makes up the *input* to the classifier whose task is to assign the input to one of the two classes. After training with the past data, a classification rule learned may be of the form

$$\text{IF income} > \theta_1 \text{ AND savings} > \theta_2 \text{ THEN low-risk ELSE high-risk}$$

for suitable values of θ_1 and θ_2 . This is an example of a *discriminant*; it is a function that separates the examples of different classes.

prediction

Once we have set a rule that fits the past data, if the future is similar to the past, then we can make correct predictions for novel instances. Given a new example with a certain income and savings, we can easily decide whether it is lowrisk or high-risk.

In some cases, instead of making a 0/1 (low-risk/high-risk) type decision, we may want to calculate a probability, namely, *P(Y|X)*, where *X* are the customer attributes and *Y* is 0 or 1 respectively for low-risk

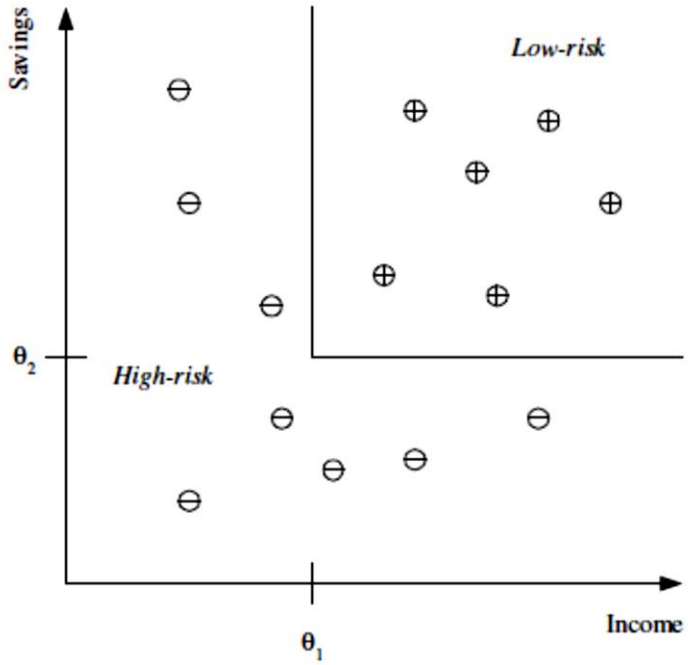


Figure Example of a training dataset where each circle corresponds to one data instance with input values in the corresponding axes and its sign indicates the class. For simplicity, only two customer attributes, income and savings, are taken as input and the two classes are low-risk ('+') and high-risk ('−'). An example discriminant that separates the two types of examples is also shown. and high-risk. From this perspective, we can see classification as learning an association from *X* to *Y*. Then for a given $X = x$, if we have $P(Y = 1|X = x) = 0.8$, we say that the customer has an 80 percent probability of being high-risk, or equivalently a 20 percent probability of being low-risk. We then decide whether to accept or refuse the loan depending on the possible gain and loss.

pattern recognition is one of the field of machine learning, It identifies similar repetitive patterns from the data space,

for example a *optical character recognition,* an problem in which it has to recognizing character codes from their images. In this example there were multiple classes, as many as there are characters we would like to recognize. Especially interesting is the case when the characters are handwritten—

for we take samples from writers and learn a definition of A-ness from these examples. But though we do not know what it is that makes an image an 'A', we are certain that all those

distinct ‘A’s have some key properties in common, which is what we want to extract from the examples. We know that a character image is a collection of strokes and has a regularity that we can capture by a learning program. If we are reading a text, one factor we can make use of is the redundancy in human languages. A word is a *sequence* of characters and successive characters are not independent but are constrained by the words of the language. This has the advantage that even if we cannot recognize a character, we can still read the word. Such contextual dependencies may also occur in higher levels, between words and sentences, through the syntax and semantics of the language. There are machine learning algorithms to learn sequences and model such dependencies.

In *face recognition*, the input is an image, the classes are people to be recognized, and the learning program should learn to associated the face images to identities. This problem is more difficult than optical character recognition because there are more classes, input image is larger, and a face is three-dimensional and differences in pose and lighting cause significant changes in the image. There may also be occlusion of certain inputs; for example, glasses may hide the eyes and eyebrows, and a beard may hide the chin.

In *medical diagnosis*, the inputs are the relevant information we have about the patient and the classes are the illnesses. The inputs contain the patient’s age, gender, past medical history, and current symptoms. Some tests may not have been applied to the patient, and thus these inputs would be missing. Tests take time, may be costly, and may inconvenience the patient so we do not want to apply them unless we believe that they will give us valuable information. In the case of a medical diagnosis, a wrong decision may lead to a wrong or no treatment, and in cases of doubt it is preferable that the classifier reject and defer decision to a human expert.

In *speech recognition*, the input is acoustic and the classes are words that can be uttered. This time the association to be learned is from an acoustic signal to a word of some language. Different people, because of differences in age, gender, or accent, pronounce the same word differently, which makes this task rather difficult. Another difference of speech is that the input is *temporal*; words are uttered in time as a sequence of speech phonemes and some words are longer than others. Acoustic information only helps up to a certain point, and as in optical character recognition, the integration of a “language model” is critical in speech recognition, and the best way to come up with a language model is again by learning it from some large corpus of example data.

In *natural language processing*, Spam filtering is one where spam generators on one side and filters on the other side keep finding more and more ingenious ways to outdo each

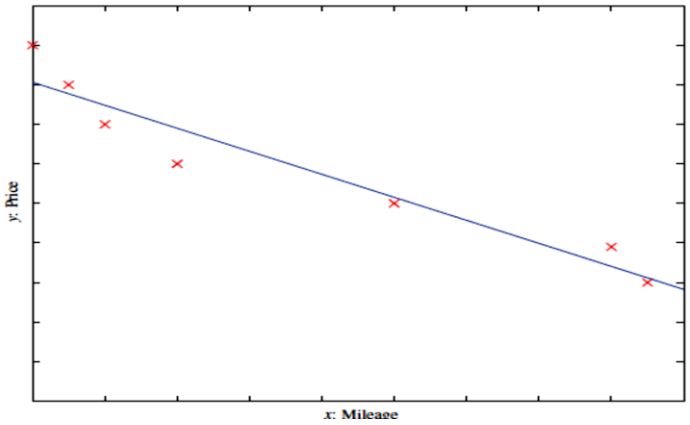
other. Summarizing large documents is another interesting example, yet another is analyzing blogs or posts on social networking sites to extract "trending" topics or to determine what to advertise. Perhaps the most impressive would be *machine translation*. After decades of research on hand-coded translation rules, it has become apparent that the most promising way is to provide a very large number of example pairs of texts in both languages and have a program figure out automatically the rules to map one to the other.

Biometrics is recognition or authentication of people using their physiological and/or behavioral characteristics that requires an integration of inputs from different modalities. Examples of physiological characteristics are images of the face, fingerprint, iris, and palm; examples of behavioral characteristics are dynamics of signature, voice, gait, and key stroke. As opposed to the usual identification procedures—photo, printed signature, or password—when there are many different (uncorrelated) inputs, forgeries (spoofing) would be more difficult and the system would be more accurate, hopefully without too much inconvenience to the users. Machine learning is used both in the separate recognizers for these different modalities and in the combination of their decisions to get an overall accept/reject decision, taking into account how reliable these different sources are.

Knowledge Extraction

Learning a rule from data also allows *knowledge extraction*. The rule is a simple model that explains the data, and looking at this model we have an explanation about the process underlying the data. For example, once we learn the discriminant separating low-risk and high-risk customers, we have the knowledge of the properties of low-risk customers. We can then use this information to target potential low-risk customers more efficiently, for example, through advertising. Learning also performs program optimizes the parameters, θ, such that the approximation error is minimized, that is, our estimates are as close as possible to the correct values given in the training set.

For example in figure, the model is linear, and w and w_0 are the parameters optimized for best fit to the training data. In cases where the linear model is too restrictive,



A training dataset of used cars and the function fitted. For simplicity, mileage is taken as the only input attribute and a linear model is used.

$$y = w_2 x_2 + w_1 x + w_0$$

Another example of regression is navigation of a mobile robot, for example, an autonomous car, where the output is the angle by which the steering wheel should be turned at each time, to advance without hitting obstacles and deviating from the route. Inputs in such a case are provided by sensors on the car—for example, a video camera, GPS, and so forth. Training data can be collected by monitoring and recording the actions of a human driver.

We can envisage other applications of regression where we are trying to optimize a function.1 Let us say we want to build a machine that roasts coffee. The machine has many inputs that affect the quality: various settings of temperatures, times, coffee bean type, and so forth. We make a number of experiments and for different settings of these inputs, we measure the quality of the coffee, for example, as consumer satisfaction. To find the optimal setting, we fit a regression model linking these inputs to coffee quality and choose new points to sample near the optimum of the current model to look for a better configuration. We sample these points, check quality, and add these to the data and fit a new model. This is generally called *response surface design*.

Sometimes instead of estimating an absolute numeric value, we want to be able to learn relative positions. For example, in a *recommendation system* for movies, we want to generate a list ordered by how much we believe the user is likely to enjoy each. Depending on the movie attributes such as genre, actors, and so on, and using the ratings of the user he/she has already seen, we would like to be able to learn a *ranking* function that we can then use to choose among new movies.

SUPERVISED LEARNING

The task of supervised learning is this:

Given a **training set** of N example input–output pairs

$$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N) ,$$

where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.

Here x and y can be any value; they need not be numbers. The function h is a **hypothesis**. Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we

give it a **test set** of examples that are distinct from the training set.

a. CLASSIFICATION

When the output y is one of a finite set of values (such as *sunny, cloudy* or *rainy*), the learning problem is called **classification**, and is called Boolean or binary classification if there are only two values.

b. REGRESSION

When y is a number (such as tomorrow's temperature), the learning problem is called **regression**. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found *exactly* the right real-valued number for y is 0.)

1. CLASSIFICATION

Learning a Class from Examples

Let us say we want to learn the *class*, C, of a "family car." We have a set of examples of cars, and we have a group of people that we survey to whom we show these cars. The people look at the cars and label them; the cars that they believe are family cars are *positive examples*, and the other cars are *negative examples*. Class learning is finding a description that is shared by all the positive examples and none of the negative examples.

ID	mpg	Displac-ement	Engine-power	weight	Acceler-ation	car_name	Price in $	Class
1	18	307	130	3504	12	chevrolet chevelle	25561	Family car
2	15	350	165	3693	11.5	buick skylark 320	24221	Family car
3	18	318	150	3436	11	plymouth satellite	27240	Family car
4	16	304	150	3433	12	amc rebel sst	33684	Family car
5	17	302	140	3449	10.5	ford torino	20000	Family car
6	15	429	198	4341	10	ford galaxie 500	30000	Family car
7	14	454	220	4354	9	chevrolet impala	35764	Not Family car
8	14	440	215	4312	8.5	plymouth fury iii	25899	Not Family car
9	14	455	225	4425	10	pontiac catalina	32882	Not Family car
10	15	390	190	3850	8.5	amc ambassador dpl	32617	Not Family car

Doing this, we can make a prediction: Given a car that we have not seen before, by checking with the description learned, we will be able to say whether it is a family car or not. Or we can do knowledge extraction: While taking car details as example, and the aim may be to understand what people expect from a family car. Among all features a car may have, the features that separate a family car from other type of cars are the price and engine power. These two attributes are the *inputs* to the class recognizer. Though one may think of other attributes such as weight and color that might be important for distinguishing among car types, we will consider only price and engine power to keep this example simple.

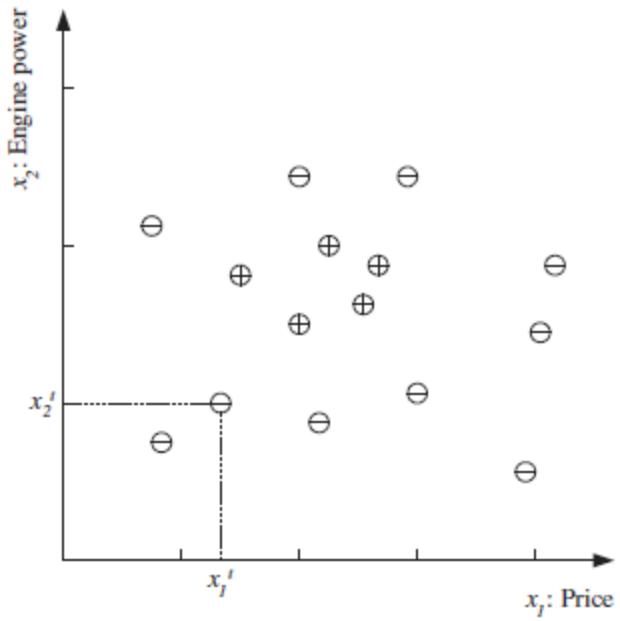


Figure Training set for the class of a "family car." Each data point corresponds to one example car, and the coordinates of the point indicate the price and engine power of that car. '+' denotes a positive example of the class (a family car), and '−' denotes a negative example (not a family car); it is another type of car.

Let us denote price as the first input attribute $x1$ and engine power as the second attribute $x2$ (e.g., engine volume in cubic centimeters). Thus we represent each car using two numeric values

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Each car is represented by such an ordered pair $(\mathbf{x}, r)$ and the training set contains N such examples

$$r = \begin{cases} 1 & \text{if } \boldsymbol{x} \text{ is a positive example} \\ 0 & \text{if } \boldsymbol{x} \text{ is a negative example} \end{cases}$$

where *t* indexes different examples in the set; it does not represent time or any such order.

$$\mathcal{X} = \{\boldsymbol{x}^t, r^t\}_{t=1}^{N}$$

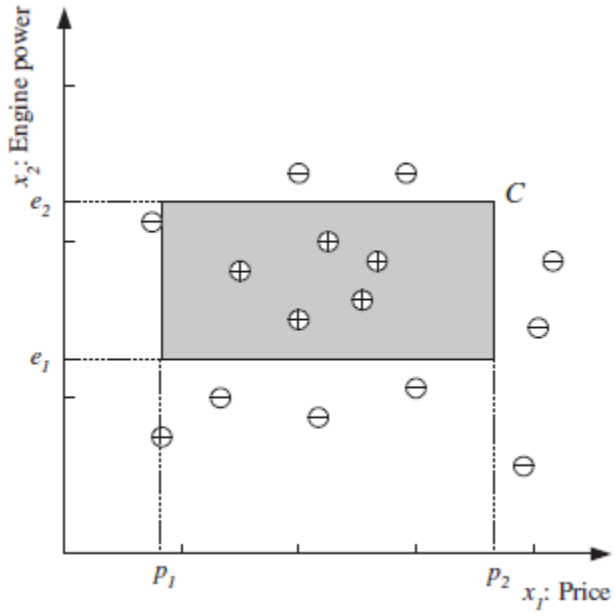


Figure Example of a hypothesis class. The class of family car is a rectangle in the price-engine power space.

In Our training data can now be plotted in the two-dimensional *(x*1, *x*2*)* space where each instance *t* is a data point at coordinates *(x*^{t} type, namely, positive versus negative, is given by *r* ^{t} After further discussions with the expert and the analysis of the data, we may have reason to believe that for a car to be a family car, its price and engine power should be in a certain range

$$(p_1 \leq \text{price} \leq p_2) \text{ AND } (e_1 \leq \text{engine power} \leq e_2)$$

for suitable values of *p*1, *p*2, *e*1, and *e*2. Equation thus assumes C to be a rectangle in the price-engine power space Equation fixes H, the *hypothesis class* from which we believe C is drawn, namely, the set of rectangles. The learning algorithm then finds the particular *hypothesis*, *h* ∈ H, specified by a particular quadruple of $(p_1^h, p_2^h, e_1^h, e_2^h)$ to approximate C as closely as possible.

Though the expert defines this hypothesis class, the values of the parameters are not known; that is, though we choose H, we do not know which particular *h*∈H is equal, or closest, to C.

But once we restrict our attention to this hypothesis class, learning the class reduces to the easier problem of finding the four parameters that define *h*. The aim is to find *h*∈H that is as similar as possible to C. Let us say the hypothesis *h* makes a prediction for an instance ***x*** such that

$$h(\boldsymbol{x}) = \begin{cases} 1 & \text{if } h \text{ classifies } \boldsymbol{x} \text{ as a positive example} \\ 0 & \text{if } h \text{ classifies } \boldsymbol{x} \text{ as a negative example} \end{cases}$$

In real life we do not know C***(x)***, so we cannot evaluate how well *h(**x**)* matches C***(x)***. What we have is the training set X, which is a small subset of the set of all possible ***x***. The *empirical error* is the proportion of training instances where *predictions* of *h* do not match the *required values* given in X. The error of hypothesis *h* given the training set X is

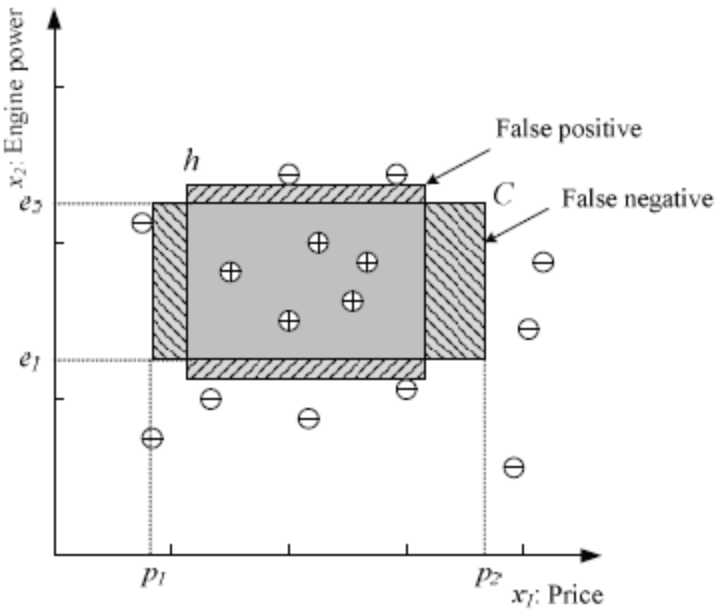
$$E(h|\mathcal{X}) = \sum_{t-1}^{N} 1(h(\boldsymbol{x}^t) \neq r^t)$$

where $1(a \neq b)$ is 1 if $a \neq b$ and is 0 if $a = b$

In our example, the hypothesis class H is the set of all possible rectangles. Each quadruple $(p_1^h, p_2^h, e_1^h, e_2^h)$ defines one hypothesis, *h*, from H, and we need to choose the best one, or in other words, we need to find the values of these four parameters given the training set, to include

all the positive examples and none of the negative examples.

This is the problem of *generalization*—that is, how well our hypothesis will correctly classify future examples that are not part of the training set.



C is the actual class and *h* is our induced hypothesis. The point where C is 1 but *h* is 0 is a false negative, and the point where C is 0 but *h* is 1 is a false positive. Other points—namely, true positives and true negatives—are correctly classified.

2. REGRESSION

Linear Regression Models

In classification, for an given an input, the output that is generated is Boolean; it is a yes/no answer. In **Regression** the output is a numeric value, what we would like to learn is not a class, but is a numeric function r.

Here we would like to write the numeric output, called the *dependent variable*, as a function of the input, called the *independent variable*. We assume that the numeric output is the sum of a deterministic function of the input and random noise:

$$r=f(x)+\epsilon$$

where $f(x)$ is the unknown function, which we would like to approximate by our estimator, $g(x|\theta)$, defined up to a set of parameters θ. If we assume that ϵ is zero mean Gaussian with constant variance $\sigma 2$, namely, $\epsilon \sim N(0,\sigma 2)$.

The simplest form of linear regression models are also linear functions of the input variables. However, we can obtain a much more useful class of functions by taking linear combinations of a fixed set of nonlinear functions of the input variables, known as *basis functions*. Such models are linear functions of the parameters, which gives them simple analytical properties, and yet can be nonlinear with respect to the input variables.

$$y(\mathbf{x},\mathbf{w}) = w0 + w1x1 + \ldots + wDxD$$

where $\mathbf{x} = (x1, \ldots, xD)^T$.

The linear function of the parameters $w0, \ldots, wD$. It is also, If one extend the class of models by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x})$$

where $\varphi j(\mathbf{x})$ are known as *basis functions*. By denoting the maximum value of the index j by $M - 1$, the total number of parameters in this model will be M.

Least Squares

One can maximize the likely-hood by minimizing sum of- squares error function. Here we show that this error function could be motivated as the maximum likelihood solution appears under an assumed Gaussian noise model. So consider the least squares approach, and its relation to maximum likelihood, in more detail. As before, we assume that the target variable t is given by a deterministic function $y(\mathbf{x},\mathbf{w})$ with additive Gaussian noise so that

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

where ϵ is a zero mean Gaussian random variable with precision (inverse variance) β.
Thus we can write

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}).$$

Recall that, if we assume a squared loss function, then the optimal prediction, for a new value of **x**, will be given by the conditional mean of the target variable. In the case of a Gaussian conditional distribution of the form, the conditional mean will be simply

$$\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\,\mathrm{d}t = y(\mathbf{x}, \mathbf{w}).$$

Note that the Gaussian noise assumption implies that the conditional distribution of t given **x** is unimodal, which may be inappropriate for some applications. An extension to mixtures of conditional Gaussian distributions, which permit multimodal conditional distributions, Now consider a data set of inputs $\mathbf{X} = \{\mathbf{x}1, \ldots, \mathbf{x}N\}$ with corresponding target values $t1, \ldots, tN$. We group the target variables $\{tn\}$ into a column vector that we denote by **t** where the typeface is chosen to distinguish it from a single observation of a multivariate target, which would be denoted **t**. Making the assumption that these data points are drawn independently from the distribution, we obtain the following expression for the likelihood function, which is a function of the adjustable parameters **w** and β, in the form.

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1})$$

Thus **x** will always appear in the set of conditioning variables, and so from now on we will drop the explicit **x** from expressions such as $p(\mathbf{t}|\mathbf{x},\mathbf{w}, \beta)$ in order to keep the notation uncluttered. Taking the logarithm of the likelihood function, and making use of the standard form for the univariate Gaussian,

$$\ln p(\mathbf{t}|\mathbf{w}, \beta) = \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi) - \beta E_D(\mathbf{w})$$

where the sum-of-squares error function is defined by

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2.$$

Having written down the likelihood function, we can use maximum likelihood to determine **w** and β. Consider first the maximization with respect to **w**. we see that maximization of the likelihood function under a conditional Gaussian noise distribution for a linear model is equivalent to minimizing a sum-of- squares error function given by *ED*(**w**). The gradient of the log likelihood function takes the form.

$$\nabla \ln p(\mathbf{t}|\mathbf{w}, \beta) = \sum_{n=1}^{N}\left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\}\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}$$

Draw back

Batch techniques, such as the maximum likelihood solution, which involve processing the entire training set in one go, can be computationally costly for large data sets.

Sequential Gradient Descent

if the data set is sufficiently large, it may be worthwhile to use *sequential* algorithms, also known as *on-line* algorithms, in which the data points are considered one at a time, and the model parameters updated after each such presentation. Sequential learning is also appropriate for realtime applications in which the data observations are arriving in a continuous stream, and predictions must be made before all of the data points are seen. We can obtain a sequential learning algorithm by applying the technique of *stochastic gradient descent*, also known as *sequential gradient descent*, as follows. If the error function comprises a sum over data points $E = \sum_n E_n$ then after presentation of pattern *n*, the stochastic gradient descent algorithm updates the parameter vector **w** using

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E_n$$

where τ denotes the iteration number, and η is a learning rate parameter. We shall discuss the choice of value for η shortly. The value of **w** is initialized to some starting vector **w**(0). For the case of the sum-of-squares error function (3.12), this gives

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \eta(t_n - \mathbf{w}^{(\tau)\mathrm{T}}\boldsymbol{\phi}_n)\boldsymbol{\phi}_n$$

where $\boldsymbol{\varphi}n = \boldsymbol{\varphi}(\mathbf{x}n)$. This is known as *least-mean-squares* or the *LMS algorithm*. The value of η needs to be chosen with care to ensure.

Single & multiple variables

So far, we have considered the case of a single target variable *t*. In some applications, we may wish to predict $K > 1$ target variables, which we denote collectively by the target vector **t**. This could be done by introducing a different set of basis functions for each component of **t**, leading to multiple, independent regression problems. However, a more interesting, and more common, approach is to use the same set of
basis functions to model all of the components of the target vector so that

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})$$

where **y** is a *K*-dimensional column vector, **W**is an $M \times K$ matrix of parameters, and $\boldsymbol{\varphi}(\mathbf{x})$ is an *M*-dimensional column vector with elements $\varphi j(\mathbf{x})$, with $\varphi_0(\mathbf{x}) = 1$ as before. Suppose we take the conditional distribution of the target vector to be an isotropic Gaussian of the form

$$p(\mathbf{t}|\mathbf{x}, \mathbf{W}, \beta) = \mathcal{N}(\mathbf{t}|\mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \beta^{-1}\mathbf{I}).$$

If we have a set of observations **t**1, . . . , **t***N*, we can combine these into a matrix **T** of size $N \times K$ such that the *n*th row is given by **t**T*n* . Similarly, we can combine the input vectors **x**1, . . . , **xn** into a matrix **X**. The log likelihood function is then given by

$$\ln p(\mathbf{T}|\mathbf{X}, \mathbf{W}, \beta) = \frac{NK}{2}\ln\left(\frac{\beta}{2\pi}\right) - \frac{\beta}{2}\sum_{n=1}^{N}\left\|\mathbf{t}_n - \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\|^2.$$

As before, we can maximize this function with respect to **W**, giving

$$\mathbf{W}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{T}.$$

If we examine this result for each target variable *tk*, we have

$$\mathbf{w}_k = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}_k = \boldsymbol{\Phi}^{\dagger}\mathbf{t}_k$$

where **t***k* is an *N*-dimensional column vector with components *tnk* for $n = 1, \ldots N$. Thus the solution to the regression problem decouples between the different target variables, and we need only compute a single pseudo-inverse matrix $\boldsymbol{\Phi}\dagger$, which is shared by all of the vectors **w***k*.

The extension to general Gaussian nose distributions having arbitrary covariance matrices is straightforward. Again, this leads to a decoupling into *K* independent regression problems. This result is unsurprising because the parameters **W** define only the mean of the Gaussian noise distribution, and we know from that the maximum likelihood solution for the

mean of a multivariate Gaussian is independent of the covariance. From now on, we shall therefore consider a single target variable t for simplicity.

Bayesian Linear Regression

Maximum likelihood for setting the parameters of a linear regression model, we have seen that the effective model complexity, governed by the number of basis functions, needs to be controlled according to the size of the data set. Adding a regularization term to the log likelihood function means the effective model complexity can then be controlled by the value of the regularization coefficient, although the choice of the number and form of the basis functions is of course still important in determining the overall behaviour of the model.

This leaves the issue of deciding the appropriate model complexity for the particular problem, which cannot be decided simply by maximizing the likelihood function, because this always leads to excessively complex models and over-fitting. Independent hold-out data can be used to determine model complexity, but this can be both computationally expensive and wasteful of valuable data. We therefore turn to a Bayesian treatment of linear regression, which will avoid the over-fitting problem of maximum likelihood, and which will also lead to automatic methods of determining model complexity using the training data alone. Again, for simplicity we will focus on the case of a single target variable t.

Bayesian treatment of linear regression by introducing a prior probability distribution over the model parameters $\mathbf{w}$. For the moment, we shall treat the noise precision parameter β as a known constant. First note that the likelihood function $p(\mathbf{t}|\mathbf{w})$ is the exponential of a quadratic function of $\mathbf{w}$. The corresponding conjugate prior is therefore given by a Gaussian distribution of the form

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_0, \mathbf{S}_0)$$

having mean $\mathbf{m}_0$ and covariance $\mathbf{S}_0$ the likelihood function and the prior. Due to the choice of a conjugate Gaussian prior distribution, the posterior will also be Gaussian. We can evaluate this distribution by the usual procedure of completing the square in the exponential, and then finding the normalization coefficient using the standard result for a normalized Gaussian. which allows us to write down the posterior distribution directly in the form

$$p(\mathbf{w}|\mathbf{t}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N)$$

$$\begin{aligned} \mathbf{m}_N &= \mathbf{S}_N \left(\mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\mathbf{\Phi}^{\mathrm{T}}\mathbf{t}\right) \\ \mathbf{S}_N^{-1} &= \mathbf{S}_0^{-1} + \beta\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}. \end{aligned}$$

Note that because the posterior distribution is Gaussian, its mode coincides with its mean. Thus the maximum posterior weight vector is simply given by **w**MAP = **m***N*. If we consider an infinitely broad prior **S**0 = $\alpha-1\mathbf{I}$ with $\alpha \rightarrow 0$, the mean **m***N* of the posterior distribution reduces to the maximum likelihood value **w**ML. Similarly, if $N = 0$, then the posterior distribution reverts to the prior. Furthermore, if data points arrive sequentially, then the posterior distribution at any stage acts as the prior distribution for the subsequent data point, such that the new posterior distribution. Specifically, we consider a zero-mean isotropic Gaussian governed by a single precision parameter α so that

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I})$$

and the corresponding posterior distribution over **w**

$$\begin{aligned} \mathbf{m}_N &= \beta\mathbf{S}_N\mathbf{\Phi}^{\mathrm{T}}\mathbf{t} \\ \mathbf{S}_N^{-1} &= \alpha\mathbf{I} + \beta\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}. \end{aligned}$$

The log of the posterior distribution is given by the sum of the log likelihood and the log of the prior and, as a function of **w**, takes the form

$$\ln p(\mathbf{w}|\mathbf{t}) = -\frac{\beta}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n)\}^2 - \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + \text{const.}$$

Maximization of this posterior distribution with respect to **w** is therefore equivalent to the minimization of the sum-of-squares error function with the addition of a quadratic regularization term, corresponding to with $\lambda = \alpha/\beta$.

LINEAR CLASSIFICATION MODELS

The goal in classification is to take an input vector **x** and to assign it to one of *K* discrete classes *Ck* where $k = 1, \ldots, K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. The input space is thereby divided into *decision regions* whose boundaries are called *decision boundaries* or *decision surfaces*. So the decision surfaces are linear functions of the input vector **x** and hence are

defined by ($D - 1$)-dimensional hyperplanes within the D-dimensional input space. Datasets whose classes can be separated exactly by linear decision surfaces are said to be *linearly separable*.

In the case of classification, there are various ways of using target values to represent class labels. For probabilistic models, the most convenient, in the case of two-class problems, is the binary representation in which there is a single target variable $t \in \{0, 1\}$ such that $t = 1$ represents class $C1$ and $t = 0$ represents class $C2$. We can interpret the value of t as the probability that the class is $C1$, with the values of probability taking only the extreme values of 0 and 1. For $K > 2$ classes, it is convenient to use a 1-of-K coding scheme in which $\mathbf{t}$ is a vector of length K such that if the class is Cj , then all elements tk of $\mathbf{t}$ are zero except element tj , which takes the value 1. For instance, if we have $K = 5$ classes, then a pattern from class 2 would be given the target vector.

Again, we can interpret the value of tk as the probability that the class is Ck. For non-probabilistic models, alternative choices of target variable representation will sometimes prove convenient.

a. Discriminant Functions

b. Probabilistic Discriminative Models

c. Probabilistic Generative Models

a. Discriminant Functions

A discriminant is a function that takes an input vector $\mathbf{x}$ and assigns it to one of K classes, denoted Ck. Here, we shall restrict attention to *linear discriminants*, namely those for which the decision surfaces are hyperplanes. To simplify the discussion, we consider first the case of two classes and then investigate the extension to $K > 2$ classes.

The simplest representation of a linear discriminant function is obtained by taking a linear function of the input vector so that

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0$$

where $\mathbf{w}$ is called a *weight vector*, and $w0$ is a *bias* (not to be confused with bias in the statistical sense). The negative of the bias is sometimes called a *threshold*. An input vector $\mathbf{x}$ is assigned to class $C1$ if $y(\mathbf{x}) \geq 0$ and to class $C2$ otherwise. The corresponding decision boundary is therefore defined by the relation $y(\mathbf{x}) = 0$, which corresponds to a ($D - 1$)-dimensional hyperplane within the D-dimensional input space. Consider two points $\mathbf{x}_A$ and $\mathbf{x}_B$ both of which lie on the decision surface. Because $y(\mathbf{x}_A) = y(\mathbf{x}_B) = 0$, we have $\mathbf{w}^{\mathrm{T}}(\mathbf{x}_A - \mathbf{x}_B) = 0$ and hence the

vector **w** is orthogonal to every vector lying within the decision surface, and so**w** determines the orientation of the decision surface. Similarly, if **x** is a point on the decision surface, then $y(\mathbf{x}) = 0$, and so the normal distance from the origin to the decision surface is given by

$$\frac{\mathbf{w}^{\mathrm{T}}\mathbf{x}}{\|\mathbf{w}\|} = -\frac{w_0}{\|\mathbf{w}\|}.$$

We therefore see that the bias parameter *w*0 determines the location of the decision surface. These properties are illustrated for the case of $D = 2$ in Figure Furthermore, we note that the value of $y(\mathbf{x})$ gives a signed measure of the perpendicular distance r of the point **x** from the decision surface. To see this, consider

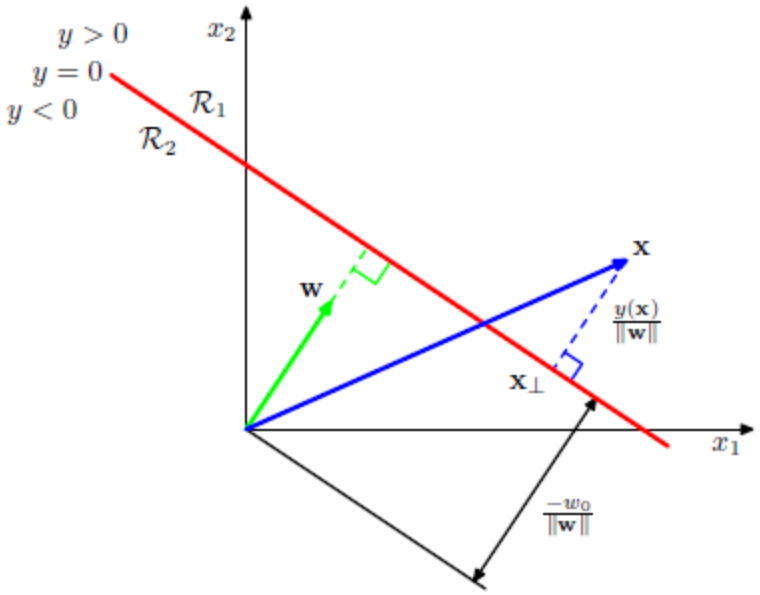


Illustration of the geometry of a linear discriminant function in two dimensions. The decision surface, shown in red, is perpendicular to **w**, and its displacement from the origin is controlled by the bias parameter w_0.

Also, the signed orthogonal distance of a general point **x** from the decision surface is given by $y(\mathbf{x})/\|\mathbf{w}\|$. an arbitrary point **x** and let **x**⊥ be its orthogonal projection onto the decision surface,

so that

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{\|\mathbf{w}\|}.$$

Multiplying both sides of this result by**w**T and adding *w*0, and making use of $y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0$ and $y(\mathbf{x}\perp) = \mathbf{w}\mathrm{T}\mathbf{x}\perp + w_0 = 0$, we have

$$r = \frac{y(\mathbf{x})}{\|\mathbf{w}\|}.$$

This result is illustrated in Figure

Least squares for classification

While considering models that has linear functions of the parameters, we saw that the minimization of a sum-of-squares error function led to a simple closed-form solution for the parameter values. It is therefore tempting to see if we can apply the same formalism to classification problems. Consider a general classification problem with *K* classes, with a 1-of-*K* binary coding scheme for the target vector **t**. One justification for using least squares in such a context is that it approximates the conditional expectation E[**t**|**x**] of the target values given the input vector. For the binary coding scheme, this conditional expectation is given by the vector of posterior class probabilities. Unfortunately, however, these probabilities are typically approximated rather poorly, indeed the approximations can have values outside the range (0, 1), due to the limited flexibility of a linear model.

Each class *Ck* is described by its own linear model so that

$$y_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0}$$

where $k = 1, \ldots, K$. We can conveniently group these together using vector notation so that

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}}$$

where , $\widetilde{\mathrm{W}}$ is a matrix whose *k*th column comprises the $D + 1$-dimensional vector $\widetilde{\mathbf{w}}_k = (w_{k0}, \mathbf{w}_k^{\mathrm{T}})^{\mathrm{T}}$ and $\widetilde{\mathrm{X}}$ is the corresponding augmented input vector $(1, \mathbf{x}^{\mathrm{T}})^{\mathrm{T}}$ with a dummy input $X_0 = 1$. The new input **x** is then assigned to the class for which the output $y_k = \widetilde{\mathrm{W}}_x^T \widetilde{\mathrm{X}}$ is largest. We now determine the parameter matrix , $\widetilde{\mathrm{W}}$ by minimizing a sum-of-squares error function. Consider a training data set *{***x***n*, **t***n}* where $n = 1, \ldots, N$, and define a matrix **T** whose *n*th row is the vector $\mathbf{t}^{\mathrm{T}}{}_n$ together with a matrix $\widetilde{\mathrm{X}}$ whose *n*th row is $\widetilde{\mathrm{x}}^{\mathrm{T}}n$. The sum-of-squares error function can then be written as

$$E_D(\widetilde{\mathbf{W}}) = \frac{1}{2}\mathrm{Tr}\left\{(\widetilde{\mathbf{X}}\widetilde{\mathbf{W}} - \mathbf{T})^{\mathrm{T}}(\widetilde{\mathbf{X}}\widetilde{\mathbf{W}} - \mathbf{T})\right\}.$$

Setting the derivative with respect to, $\widetilde{\mathrm{W}}$ to zero, and rearranging, we then obtain the solution for , $\widetilde{\mathrm{W}}$ in the form

$$\widetilde{\mathbf{W}} = (\widetilde{\mathbf{X}}^{\mathrm{T}}\widetilde{\mathbf{X}})^{-1}\widetilde{\mathbf{X}}^{\mathrm{T}}\mathbf{T} = \widetilde{\mathbf{X}}^{\dagger}\mathbf{T}$$

where $\widetilde{\mathbf{X}}^{\dagger}$ is the pseudo-inverse of the matrix $\widetilde{\mathbf{X}}$, So the discriminant function takes the form

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{T}^{\mathrm{T}}\left(\widetilde{\mathbf{X}}^{\dagger}\right)^{\mathrm{T}}\widetilde{\mathbf{x}}.$$

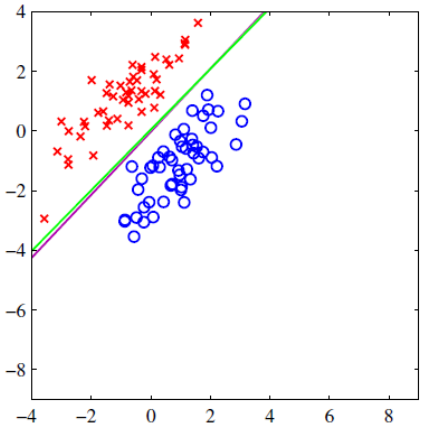
An interesting property of least-squares solutions with multiple target variables is that if every target vector in the training set satisfies some linear constraint

$$\mathbf{a}^{\mathrm{T}}\mathbf{t}_n + b = 0$$

for some constants **a** and *b*, then the model prediction for any value of **x** will satisfy the same constraint so that

$$\mathbf{a}^{\mathrm{T}}\mathbf{y}(\mathbf{x}) + b = 0.$$

Thus if we use a 1-of-*K* coding scheme for *K* classes, then the predictions made by the model will have the property that the elements of **y**(**x**) will sum to 1 for any value of **x**. However, this summation constraint alone is not sufficient to allow the model outputs to be interpreted as probabilities because they are not constrained to lie within the interval (0, 1). The least-squares approach gives an exact closed-form solution for the discriminant function parameters. However, even as a discriminant function it suffers from some severe problems. We have already seen that least-squares solutions lack robustness to outliers, and this applies equally to the classification application, as illustrated in Figure



The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve)

b. Probabilistic Discriminative Models

For the two-class classification problem, we have seen that the posterior probability of class $C1$ can be written as a logistic sigmoid acting on a linear function of $\mathbf{x}$, for a wide choice of class-conditional distributions $p(\mathbf{x}|Ck)$. Similarly, for the multiclass case, the posterior probability of class Ck is given by a softmax transformation of a linear function of $\mathbf{x}$. For specific choices of the class-conditional densities $p(\mathbf{x}|Ck)$, we have used maximum likelihood to determine the parameters of the densities as well as the class priors $p(Ck)$ and then used Bayes' theorem to find the posterior class probabilities.

However, an alternative approach is to use the functional form of the generalized linear model explicitly and to determine its parameters directly by using maximum likelihood. We shall see that there is an efficient algorithm finding such solutions known as *iterative reweighted least squares*, or *IRLS*.

The indirect approach to finding the parameters of a generalized linear model, by fitting class-conditional densities and class priors separately and then applying Bayes' theorem, represents an example of *generative* modelling, because we could take such a model and generate synthetic data by drawing values of $\mathbf{x}$ from the marginal distribution $p(\mathbf{x})$. In the direct approach, we are maximizing a likelihood function defined through the conditional distribution $p(Ck|\mathbf{x})$, which represents a form of *discriminative* training. One advantage of the discriminative approach is that there will typically be fewer adaptive parameters to be determined, as we shall see shortly. It may also lead to improved predictive performance, particularly when the class-conditional density assumptions give a poor approximation to the true distributions.

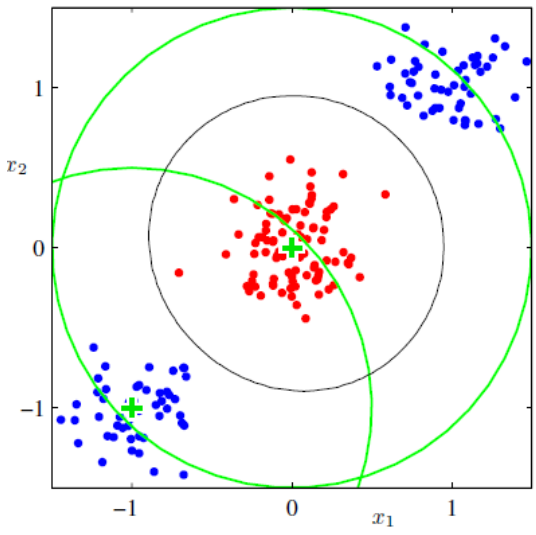


Figure of the role of nonlinear basis functions in linear classification models. The left plot shows the original input space ($x1$, $x2$) together with data points from two classes labelled red and blue. Two 'Gaussian' basis functions $\varphi 1(\mathbf{x})$ and $\varphi 2(\mathbf{x})$ are defined in this space with centres shown by the green crosses and with contours shown by the green circles.

Logistic regression

We begin our treatment of generalized linear models by considering the problem of two-class classification. In generative approaches, we saw that under rather general assumptions, the posterior probability of class $C1$ can be written as a logistic sigmoid acting on a linear function of the feature vector

$$p(\mathcal{C}_1|\phi) = y(\phi) = \sigma\left(\mathbf{w}^{\mathrm{T}}\phi\right)$$

with $p(C2|\boldsymbol{\varphi}) = 1 - p(C1|\boldsymbol{\varphi})$. Here $\sigma(\cdot)$ is the *logistic sigmoid* function. In the terminology of statistics, this model is known as *logistic regression*, although it should be emphasized that this is a model for classification rather than regression. For an M-dimensional feature space $\boldsymbol{\varphi}$, this model has M adjustable parameters. By contrast, if we had fitted Gaussian class conditional densities using maximum likelihood, we would have used $2M$ parameters for the means and $M(M + 1)/2$ parameters for the (shared) covariance matrix. Together with the class prior $p(C1)$, this gives a total of $M(M+5)/2+1$ parameters, which grows quadratically with M, in contrast to the linear dependence on M of the number of parameters in logistic regression. For large values of M, there is a clear advantage in working with the logistic regression model directly.

We now use maximum likelihood to determine the parameters of the logistic regression model. To do this, we shall make use of the derivative of the logistic sigmoid function, which can conveniently be expressed in terms of the sigmoid function itself

$$\frac{d\sigma}{da} = \sigma(1 - \sigma)$$

For a data set $\{\boldsymbol{\varphi}n, tn\}$, where $tn \in \{0, 1\}$ and $\boldsymbol{\varphi}n = \boldsymbol{\varphi}(\mathbf{xn})$, with $n = 1, \ldots, N$, the likelihood function can be written

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

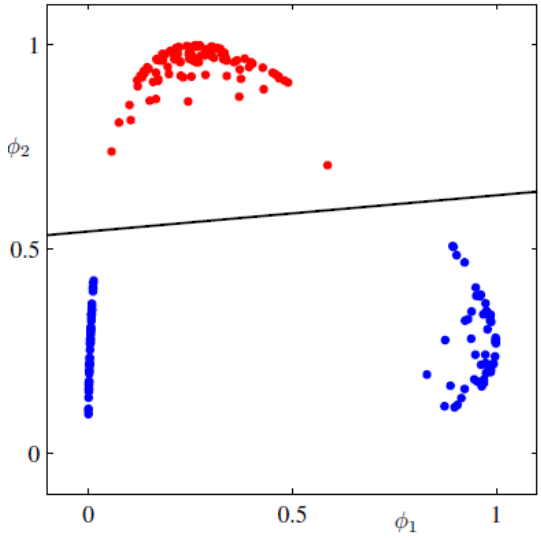
where **t** = (*t*1, . . . , *tN*)T and *yn* = *p*(*C*1|***φ****n*). As usual, we can define an error function by taking the negative logarithm of the likelihood, which gives the *crossentropy* error function in the form

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

where *yn* = *σ*(*an*) and *an* = **w**T***φ****n***. Taking the gradient of the error function with respect to **w**, we obtain

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\phi_n$$

We see that the factor involving the derivative of the logistic sigmoid has cancelled, leading to a simplified form for the gradient of the log likelihood. In particular, the contribution to the gradient from data point *n* is given by the 'error' *yn* − *tn* between the target value and the prediction of the model, times the basis function vector ***φ****n*. Furthermore, this takes precisely the same form as the gradient of the sum-of-squares error function for the linear regression model.



The Figure plot shows the corresponding feature space (*φ*1, *φ*2) together with the linear decision boundary obtained given by a logistic regression model of the form. This corresponds to a nonlinear decision boundary in the original input space, shown by the black curve in the left-hand plot.

c. Probabilistic Generative Models

In this approach, we model the class-conditional densities $p(\mathbf{x}|Ck)$, as well as the class priors $p(Ck)$, and then use these to compute posterior probabilities $p(Ck|\mathbf{x})$ through Bayes' theorem. Consider first of all the case of two classes. The posterior probability for class $C1$ can be written as.

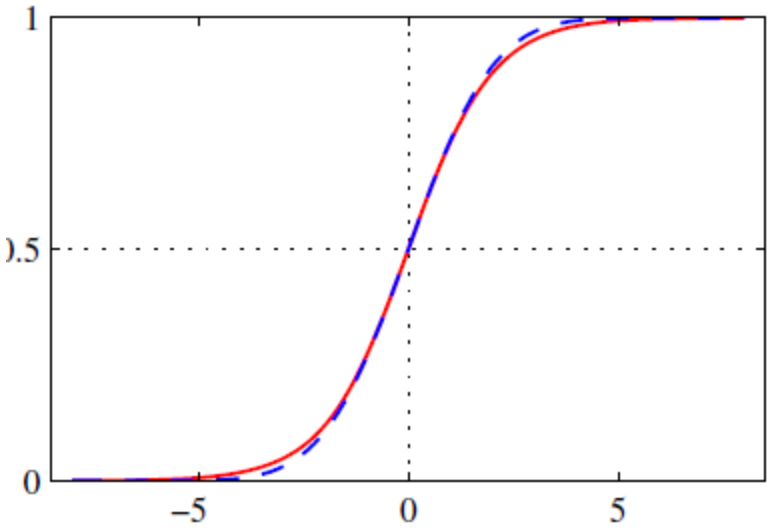


Figure Plot of the logistic sigmoid function $\sigma(a)$, shown in red, together with the scaled probit function $\Phi(\lambda a)$, for $\lambda 2 = \pi/8$, shown in dashed blue, The scaling factor $\pi/8$ is chosen so that the derivatives of the two curves are equal for $a = 0$.

$$p(\mathcal{C}_1|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} = \frac{1}{1+\exp(-a)} = \sigma(a)$$

$$a = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}$$

$$\sigma(a) = \frac{1}{1+\exp(-a)}$$

and $\sigma(a)$ is the *logistic sigmoid* function defined by which is plotted in Figure. The term 'sigmoid' means S-shaped. This type of function is sometimes also called a 'squashing function' because it maps the whole real axis into a finite interval and It plays an important role in many classification algorithms. It satisfies the following symmetry property

$$\sigma(-a) = 1 - \sigma(a)$$

as is easily verified. The inverse of the logistic sigmoid is given by

$$a = \ln\left(\frac{\sigma}{1-\sigma}\right)$$

and is known as the *logit* function. It represents the log of the ratio of probabilities ln [$p(C1|\mathbf{x})/p(C2|\mathbf{x})$] for the two classes, also known as the *log odds*.

we have simply rewritten the posterior probabilities in an equivalent form, and so the appearance of the logistic sigmoid may seem rather vacuous. it will have significance provided $a(\mathbf{x})$ takes a simple functional
form.

$$\begin{aligned} p(\mathcal{C}_k|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} \\ &= \frac{\exp(a_k)}{\sum_j \exp(a_j)} \end{aligned}$$

$$a_k = \ln p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k).$$

MAXIMUM MARGIN CLASSIFIERS

Here the two-class classification problem using linear models of the form

$$y(\mathbf{x}) = \mathbf{w}\mathrm{T}\boldsymbol{\varphi}(\mathbf{x}) + b$$

where $\boldsymbol{\varphi}(\mathbf{x})$ denotes a fixed feature-space transformation, and we have made the bias parameter b explicit. Note that we shall shortly introduce a dual representation expressed in terms of kernel functions, which avoids having to work explicitly in feature space. The training data set comprises N input vectors $\mathbf{x}1, \ldots, \mathbf{xn}$, with corresponding target values $t1, \ldots, tN$ where $tn \in \{-1, 1\}$, and new data points $\mathbf{x}$ are classified according to the sign of $y(\mathbf{x})$.

We shall assume for the moment that the training data set is linearly separable in feature space, so that by definition there exists at least one choice of the parameters $\mathbf{w}$ and b such that

a function satisfies $y(\mathbf{xn}) > 0$ for points having $tn = +1$ and $y(\mathbf{x}n) < 0$ for points having $tn = -1$, so that $tny(\mathbf{x}n) > 0$ for all training data points.

There may of course exist many such solutions that separate the classes exactly. If there are multiple solutions all of which classify the training data set exactly, then we should try to find the one that will give the smallest generalization error. The support vector machine approaches this problem through the concept of the *margin*, which is defined to be the smallest distance between the decision boundary and any of the samples, as illustrated in Figure In support vector machines the decision boundary is chosen to be the one for which the margin is maximized. The maximum margin solution can be motivated using *computational learning theory*, also known as *statistical learning theory*. They first model the distribution over input vectors **x** for each class using a Parzen density estimator with Gaussian kernels.

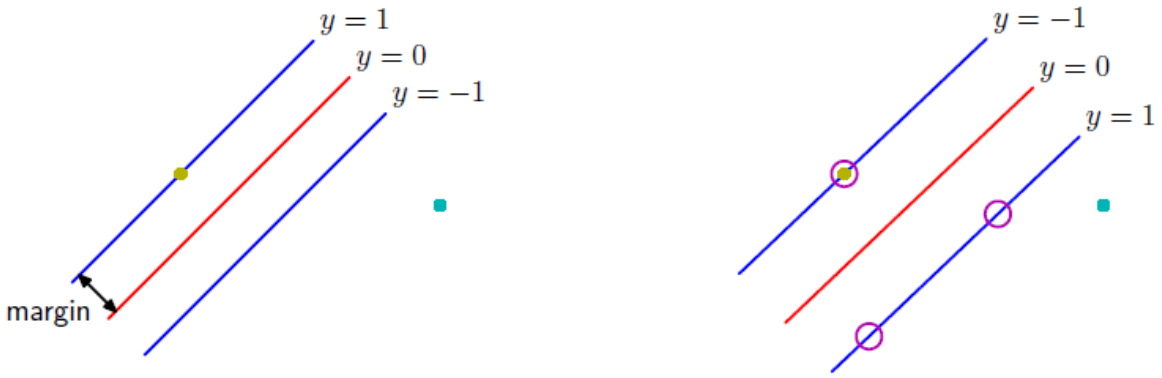


Figure The margin is defined as the perpendicular distance between the decision boundary and the closest of the data points, as shown on the left figure. Maximizing the margin leads to a particular choice of decision boundary, as shown on the right. The location of this boundary is determined by a subset of the data points, known as support vectors, which are indicated by the circles. having a common parameter $\sigma 2$. Together with the class priors, this defines an optimal misclassification-rate decision boundary. However, instead of using this optimal boundary, they determine the best hyperplane by minimizing the probability of error relative to the learned density model. In the limit $\sigma 2 \rightarrow 0$, the optimal hyperplane is shown to be the one having maximum margin. The intuition behind this result is that as $\sigma 2$ is reduced, the hyperplane is increasingly dominated by nearby data points relative to more distant ones. In the limit, the hyperplane becomes independent of data points that are not support vectors.

From the above figure that marginalization with respect to the prior distribution of the parameters in a Bayesian approach for a simple linearly separable dataset leads to a decision

boundary that lies in the middle of the region separating the data points. The large margin solution has similar behaviour.

The perpendicular distance of a point **x** from a hyperplane defined by $y(\mathbf{x}) = 0$ where $y(\mathbf{x})$ is given by $|y(\mathbf{x})|/\|\mathbf{w}\|$. Furthermore, we are only interested in solutions for which all data points are correctly classified, so that $t_n y(\mathbf{x}_n) > 0$ for all n. Thus the distance of a point $\mathbf{x}_n$ to the decision surface is given by

$$\frac{t_n y(\mathbf{x}_n)}{\|\mathbf{w}\|} = \frac{t_n(\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) + b)}{\|\mathbf{w}\|}$$

The margin is given by the perpendicular distance to the closest point $\mathbf{x}_n$ from the data set, and we wish to optimize the parameters **w** and b in order to maximize this distance. Thus the maximum margin solution is found by solving

$$\arg\max_{\mathbf{w},b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_n \left[t_n \left(\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) + b \right) \right] \right\}$$

where we have taken the factor $1/\|\mathbf{w}\|$ outside the optimization over n because **w** does not depend on n. Direct solution of this optimization problem would be very complex, and so we shall convert it into an equivalent problem that is much easier to solve. To do this we note that if we make the rescaling $\mathbf{w} \to \kappa\mathbf{w}$ and $b \to \kappa b$, then the distance from any point $\mathbf{x}_n$ to the decision surface, given by $t_n y(\mathbf{x}_n)/\|\mathbf{w}\|$, is unchanged. We can use this freedom to set

$$t_n \left(\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) + b \right) = 1$$

for the point that is closest to the surface. In this case, all data points will satisfy the constraints

$$t_n \left(\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) + b \right) \geqslant 1, \qquad n = 1, \ldots, N.$$

This is known as the canonical representation of the decision hyperplane. In the case of data points for which the equality holds, the constraints are said to be *active*, whereas for the remainder they are said to be *inactive*. By definition, there will always be at least one active constraint, because there will always be a closest point, and once the margin has been maximized there will be at least two active constraints. The optimization problem then simply requires that we maximize $\|\mathbf{w}\|^{-1}$, which is equivalent to minimizing $\|\mathbf{w}\|^2$, and so we have to solve the optimization problem

$$\arg\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

***SUPPORT VECTOR MACHINE* (SVM)**

SVM became popular in some years ago for solving problems in classification. An important property of support vector machines is that the determination of the model parameters corresponds to a convex optimization problem, and so any local solution is also a global optimum. Because the discussion of support vector machines makes extensive use of Lagrange multipliers,

later generalized under the name *kernel machine*, has been popular in recent years for a number of reasons:

1. It is a discriminant-based method and uses Vapnik's principle to never solve a more complex problem as a first step before the actual problem. For example, in classification, when the task is to learn the discriminant, it is not necessary to estimate where the class densities $p(\mathbf{x}|C_i)$ or the exact posterior probability values $P(C_i|\mathbf{x})$; we only need to estimate where the class boundaries lie, that is, $\mathbf{x}$ where $P(C_i|\mathbf{x}) = P(C_j|\mathbf{x})$. Similarly, for outlier detection, we do not need to estimate the full density $p(\mathbf{x})$; we only need to find the boundary separating those $\mathbf{x}$ that have low $p(\mathbf{x})$, that is, $\mathbf{x}$ where $p(\mathbf{x}) < \theta$, for some threshold $\theta \in (0, 1)$.
2. After training, the parameter of the linear model, the weight vector, can be written down in terms of a subset of the training set, which are the so-called ***support vectors***. In classification, these are the cases that are close to the boundary and as such, knowing them allows knowledge extraction: Those are the uncertain or erroneous cases that lie in the vicinity of the boundary between two classes. Their number gives us an estimate of the generalization error.
3. the output is written as a sum of the influences of support vectors and these are given by *kernel functions* that are application-specific measures of similarity between data instances.
4. Data points are represented as vectors, For example, $G1$ and $G2$ may be two graphs and $K(G1,G2)$ may correspond to the number of shared paths, which we can calculate without needing to represent $G1$ or $G2$ explicitly as vectors.
5. Kernel-based algorithms are formulated as convex optimization problems, and there is a single optimum that we can solve for analytically. Therefore we are no longer

bothered with heuristics for learning rates, initializations, checking for convergence, and such the case of classification, and then generalize to ranking, outlier (novelty) detection, and then dimensionality reduction. We see that in all cases basically we have the similar quadratic program template to maximize the separability, or *margin*, of instances subject to a constraint of the smoothness of solution. Solving for it, we get the support vectors. The kernel function defines the space according to its notion of similarity and a kernel function is good if we have better separation in its corresponding space.

Optimal Separating Hyperplane

Let us start again with two classes and use labels −1/ + 1 for the two classes. The sample is $\mathcal{X} = \{\boldsymbol{x}^t, r^t\}$ where $r^t = +1$ if $\boldsymbol{x}^t \in$ C1 and $r^t = -1$ if $\boldsymbol{x}^t \in$ C2. We would like to find $\boldsymbol{w}$ and $w0$ such that

$$\begin{aligned} \boldsymbol{w}^T\boldsymbol{x}^t + w_0 \geq +1 \quad &\text{for} \quad r^t = +1 \\ \boldsymbol{w}^T\boldsymbol{x}^t + w_0 \leq -1 \quad &\text{for} \quad r^t = -1 \end{aligned}$$

which can be rewritten as

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq +1$$

Note that we do not simply require

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq 0$$

Not only do we want the instances to be on the right side of the hyperplane, but we also want them some distance away, for better generalization. The distance from the hyperplane to the instances closest to it on either side is called the ***margin***, which we want to maximize for best generalization.

It is better to take a rectangle halfway between *S* and *G*, to get a breathing space. This is so that in case noise shifts a test instance slightly, it will still be on the right side of the boundary.

Similarly, now that we are using the hypothesis class of lines, the *optimal separating hyperplane* is the one that maximizes the margin. the distance of $\boldsymbol{x}^t$ to the discriminant is

$$\frac{|\boldsymbol{w}^T\boldsymbol{x}^t + w_0|}{\|\boldsymbol{w}\|}$$

which, when $r^t \in \{-1,+1\}$, can be written as

$$\frac{r^t(\mathbf{w}^T\mathbf{x}^t + w_0)}{\|\mathbf{w}\|} \geq \rho, \forall t$$

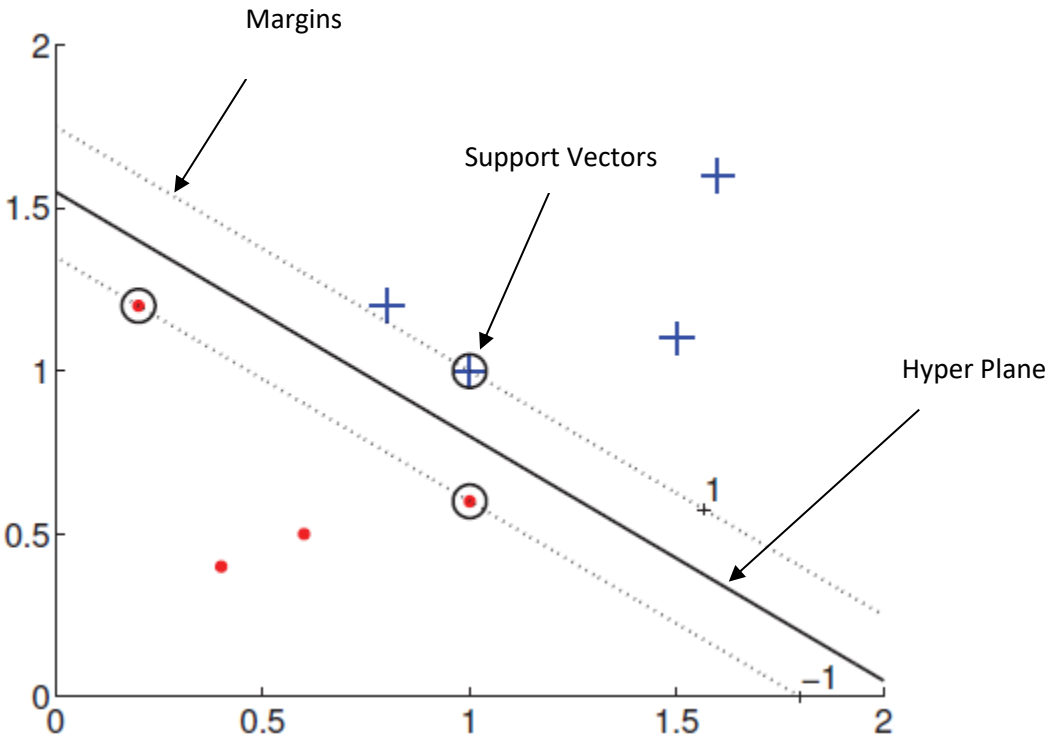
We would like to maximize ρ but there are an infinite number of solutions that we can get by scaling $\mathbf{w}$ and for a unique solution, we fix $\rho\|\mathbf{w}\| = 1$ and thus, to maximize the margin, we minimize $\|\mathbf{w}\|$

$$\min \frac{1}{2}\|\mathbf{w}\|^2 \text{ subject to } r^t(\mathbf{w}^T\mathbf{x}^t + w_0) \geq +1, \forall t$$

This is a standard quadratic optimization problem, whose complexity depends on d, and it can be solved directly to find $\mathbf{w}$ and $w0$. Then, on both sides of the hyperplane, there will be instances that are $1/\|\mathbf{w}\|$ away from the hyperplane and the total margin will be $2/\|\mathbf{w}\|$.

In finding the optimal hyperplane, we can convert the optimization problem to a form whose complexity depends on N, the number of training instances, and not on d. Another advantage of this new formulation is that it will allow us to rewrite the basis functions in terms of kernel functions

$$L_p = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_t \alpha^t r^t(\mathbf{w}^T\mathbf{x}^t + w_0) + \sum_t \alpha^t$$



For a two-class problem where the instances of the classes are shown by plus signs and dots, the thick line is the boundary and the dashed lines define the margins on either side. Circled instances are the support vectors

$$w_0 = r^t - \boldsymbol{w}^T \boldsymbol{x}^t$$

For numerical stability, it is advised that this be done for all support vectors and an average be taken. The discriminant thus found is called the *support vector machine* (SVM)

Decision Tree Classifier

The process of selecting a specific model, given a new input **x**, can be described by a sequential decision making process corresponding to the traversal of a binary tree (one that splits into two branches at each node). The tree-based framework called *classification and regression trees*, Figure shows an illustration of a recursive binary partitioning of the input space, along with the corresponding tree structure.

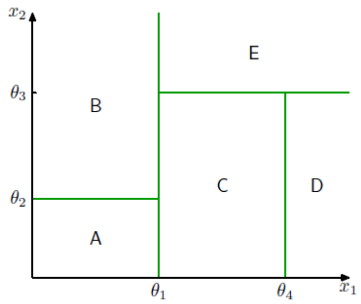
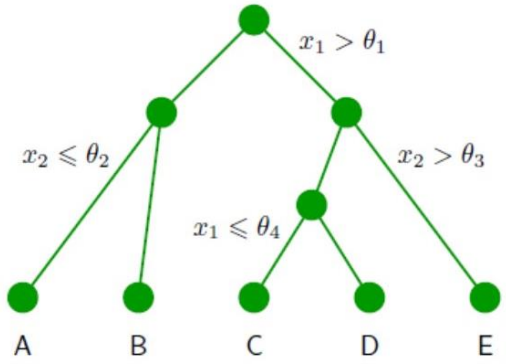


Figure shows an illustration of a recursive binary partitioning of the input space, along with the corresponding tree structure. In this example, the first step divides the whole of the input space into two regions according to whether $x1 \_ \theta 1$ or $x1 > \theta 1$ where $\theta 1$ is a parameter of the model. This creates two subregions, each of which can then be subdivided independently. For instance, the region $x1 \_ \theta 1$ is further subdivided according to whether $x2 \_ \theta 2$ or $x2 > \theta 2$, giving rise to the regions denoted A and B. The recursive subdivision can be described by the traversal of the binary tree shown in Figure



For any new input **x**, we determine which region it falls into by starting at the top of the tree at the root node and following a path down to a specific leaf node according to the decision criteria at each node. Note that such decision trees are not probabilistic graphical models. Within each region, there is a separate model to predict the target variable. For instance, or in classification we might assign each region to a specific class. A key property of tree based models, which makes them popular in fields such as medical diagnosis, for example, is that they are readily interpretable by humans because they correspond to a sequence of binary decisions applied to the individual input variables. For instance, to predict a patient's disease, we might first ask "Treat patient with blood preassure lowering Drug?". If the answer is yes, then we might next ask "is there is there any side effects with the drug?". Each leaf of the tree is then associated with a specific diagnosis. In order to learn such a model from a training set, we have to determine the structure of the tree, including which input variable is chosen at each node to form the split criterion as well as the value of the threshold parameter θi for the split. We also have to determine the values of the predictive variable within each region. In a Classification problem the goal is to classify a target variable t_1 from a D-dimensional vector $\mathbf{x} = (x1, \ldots, xD)^T$ of input variables. The training data consists of input vectors $\{\mathbf{x}1, \ldots, \mathbf{x}N\}$ along with the corresponding labels $\{t1,t2\}$. If the partitioning of the input space is given, The pruning criterion is then given by

$$C(T) = \sum_{\tau=1}^{|T|} Q_\tau(T) + \lambda|T|$$

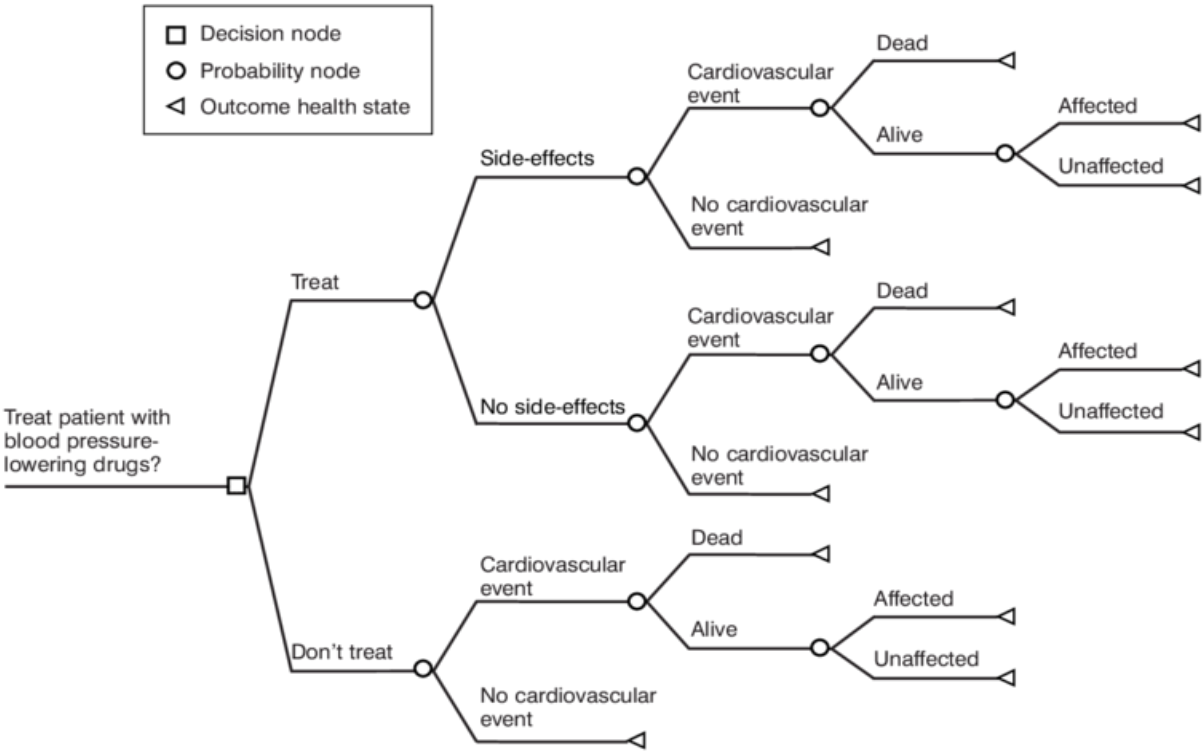
For classification problems, the process of growing and pruning the tree, While training the Dataset the growing of tree is done, while testing the dataset search operation is done on the tree, also we define $p\tau k$ to be the proportion of data points in region $R\tau$ assigned to class k, where $k = 1, \ldots, K$, then two commonly used choices are the cross-entropy.

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} \ln p_{\tau k}$$

And the Gini Index

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} (1 - p_{\tau k}) .$$

These both vanish for $p_{\tau k}$ = 0and $p_{\tau k}$ = 1and have a maximum at $p_{\tau k}$ = 0.5. They encourage the formation of regions in which a high proportion of the data points are assigned to one class. The cross entropy and the Gini index are better measures than the misclassification rate for growing the tree because they are more sensitive to the node probabilities. Also, unlike misclassification rate, they are differentiable and hence better suited to gradient based optimization methods. For subsequent pruning of the tree, the misclassification rate is generally used.



However, in practice it is found that the particular tree structure that is learned is very sensitive to the details of the data set, so that a small change to the training data can result in a very different set of splits.

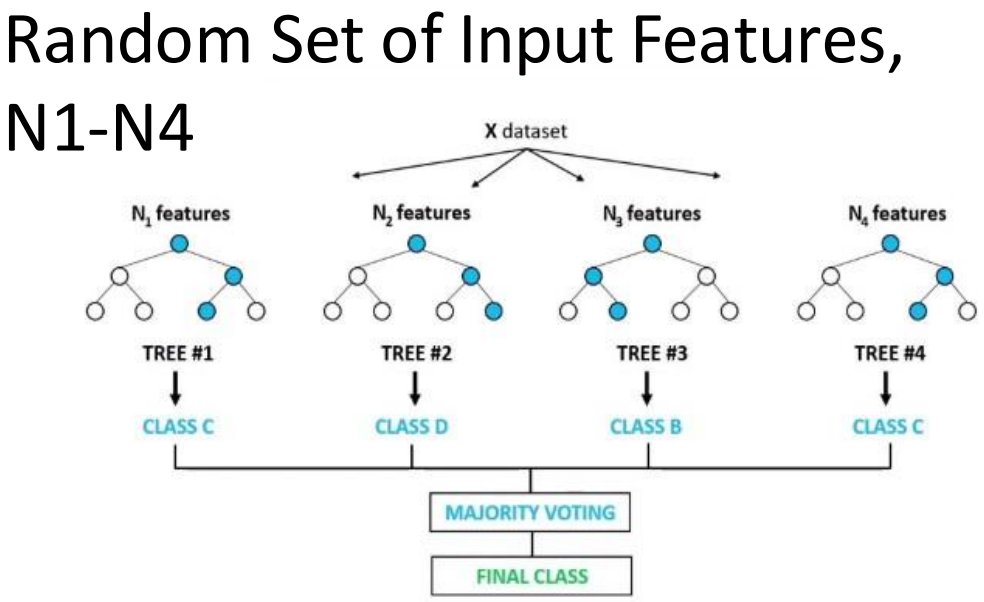
Random Forest

Our model extends existing forest-based techniques as it unifies classification, regression, density estimation, manifold learning, semi-supervised learning and active learning under the same decision forest framework. This means that the core implementation needs be written and optimized only once, and can then be applied to many diverse tasks. The proposed model may be used both in a generative or discriminative way and may be applied to discrete or continuous, labelled or unlabelled data.

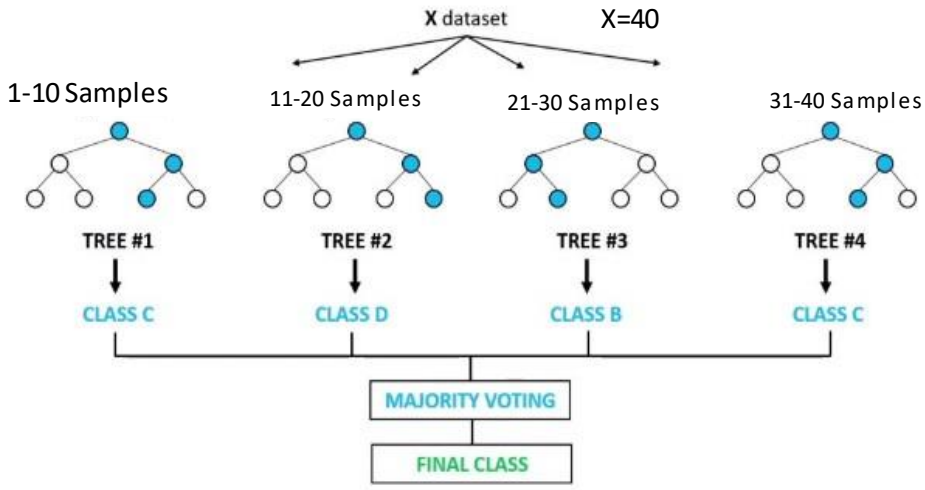
if we train not one but many decision trees, each on a random subset of training set or a random subset of the input features, and combine their predictions, overall accuracy can be significantly increased. This is the idea behind the random forest method.

The random forest algorithm works by completing the following steps:

Step 1: The algorithm select random samples from the dataset provided.
Step 2: The algorithm will create a decision tree for each sample selected. Then it will get a prediction result from each decision tree created.
Step 3: Voting will then be performed for every predicted result. For a classification problem, it will use **mode**, and for a regression problem, it will use **mean**.
Step 4: And finally, the algorithm will select the most voted prediction result as the final prediction.



Random Set of Input Samples

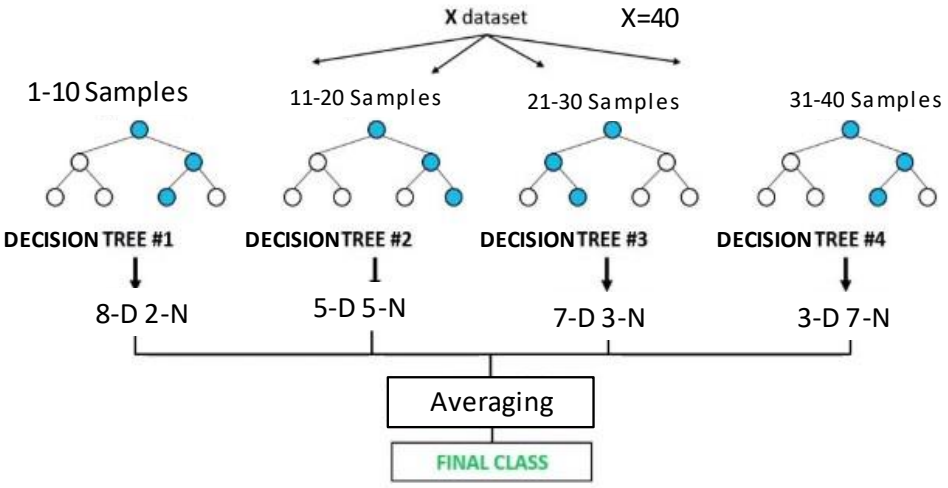


Splitting the dataset into Training and Test data

- We now split our processed dataset into training and test data.
- we split the feature (X) and target (y) dataframes into a training set (80%) and testing set (20%).
- Training set is used for building classification model
- Testing set is used for evaluating the performance of the classification model

Training using RF algorithm

- We need to separate the dataset into important features and target variables. we call the data sample frame with feature variables as X and the one with target variable as y.
- A random forest is a meta estimator that fits a number of decision tree Algorithm.
- It classifies on various sub-samples of the dataset and uses averaging to find Accuracy
- It improves the predictive accuracy and control over-fitting.
- The sub-sample size is controlled n=10,etc



Majority Voting 1D+1D+1D+0D=3D

Averaging (80+50+70+30)/4=57.5 D

Testing (Prediction)

- The unknown variables of testing dataset are predicted using the trained model formed at training stage and the error rate is calculated.

UNIT IV

ENSEMBLE TECHNIQUES AND UNSUPERVISED LEARNING

Combining Multiple Learners

- While using different models for solving classification and regression problems are analysed individually- only low performance
- Improved performance can be obtained by combining multiple models together in some way, instead a single model.
- For instance, we might train ***L*** different models and then make predictions using the average of the predictions made by each model.
- Such combinations of models are sometimes called *committees*.
- There are multiple ways to apply the committee concept, and It also give some insight into why it can sometimes be an effective procedure.

Committees

- The simplest way to construct a committee is to average the predictions by a set of individual models.
- Such a procedure can be motivated from a frequentist perspective by considering the trade-off between bias and variance, which decomposes the error due to a model into the bias component that arises from differences between the model and the true function to be predicted, and the variance component that represents the sensitivity of the model to the individual data points.
- One approach is to use *bootstrap* data sets, Consider a regression problem in which we are trying to predict the value of a single continuous variable, and suppose we generate M bootstrap data sets and then use each to train a separate copy $y_m(\mathbf{x})$ of a predictive models where $m = 1, \ldots, M$. The committee prediction is given by

$$y_{\mathrm{COM}}(\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} y_m(\mathbf{x}).$$

Suppose the true regression function that we are trying to predict is given by h(x), so that the output of each of the models can be written as the true value plus an error in the form

$$y_m(\mathbf{x}) = h(\mathbf{x}) + \epsilon_m(\mathbf{x}).$$

h(x)- True Value $\epsilon_m(x)$- Error

The average sum-of-squares error then takes the form

$$\mathbb{E}_{\mathbf{x}}\left[\{y_m(\mathbf{x}) - h(\mathbf{x})\}^2\right] = \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right]$$

Where,

Ex[·] - Frequentist Expectation with respect to the distribution of the input vector x.

The average error made by the models acting individually is therefore

$$E_{\mathrm{AV}} = \frac{1}{M}\sum_{m=1}^{M}\mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right]$$

$$\boldsymbol{E}_{\mathbf{COM}} = \mathbf{1}/\boldsymbol{M}(\boldsymbol{E}_{\mathbf{AV}})$$

- the Average Error of a model can be reduced by a factor of *M* simply by averaging *M* versions of the model.
- Unfortunately, it depends on the key assumption that the errors due to the individual models are uncorrelated.
- In practice, the errors are typically highly correlated, and the reduction in overall error is generally small.
- It can, however, be shown that the expected committee error will not exceed the expected error of the constituent models, so that *E*COM ≤*E*AV.

Model Combination Schemes

There are also different ways the multiple base-learners are combined to generate the final output:

a. ***Multiexpert combination* methods**

- These have base-learners that work in *parallel*.
- These methods can in turn be divided into two:

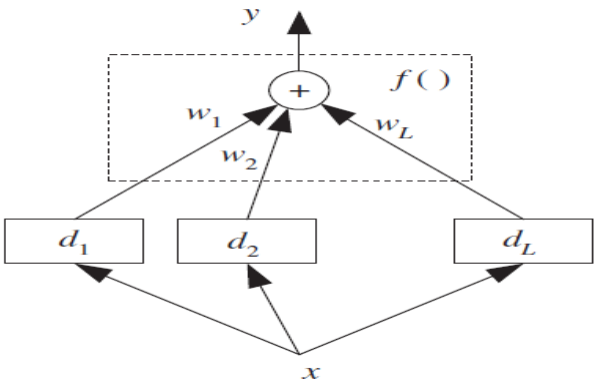
***Global* approach**, (*learner fusion),*

- given an input, all base-learners generate an output and all these outputs are used.
- Examples are *voting* and *stacking*.

***Local* approach**, (*learner selection)*

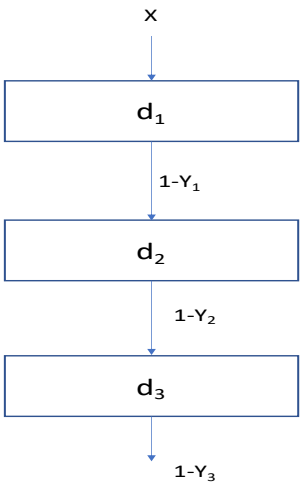
- which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.
- example, In a *mixture of experts* a *gating* model is used

- Base-learners are d_j and their outputs are combined using $f(\cdot)$.



***b. Multistage combination* methods**

- This method uses a *serial* approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough.
- The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident.
- An example is *cascading*.
- Let us say that we have L base-learners.
- We denote by $d_j(x)$ the prediction of base-learner M_j given the arbitrary dimensional input x. In the case of multiple representations, each M_j uses a different input representation x_j .
- The final prediction is calculated from the predictions of the base-learners:



$$y = f(d_1, d_2, \ldots, d_L | \Phi)$$

SIMPLE ENSEMBLE TECHNIQUES

A few simple but powerful techniques

- Max Voting
- Averaging
- Weighted Averaging

1. Max Voting / Majority Voting

This voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data predictions by each model are considered as a vote.

from the majority of models are used as the final prediction. For example, when you asked 5 of your friends to rate your painting (out of 5); We will assume the out them rated it as 4 while two of them gave it a 5. Since the majority gave the rating of 4, the final will be taken as 4 You can consider this

as taking the mode of all the predictions.

	Student 1	**Student 2**	**Student 3**	**Student 4**	**Student 5**	**Final Rating**
Rating	**5**	**4**	**5**	**4**	**4**	**4**

1. Averaging

Similar to the max voting technique, multiple predictions are made for each point in averaging. In this method, we take an average of predictions from all the models and use it to make the final prediction. Averaging can be used for making predictions in regression problems or while calculating probabilities for classification problems. For example, in the below case, the averaging method would take the average of all the values.

	Student 1	**Student 2**	**Student 3**	**Student 4**	**Student 5**	**Final Rating**
Rating	**5**	**4**	**5**	**4**	**4**	**4.4**

3. Weighted Average

This is an extension of the averaging method. All models are assigned different weights define the importance in each model for prediction. For instance, if two of your friends are critics, while others have no prior experience in this field, the answers by these two friends are given more Importance as compared, to the other. The result is calculated as

	Student 1	Student 2	Student 3	Student 4	Student 5	Final Rating
Weights	**0.23**	**0.23**	**0.18**	**0.18**	**0.18**	**4.41**
Rating	**5**	**4**	**5**	**4**	**4**	

5x0.23+4x0.23+5x0.18+4x0.18+4x0.18=4.41

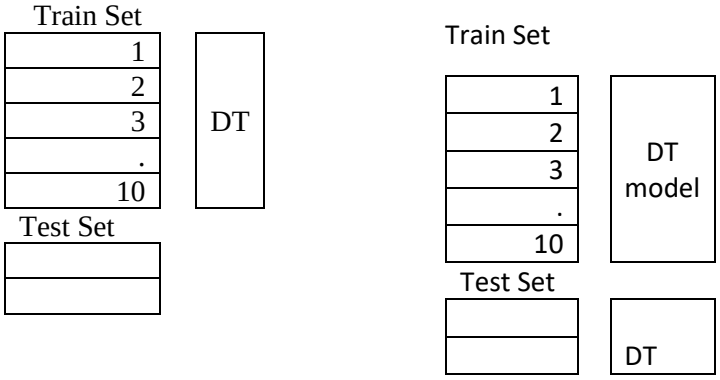
ADVANCED ENSEMBLE TECHNIQUES

1. **STACKING**
2. **BLENDING**

STACKING

1. In an ensemble learning technique, it uses predictions from multiple methods. (decision tree, SVM, etc.) to build a new model. This model us used for predictions on the test set.

2. The base model (decision tree) Is used on 9 parts of dataset, and predictions are made using the 10th part of the dataset.

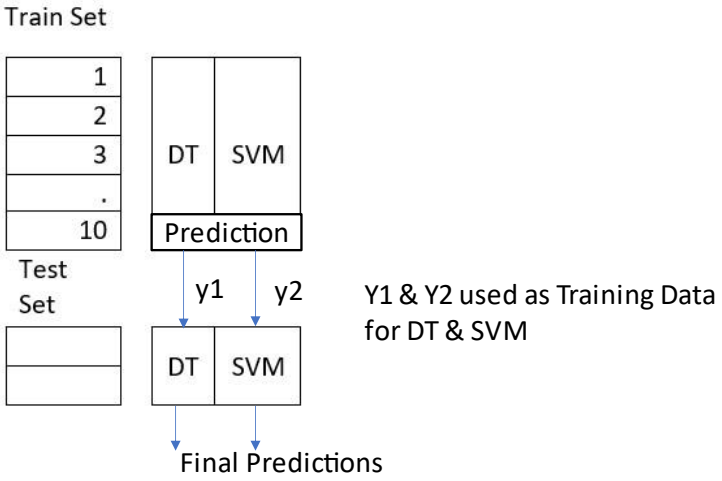


3. The base model (decision tree) Is fitted on the whole train dataset, and predictions are made on the test.

4.Thus repeating for another base model say SVM, the train on 10 sets of dataset, resulting in another set of predictions for and test data.

5. Prediction from the train set (Y1&Y2) are used as features to build new model.

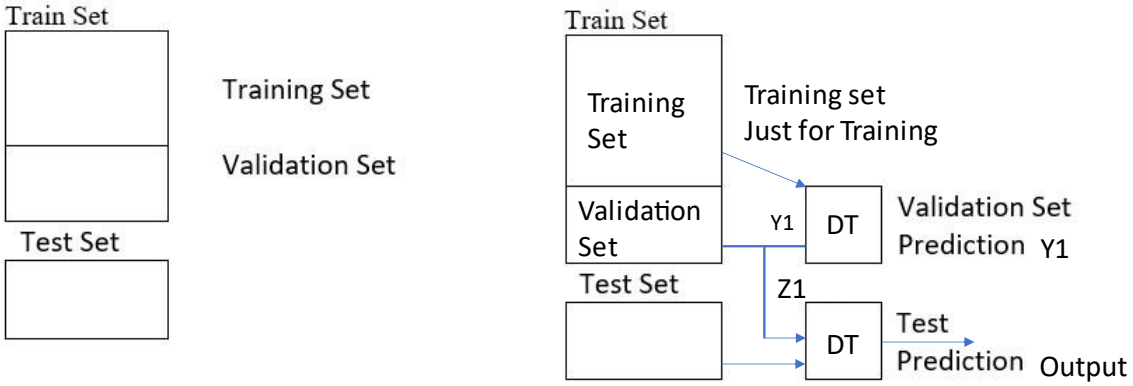
6.This model is used to make final predictions on the test prediction set



Blending

This method follows the same approach as stacking

- but it uses only a holdout validation set from the train set to make predictions.
- The predictions are made on the holdout set only.
- The holdout set and the predictions are used to build a model which will run on the test dataset.



Y1 is the Predicted Output used along with Validation dataset data samples To form new Training Set (Z1)

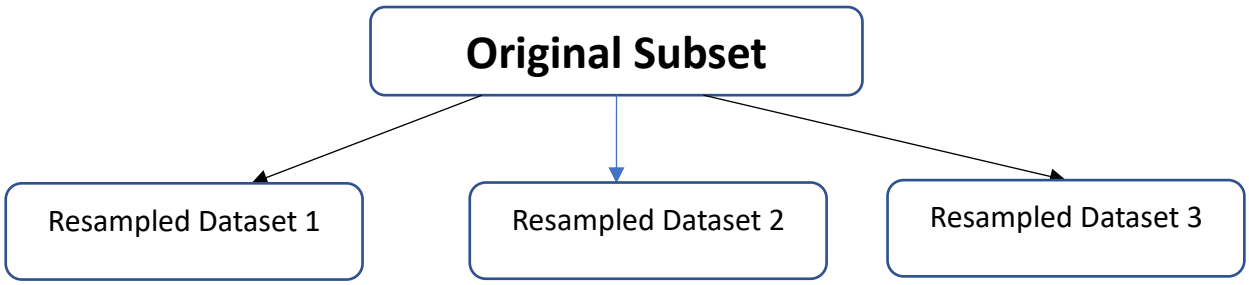
.

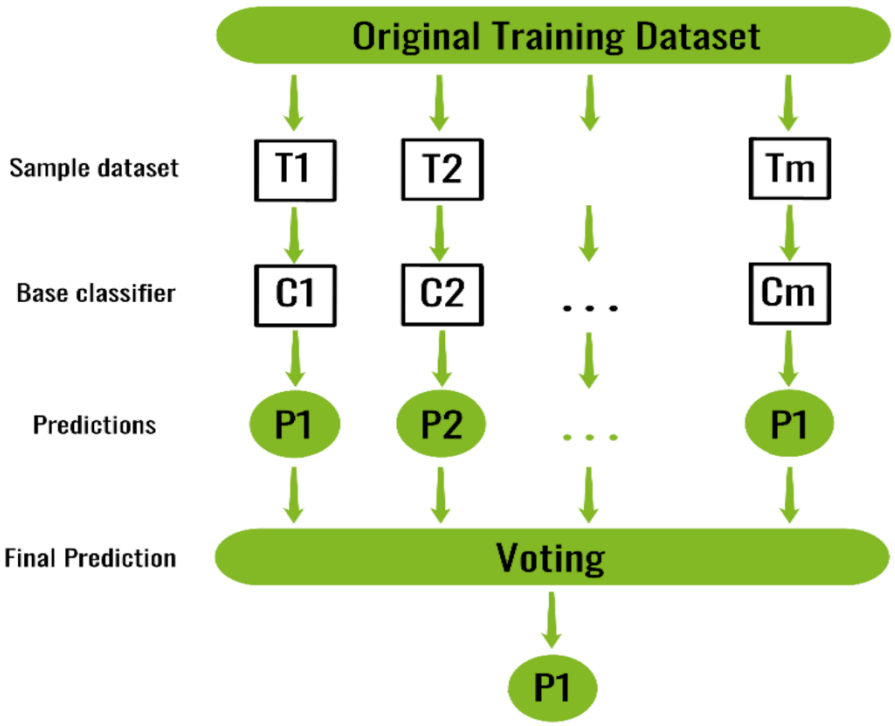
Bagging (Bootstrap Aggregating)

- The Bagging classifier is a general-purpose ensemble method that can be used with a variety of different base models, such as decision trees, neural networks, and linear models.
- It uses bootstrap resampling to generate multiple different subsets of the training data, and then trains a separate model on each subset.
- The final predictions are made by combining the predictions of all the models by using voting method whereby base-learners are made different by training them over slightly different training sets.
- As Bagging resamples the original training dataset with replacement, some instance (or data) may be present multiple times while others are left out.
- The Bagging classifier can be used to improve the performance of any base classifier that has high variance, for example, decision tree classifiers.
- The Bagging classifier can be used in the same way as the base classifier with the only difference being the number of estimators and the bootstrap parameter.
- It uses bootstrap resampling to generate multiple different subsets of the training data, and then trains a separate model on each subset.
- It reduces the variance of the model and can help to reduce overfitting.

Bootstrap Resampling

- **Original training dataset (samples):** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- **Resampled training set 1:** 2, 3, 3, 5, 6, 1, 8, 10, 9, 1
- **Resampled training set 2:** 1, 1, 5, 6, 3, 8, 9, 10, 2, 7
- **Resampled training set 3:** 1, 5, 8, 9, 2, 10, 9, 7, 5, 4





Algorithm

Classifier generation:

1. Let N be the size of the training set.
2. for each of t iterations:
    - sample N instances with replacement from the original training set.
    - apply the learning algorithm to the sample.
    - store the resulting classifier.

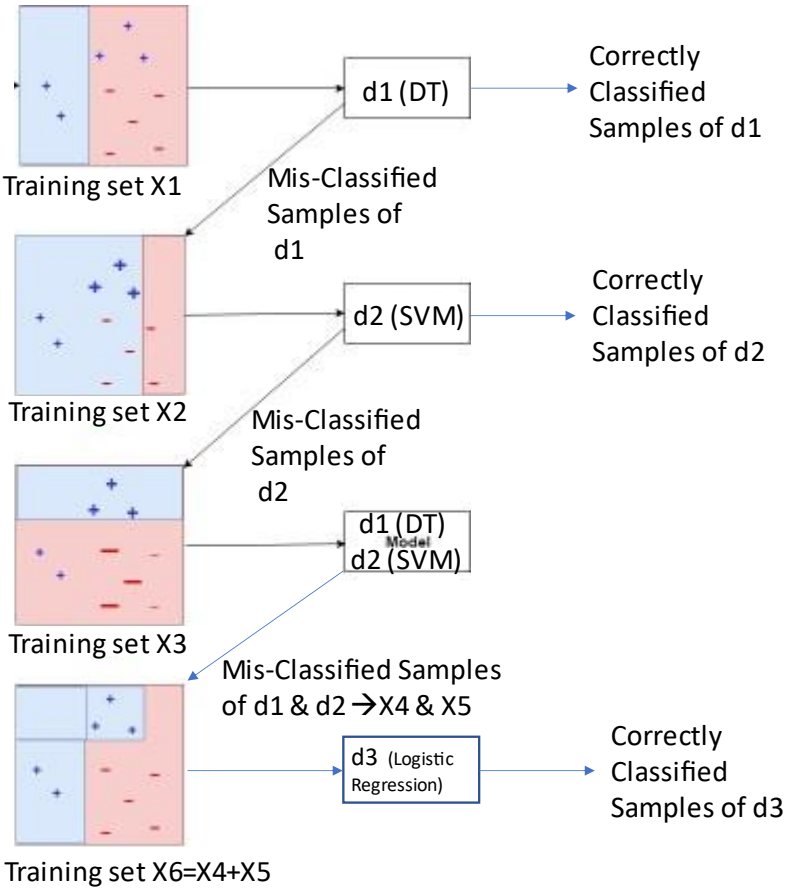
Classification:

3. for each of the t classifiers:
    - predict class of instance using classifier.
4. return class that was predicted most often.

Boosting

- In boosting, Here Multi stage Combination of base-learners is done by training the next learner on the mistakes of the previous learners.
- The original *boosting* algorithm combines three weak learners to generate a strong learner.

- A *weak learner* has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a *strong learner* has arbitrarily small error probability.
- Given a large training set, (X_1,X_2,X_3) we randomly divide it into three.
- We use 2 Models d_1,d_2,d_3
- We use X_1 to train d_1.
- Then take X_2 to Predict d_2, take only the instances of misclassified by d_1
- We then take X_3 and predict it to d_1 and d_2.
- The instances on which d_1 and d_2 disagree form the training set of d_3.
- During testing, given an instance, we give it to d_1 and d_2; if they agree, that is the response, otherwise the response of d_3 is taken as the output.
- This overall system has reduced error rate



AdaBoost was the first really successful boosting algorithm developed for the purpose of binary classification. AdaBoost is short for Adaptive Boosting and is a very popular boosting technique that combines multiple "weak classifiers" into a single "strong classifier".

Algorithm(AdaBoost)

1. Initialise the dataset and assign equal weight to each of the data point.
2. Provide this as input to the model and identify the wrongly classified data points.
3. Increase the weight of the wrongly classified data points.
4. if (got required results)
   Goto step 5
   else
   Goto step 2
5. End

X1 consists of 10 data points which consist of two types namely plus(+) and minus(-) and 5 of which are plus(+) and the other 5 are minus(-) and each one has been assigned equal weight initially. The first model tries to classify the data points and generates a vertical separator line but it wrongly classifies 3 plus(+) as minus(-).

X2 consists of the 10 data points from the previous model in which the 3 wrongly classified plus(+) are weighted more so that the current model tries more to classify these pluses(+) correctly. This model generates a vertical separator line that correctly classifies the previously wrongly classified pluses(+) but in this attempt, it wrongly classifies three minuses(-).

X3 consists of the 10 data points from the previous model in which the 3 wrongly classified minus(-) are weighted more so that the two model tries more to classify these minuses(-) correctly. This model generates a horizontal separator line that correctly classifies the previously wrongly classified minuses(-).

X4 combines together X4, X6 in order to build a strong prediction model which is much better than any individual model used.

Stacked Generalization

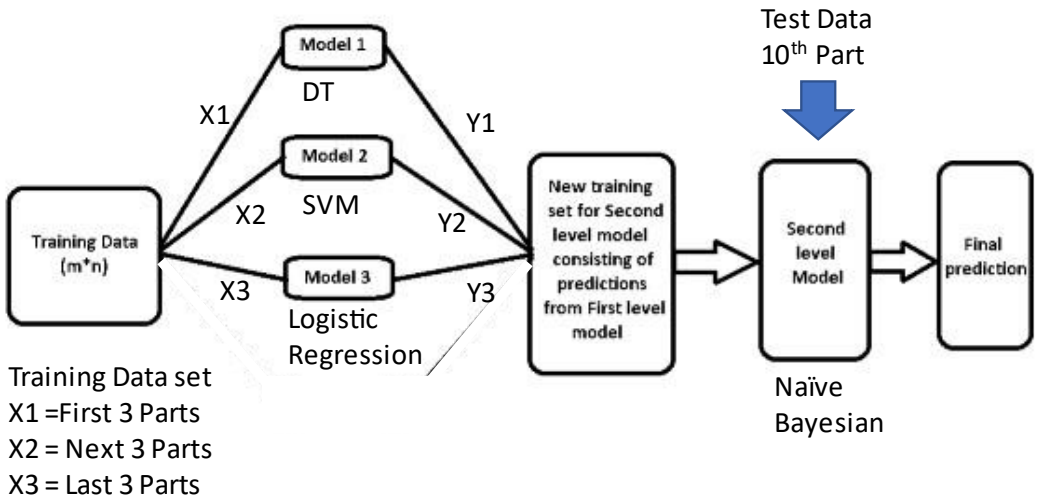
- The point of stacking is to explore a space of different models for the same problem.
- The idea is that you can attack a learning problem with different types of models which are capable to learn some part of the problem, but not the whole space of the problem.
- you can build multiple different learners and you use them to build an intermediate prediction, one prediction for each learned model.
- Then you add a new model which learns from the intermediate predictions the same target.

- This final model is said to be stacked on the top of the others, hence the name.
- Thus, it improves The overall performance

Algorithm

1. We split the training data into K subsets.
2. A base model is fitted on the K-1 parts and predictions are made for Kth part.
3. We do for each part of the training data.
4. The base model is then fitted on the whole train data set to calculate its performance on the test set.
5. We repeat the last 3 steps for other base models.
6. Predictions from the train set are used as features for the second level model.

Second level model is used to make a prediction on the test set.



Supervised learning:

- discover patterns in the data that relate data attributes with a target (class) attribute.
- These patterns are then utilized to predict the values of the target attribute in future data instances.
- Eg. KNN

Unsupervised learning:

- The data have no target attribute.
- We want to explore the data to find some intrinsic structures in them.
- Eg. Clustering (K-Means)

UNSUPERVISED LEARNING

In unsupervised learning, there is no such supervisor and we only have input data. The aim is to find the regularities in the input. There is a structure to the input space such that certain patterns occur more often than others, and we want to see what generally happens and what does not. In statistics, density estimation this is called *density estimation*.

a. Clustering

One method for density estimation is *clustering* where the aim is to find clusters or groupings of input. In the case of a company with a data of past customers, the customer data contains the demographic information as well as the past transactions with the company, and the company may want to see the distribution of the profile of its customers, to see what type of customers frequently occur. In such a case, a clustering model allocates customers similar in their attributes to the same group, providing the company with natural groupings of its customers; this is called *customer segmentation*. Once such groups are found, the company may decide strategies, for example, services and products, specific to different groups; this is known as *customer relationship management*. Such a grouping also allows identifying those who are outliers, namely, those who are different from other customers, which may imply a niche in the market that can be further exploited by the company.

An interesting application of clustering is in *image compression*. In this case, the input instances are image pixels represented as RGB values. A clustering program groups pixels with similar colors in the same group, and such groups correspond to the colors occurring frequently in the image. If in an image, there are only shades of a small number of colors, and if we code those belonging to the same group with one color, for example, their average, then the image is quantized. Let us say the pixels are 24 bits to represent 16 million colors, but if there are shades of only 64 main colors, for each pixel we need 6 bits instead of 24. For example, if the scene has various shades of blue in different parts of the image, and if we use the same average blue for all of them, we lose the details in the image but gain space in storage and transmission. Ideally, we would like to identify higher-level regularities by analyzing repeated image patterns, for example, texture, objects, and so forth. This allows a higher-level, simpler, and more useful description of the scene, and for example, achieves better compression than compressing at the pixel level. If we have scanned document pages, we do not have random on/off pixels but bitmap images of characters. There is structure in the data, and
we make use of this redundancy by finding a shorter description of the data: 16 × 16 bitmap of ‘A’ takes 32 bytes; its ASCII code is only 1 byte.

In *document clustering*, the aim is to group similar documents. For example, news reports can be subdivided as those related to politics, sports, fashion, arts, and so on. Commonly, a document is represented as a *bag of words*—that is, we predefine a lexicon of N words, and each document is an N-dimensional binary vector whose element i is 1 if word i appears in the document; suffixes "–s" and "–ing" are removed to avoid duplicates and words such as "of," "and," and so forth, which are not informative, are not used. Documents are then grouped depending on the number of shared words. It is of course critical how the lexicon is chosen.

Machine learning methods are also used in *bioinformatics*. DNA in our genome is the "blueprint of life" and is a sequence of bases, namely, A, G, C, and T. RNA is transcribed from DNA, and proteins are translated from the RNA. Proteins are what the living body is and does. Just as a DNA is a sequence of bases, a protein is a sequence of amino acids (as defined by bases). One application area of computer science in molecular biology is *alignment*, which is matching one sequence to another. This is a difficult string matching problem because strings may be quite long, there are many template strings to match against, and there may be deletions, insertions, and substitutions. Clustering is used in learning, which are sequences of amino acids that occur repeatedly in proteins. ***Motifs*** are of interest because they may correspond to structural or functional elements within the sequences they characterize. The analogy is that if the amino acids are letters and proteins are sentences, motifs are like words, namely, a string of letters with a particular meaning occurring frequently in different sentences.

b. Reinforcement Learning

In some applications, the output of the system is a sequence of *actions*. In such a case, a single action is not important; what is important is the *policy* that is the sequence of correct actions to reach the goal. There is no such thing as the best action in any intermediate state; an action is good if it is part of a good policy. In such a case, the machine learning program should be able to assess the goodness of policies and learn from past good action sequences to be able to generate a policy. Such learning methods are called *reinforcement learning* algorithms.
learning A good example is *game playing* where a single move by itself is not that important; it is the sequence of right moves that is good. A move is good if it is part of a good game playing policy. Game playing is an important research area in both artificial intelligence and machine learning. This is because games are easy to describe and at the same time, they are quite difficult to play well. A game like chess has a small number of rules but it is very complex because of the large number of possible moves at each state and the large number of moves

that a game contains. Once we have good algorithms that can learn to play games well, we can also apply them to applications with more evident economic utility.

A robot navigating in an environment in search of a goal location is another application area of reinforcement learning. At any time, the robot can move in one of a number of directions. After a number of trial runs, it should learn the correct sequence of actions to reach to the goal state from an initial state, doing this as quickly as possible and without hitting any of the obstacles. One factor that makes reinforcement learning harder is when the system has unreliable and partial sensory information. For example, a robot equipped with a video camera has incomplete information and thus at any time is in a *partially observable state* and should decide on its action taking into account this uncertainty; for example, it may not know its exact location in a room but only that there is a wall to its left. A task may also require a concurrent operation of *multiple agents* that should interact and cooperate to accomplish a common goal. An example is a team of robots playing soccer.

Clustering

- Clustering is a technique for finding Identical groups in data, called **clusters**. I.e.,
- it groups data samples that are similar (near) to each other in one cluster and data instances that are very different (far away) from each other into different clusters.
- Clustering is often called an **unsupervised learning** task as no class values denoting an a priori grouping of the data instances are given, which is the case in supervised learning.
- Uses A distance function (for finding similarity)
- Clustering quality
    - Inter-clusters distance $\Rightarrow$ maximized
    - Intra-clusters distance $\Rightarrow$ minimized
- The quality of a clustering result depends on the algorithm, the distance function, and the application.

K-means Clustering

- K-means is a clustering algorithm
- Let the set of data points (or instances) X be

  $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, Samples
    - where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ir})$ is a vector in a real-valued space $X \subseteq R^r$,

- *r* is the number of attributes (Features) in the data.

- The *k*-means algorithm partitions the given data (X_i) into any of the *k* clusters.
  - Each cluster has a cluster **center**, called **centroid**.
  - *k* is specified by the user

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$

Algorithm

1) Randomly choose *k* data points (seeds) to be the initial cluster centers (centroids),
2) Assign each data point to the closest cluster centers by finding the distance of Sample and cluster center
3) Re-compute the cluster centers (centroids) using the current cluster memberships.
4) If a convergence criterion is not met, go to (2).

Cluster Center (μ_j)

- The Cluster Center is the data point selected by averaging other datapoints in a cluster. It is used for finding the membership of the class.

$$\boldsymbol{\mu j} = \frac{1}{|C_j|} \sum_{X_i \in C_j} X_i$$

|C_j| –No of Data points in Cluster C_j

Distance Function

- The Distance from one Data sample (x_i) to a Mean (**μj** - Cluster Center) is computed with
- If *h* = 2, it is the Euclidean distance

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(x_{i1} - \mu_{j1})^2 + (x_{i2} - \mu_{j2})^2 + \ldots + (x_{ir} - \mu_{jr})^2}$$

Membership class of an Sample

- The Algorithm computes the distance between each datapoint with the every cluster center.
- Finds the shortest distance between the data point and the considered clusters.
- And choose the shortest distance of cluster as the membership of the class.
- By consecutively, finding the minimum distance between the cluster centers and the every data points, membership class can be found for each data points.

Stopping/convergence criterion

- no (or minimum) re-assignments of data points to different clusters,
- no (or minimum) change of centroids, or
- minimum decrease in the sum of squared error (SSE),
- C_i is the j^{th} cluster, m_j is the centroid of cluster C_j (the mean vector of all the data points in C_j), and $dist(x, m_j)$ is the distance between data point x and centroid m_j.

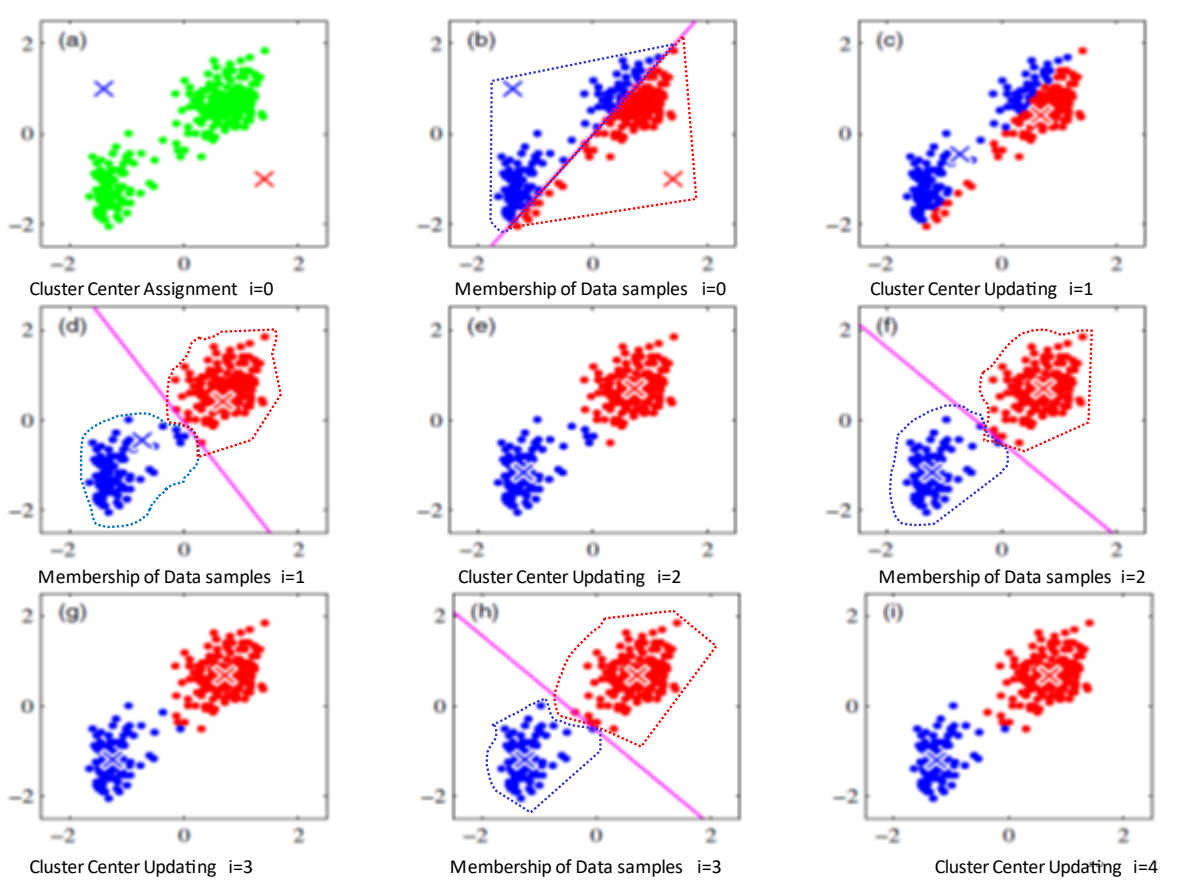


Illustration of the *K*-means algorithm using the re-scaled Old Faithful data set.

(a) Green points denote the data set in a two-dimensional Euclidean space. The initial choices for centres $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are shown by the red and blue crosses, respectively.

(b) In the initial E step, each data point is assigned either to the red cluster or to the blue cluster, according to which cluster centre is nearer. This is equivalent to classifying the points according to which side of the perpendicular bisector of the two cluster centres, shown by the magenta line, they lie on.

(c) In the subsequent M step, each cluster centre is re-computed to be the mean of the points assigned to the corresponding cluster.

(d)–(i) show successive E and M steps through to final convergence of the algorithm.

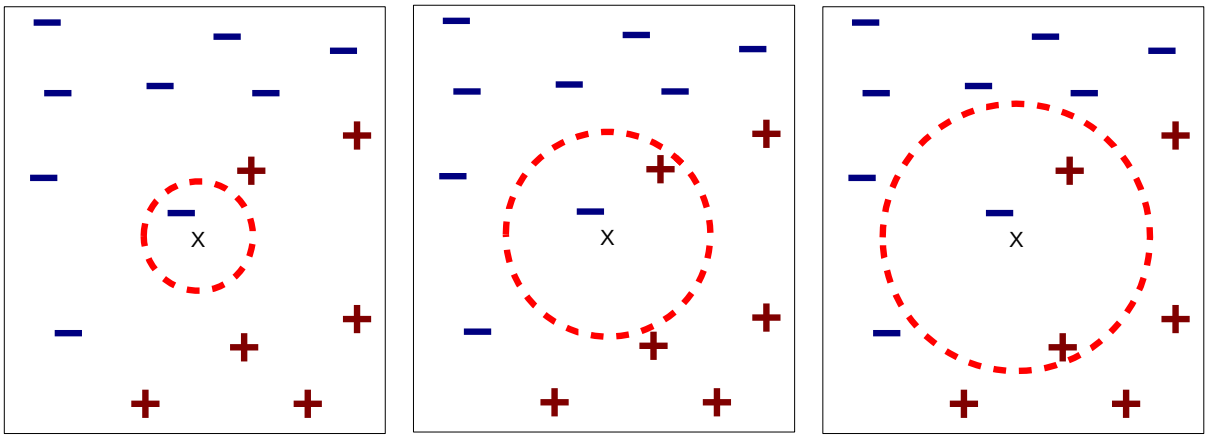
Instance Based Learning

In machine learning literature, nonparametric methods are also called *instance-based* or *memory-based learning* algorithms, since what they do is store the training instances in a lookup table and interpolate from these. This implies that all of the training instances should be stored and storing all requires memory of O*(N)*. Furthermore, given an input, similar ones should be found, andl finding them requires computation of O*(N)*.

k-Nearest Neighbour Estimator

The nearest neighbour class of estimators adapts the amount of smoothing to the *local* density of data. The degree of smoothing is controlled by *k*, the number of neighbours taken into account, which is much smaller than *N*, the sample size.

- The *k*-nearest neighbor classifier classifies a point based on the majority of the *k* closest training points.
- You can use nearest neighbor classifiers if you have some way of defining "distances" between attributes.

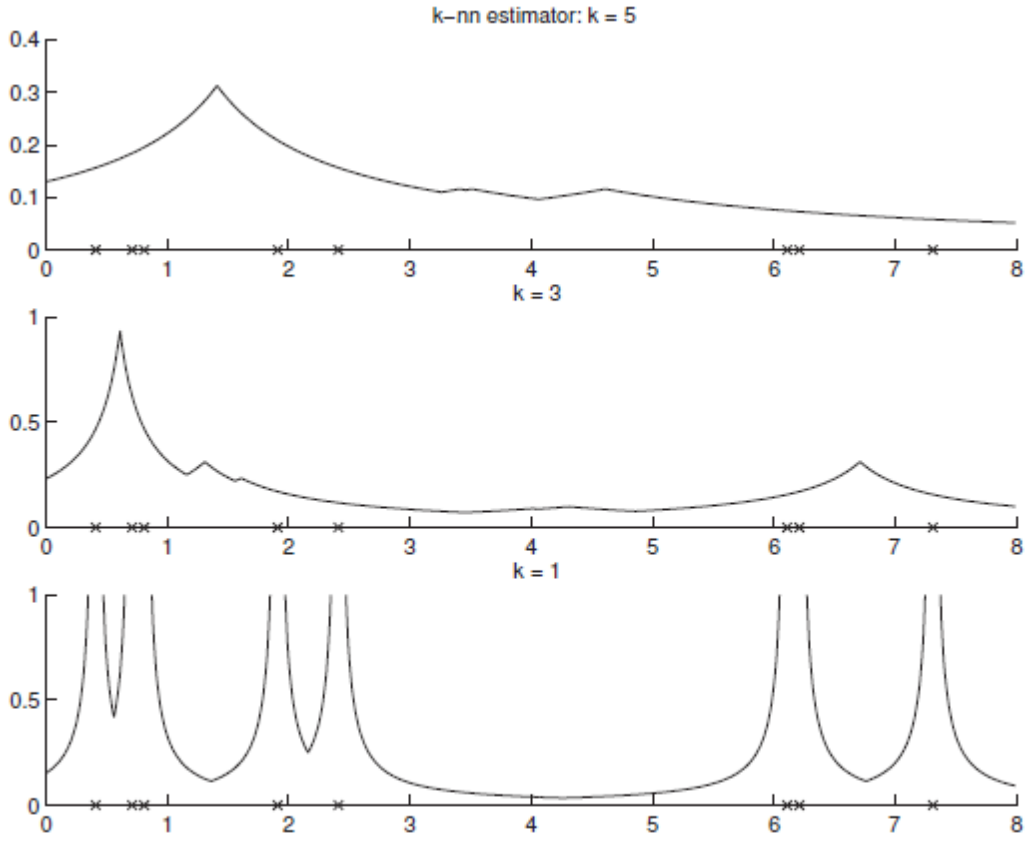


(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

- Let us define a distance between a and b, for example, $|a - b|$, and for each x,
- we $d_1(x) \leq d_2(x) \leq \cdots \leq d_N(x)$ to be the distances arranged in ascending order, from x to the points in the sample:
- $d_1(x)$ is the distance to the nearest sample
- $d_2(x)$ is the distance to the next nearest, and so on.

- If x_t are the data points, then we define
- $d_1(x) = \min_t |x - x_t|$,
- if i is the index of the closest sample, namely, $i = \arg\min_t |x - x_t|$, then
- $d_2(x) = \min_{j=i} |x - x_j|$, and so forth.
- The *k-nearest neighbor* (*k*-nn) density estimate is

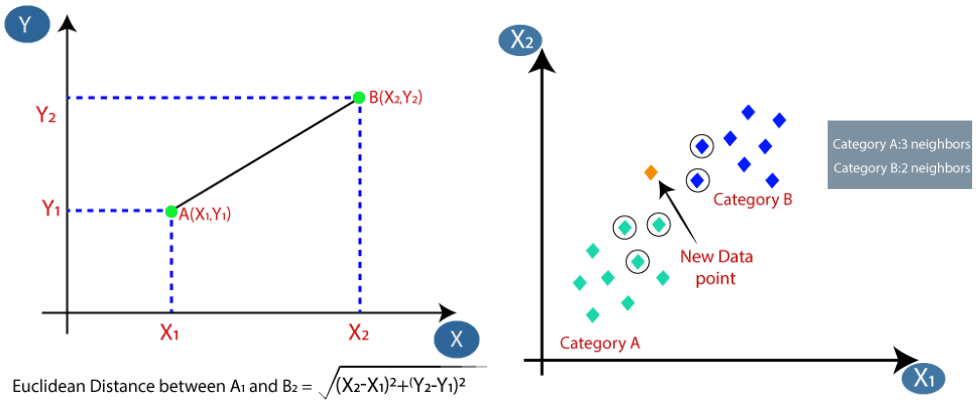
$$\hat{p}(x) = \frac{k}{2Nd_k(x)}$$



KNN Classifier

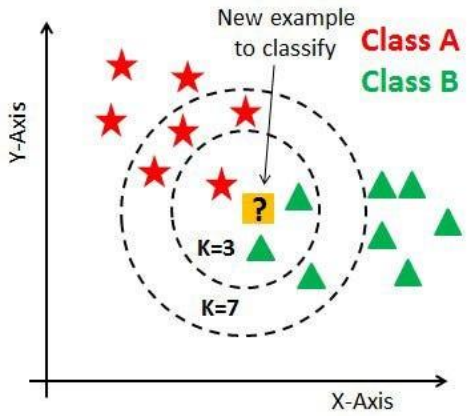
- The nearest neighbor algorithm is an instance-based Lazy learning algorithm.
- It defer the computation for classifying a sample until a test sample is ready to be classified.
- It meets the criteria by storing the entire training set in memory and calculating the distance from a test sample to every training sample at classification time.
- the predicted class of the test sample is the class of the closest training sample.

- The nearest neighbor algorithm is a specific instance of the k-nearest neighbor algorithm where k = 1.
- While a test sample is there , In order to classify, we tabulate the classes for each of the k closest training samples and predict the class of the test sample as the mode of the training samples' classes.
- In binary classification tasks, k is normally chosen to be an odd number in order to avoid ties.



- By calculating the Euclidean distance, we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B.

- K-value indicates the count of the nearest neighbors.
- We have to compute distances between test points and trained labels points.
- Updating distance metrics with every iteration is computationally expensive, and that's why KNN is a lazy learning algorithm.



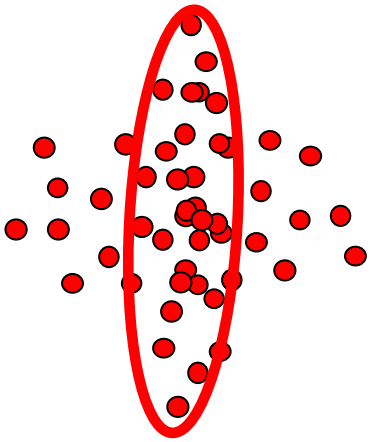
- if we proceed with K=3, then we predict that test input belongs to class B, and if we continue with K=7, then we predict that test input belongs to class A.

Gaussian Mixture Models

- Clusters modeled as Gaussians and Not just by their mean.
- **EM algorithm:** assign data to cluster with some *probability*
- It models each cluster using one of these Gaussian bells
- The Gaussian mixture model uses a simple linear superposition of Gaussian components.
- Aimed at providing a richer class of density models than the single Gaussian.
- Now turn to a formulation of Gaussian mixtures in terms of discrete *latent* variables.
- Recall from the Gaussian mixture distribution can be written as a linear superposition of Gaussians in the form.

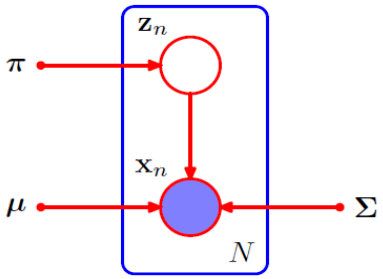
$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

$$\boldsymbol{\Sigma}_k = \sigma_k^2 \mathbf{I},$$



$$p(z_k = 1) = \pi_k$$

- Graphical representation of a Gaussian mixture model
- a set of *N number of* data points $\{\mathbf{x}_k\}$,
- corresponding latent points $\{\mathbf{z}_k\}$,
- where $k = 1, \ldots, N$.
- $\pi_k \rightarrow$ Mixing coefficients

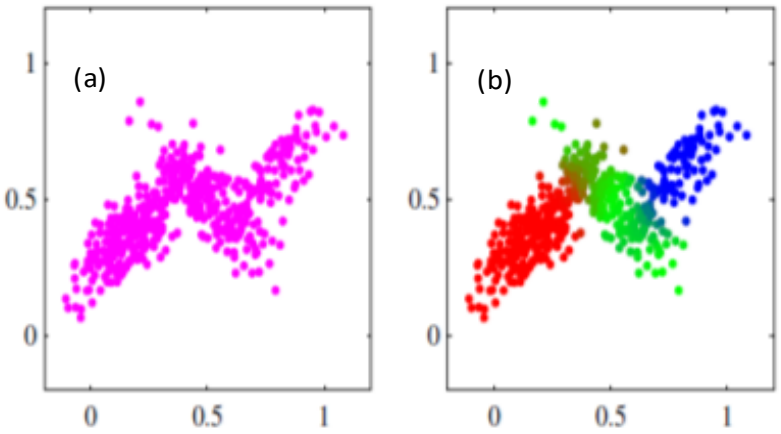


Example 500 points drawn from the mixture of 3 Gaussians

(a) The corresponding samples from the marginal distribution $p(\mathbf{x})$, which is obtained by simply ignoring the values of $\mathbf{z}$ and just plotting the $\mathbf{x}$ values. The data set in (a) is said to be incomplete,

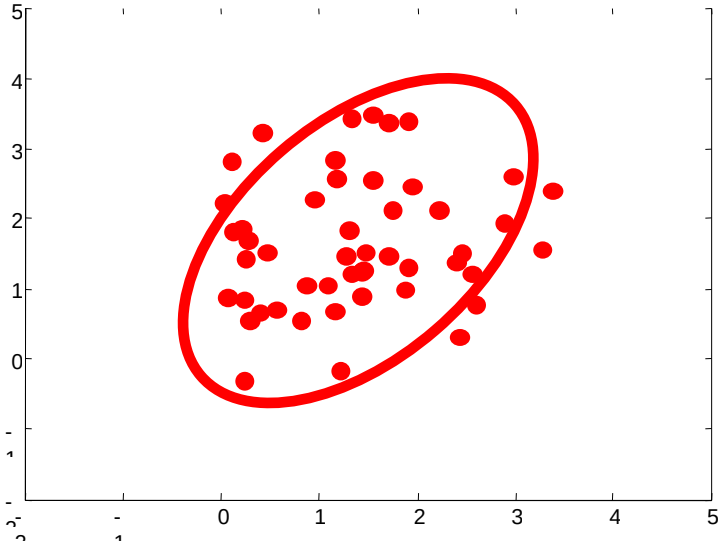
(b) Samples from the joint distribution $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ in which the three states of $\mathbf{z}$, corresponding to the three components of the mixture, are depicted in red, green, and blue.

The same samples in which the colours represent the value of the responsibilities whereas that in (b) is complete.



Multivariate Gaussian models

$$\mathcal{N}(\underline{x}\ ;\ \underline{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}} |\Sigma|^{-1/2} \exp\left\{ -\frac{1}{2}(\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu}) \right\}$$



$$\hat{\mu} = \frac{1}{N} \sum_i x^{(i)}$$

$$\hat{\Sigma} = \frac{1}{N} \sum_i (x^{(i)} - \hat{\mu})^T (x^{(i)} - \hat{\mu})$$

Expectation Maximization for Gaussian mixtures

An elegant and powerful method for finding maximum likelihood solutions for models with latent variables is called the *expectation-maximization* algorithm, or *EM* algorithm. Initially, we shall motivate the EM algorithm by giving a relatively informal treatment in the context of the Gaussian mixture model. the conditions that must be satisfied at a maximum of the likelihood function are, Setting the derivatives of ln $p(\mathbf{X}|\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\Sigma})$ in with respect to the means $\boldsymbol{\mu}_k$ of the Gaussian components to zero, we obtain

$$\gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}.$$

- ɣ(z_k) can also be viewed as the responsibility that component k takes for 'explaining' the observation x

E-step "Expectation" (r_{ic})

- Start with parameters describing each cluster
- Mean ^1_c, Covariance σc, "size" $\frac{1}{4}c$
- For each datum (example) x_i,
- Compute "r_{ic}", the probability that it belongs to cluster c
- Compute its probability under model c
- Normalize to sum to one (over clusters c)

$$r_{ic} = \frac{\pi_c \mathcal{N}(x_i \ ; \ \mu_c, \Sigma_c)}{\sum_{c'} \pi_{c'} \mathcal{N}(x_i \ ; \ \mu_{c'}, \Sigma_{c'})}$$

- If x_i is very likely under the c^{th} Gaussian, it gets high weight
- Denominator just makes r's sum to one.

M-step "Maximization"

- Start with assignment probabilities r_{ic}
- Update parameters: mean ^1_c, Covariance §c, "size" ¼c
- For each cluster (Gaussian) x_c,
- Update its parameters using the (weighted) data points.

$$N_c = \sum_i r_{ic}$$ Total responsibility allocated to cluster c

$$\pi_c = \frac{N_c}{N}$$ Fraction of total assigned to cluster c

$$\mu_c = \frac{1}{N_c} \sum_i r_{ic} x_i$$ Weighted mean of assigned data

$$\Sigma_c = \frac{1}{N_c} \sum_i r_{ic} (x_i - \mu_c)^T (x_i - \mu_c)$$ Weighted covariance of assigned data

Expectation-Maximization Convergence

- Each step increases the log-likelihood of this model

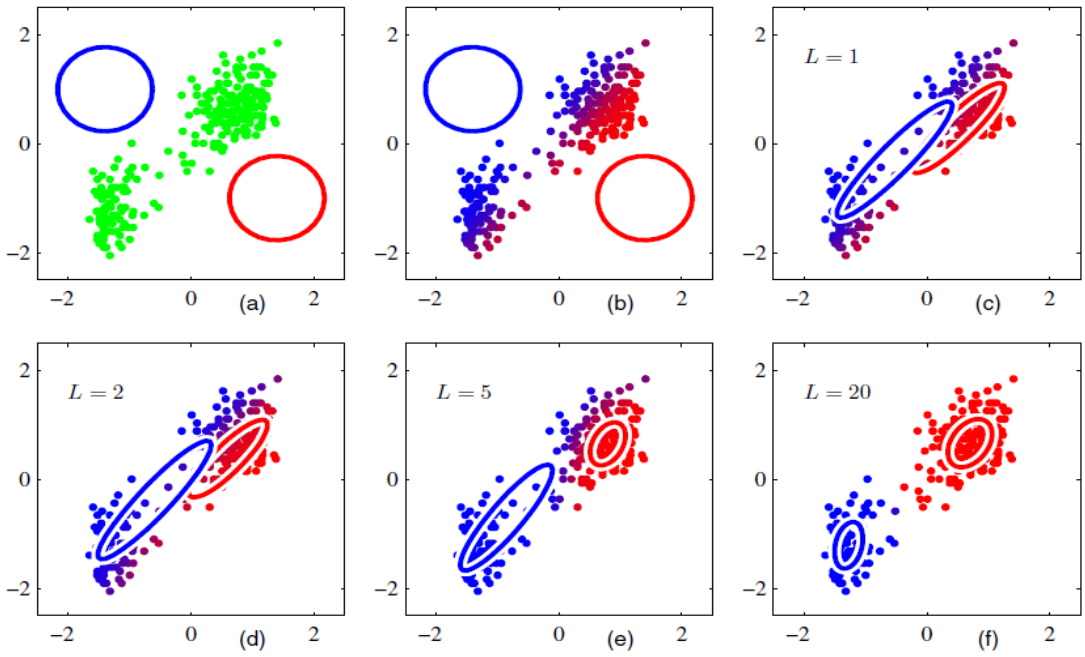
$$\log p(\underline{X}) = \sum_i \log \left[\sum_c \pi_c \ \mathcal{N}(x_i \ ; \ \mu_c, \Sigma_c) \right]$$

- Iterate until convergence
- Convergence guaranteed – another ascent method

Algorithm EM

1. At first choose some initial values for the means, covariances, and mixing coefficients.

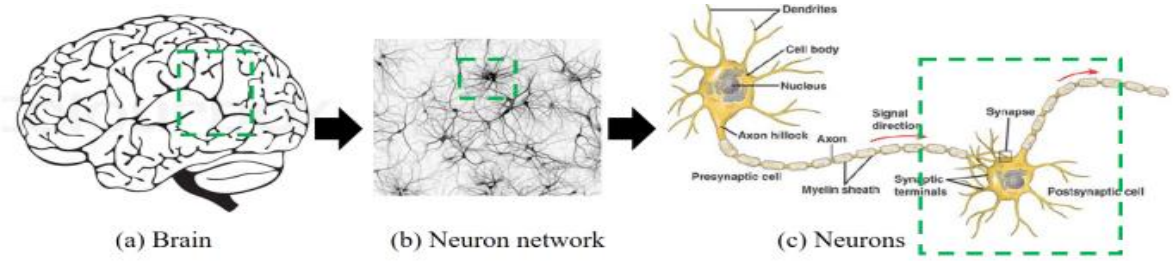
2. Then, call the E step and the M step alternatively, for reasons that will become apparent shortly.
3. In the ***expectation*** step, or E step, we use the current values for the parameters to evaluate the posterior probabilities, or responsibilities.
4. In the ***maximization*** step, or M step, use these probabilities to re-estimate the means, covariances, and mixing coefficients
5. There is an guaranteed increase in the log **likelihood** function (L). At each update, to the parameters resulting from an E step followed by an M step.
6. Thus, the algorithm is deemed to have converged when the change in the log likelihood function, falls below some threshold.



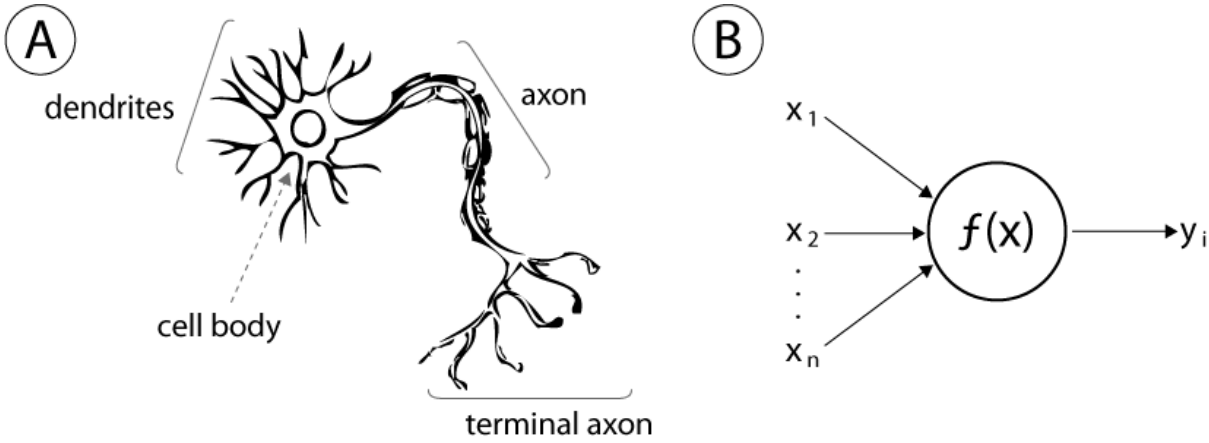
UNIT V

NEURAL NETWORKS

- Computational models Design was inspired by the human brain:
- Algorithms that try to mimic the brain.
- made up of simple processing units (neurons)
- Synaptic connection strengths among neurons are used to store the acquired knowledge.
- Knowledge is acquired by the network from its environment through a learning process



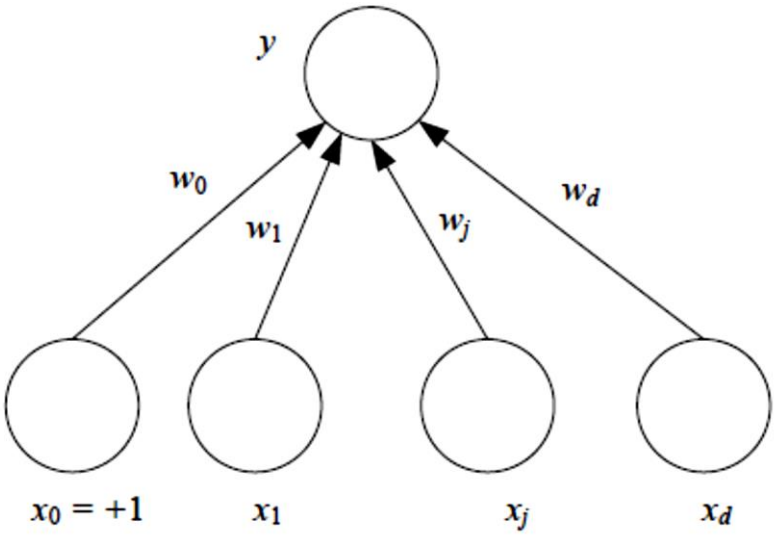
(a) Brain (b) Neuron network (c) Neurons



Perceptron

- First neural network learning model in the 1960's
- Simple and limited (single layer model)
- Still used in some current applications (large business problems, where intelligibility is needed, etc.)

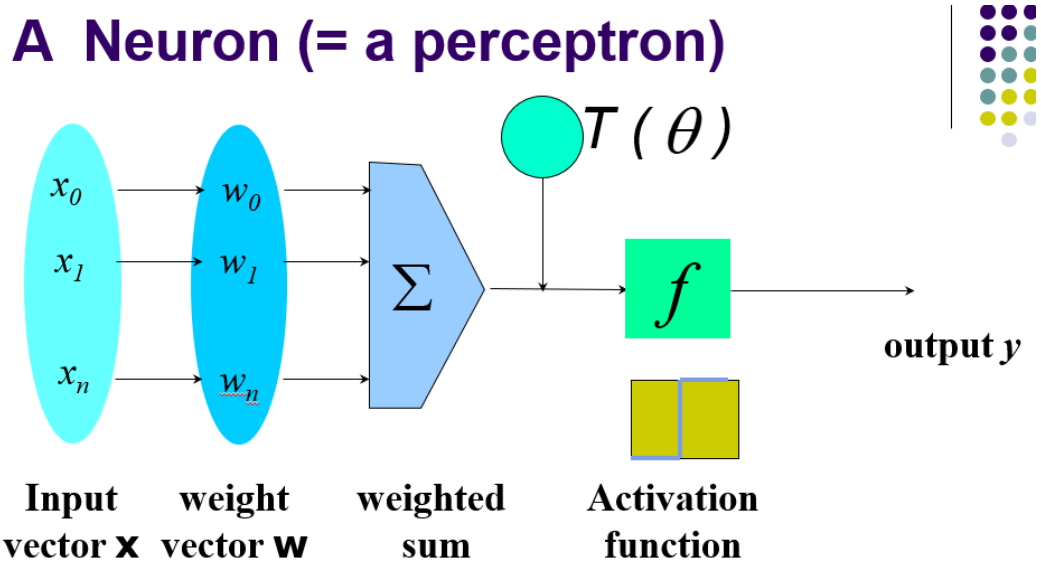
- The *perceptron* is the basic processing element.
- It has inputs that may come from the environment. Associated with each input is a connection weight, or synaptic weight and form output **y** In the simplest case, Output **y** is a weighted sum of the inputs $x_j \in \Re, j = 1, \ldots, d,$ is a *connection weight*, or *synaptic weight* $w_j \in \Re$, to form the output, *y*,
- In the simplest case, Output **y** is a weighted sum of the inputs



Simple Perceptron

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$

- w_0 is the intercept value to make the model more general; it is generally modeled as the weight coming from an extra bias unit, x_0, which is always 1.
- We can write the output of the perceptron as a dot product $y = \mathbf{w}^T \mathbf{x}$



- It is a Basic unit in a neural network
  - N inputs, x_1 ... x_n
  - Weights for each input, w_1 ... w_n
  - A bias input x_0 (constant) and associated weight w_0

- Weighted sum of inputs,

$$y = w_0x_0 + w_1x_1 + \ldots + w_nx_n$$

- A threshold function or activation function,
  - is 1 if y > t,
  - is-1 if y <= t

***Augmented* vectors**

- where $\boldsymbol{w} = [w_0, w_1, \ldots, w_d]^T$ and $\boldsymbol{x} = [1, x_1, \ldots, x_d]^T$ are *augmented* vectors to include also the bias weight and input.
- During testing, with given weights, $\boldsymbol{w}$, for input $\boldsymbol{x}$, we compute the output $\boldsymbol{y}$.
- To implement a given task, we need to *learn* the weights $\boldsymbol{w}$, the parameters of the system, such that correct outputs are generated given the inputs.

θ-Threshold

- Treat threshold like any other weight. No special case. Call it a *bias* since it biases the output up or down.
- the perceptron can separate two classes by checking the sign of the output.
- s(·) as the threshold function

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\boldsymbol{w}^T\boldsymbol{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

- Remember that using a linear discriminant assumes that classes are linearly separable.
- It is assumed that a hyperplane $w^T x = 0$ can be found that separates $x_t \in C_1$ and $x_t \in C_2$.

Perceptron Rule Learning (Convergence Theorem)

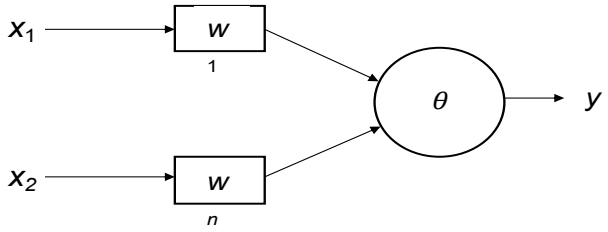
$$\Delta w_i = c(t - z)\, x_i$$

- Where,
  - w_i is the weight from input *i* to perceptron node,
  - c is the learning rate,

- ***t*** is the target for the current instance,
- ***z*** is the current output, and x_i is i^{th} input

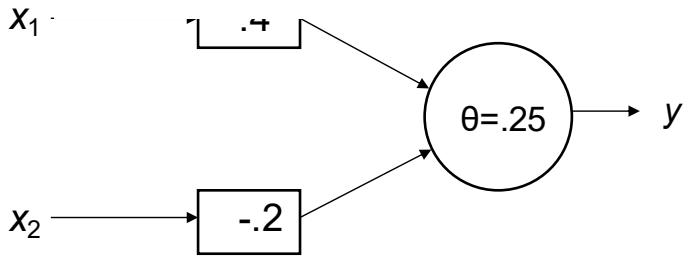
- Create a perceptron node with *n* inputs
- Iteratively apply a pattern from the training set and apply the perceptron rule
- Each iteration through the training set is an *epoch*
- Continue training until total training set error ceases to improve
- **Perceptron Convergence Theorem:** Guaranteed to find a solution in finite time if a solution exists

Example N=2

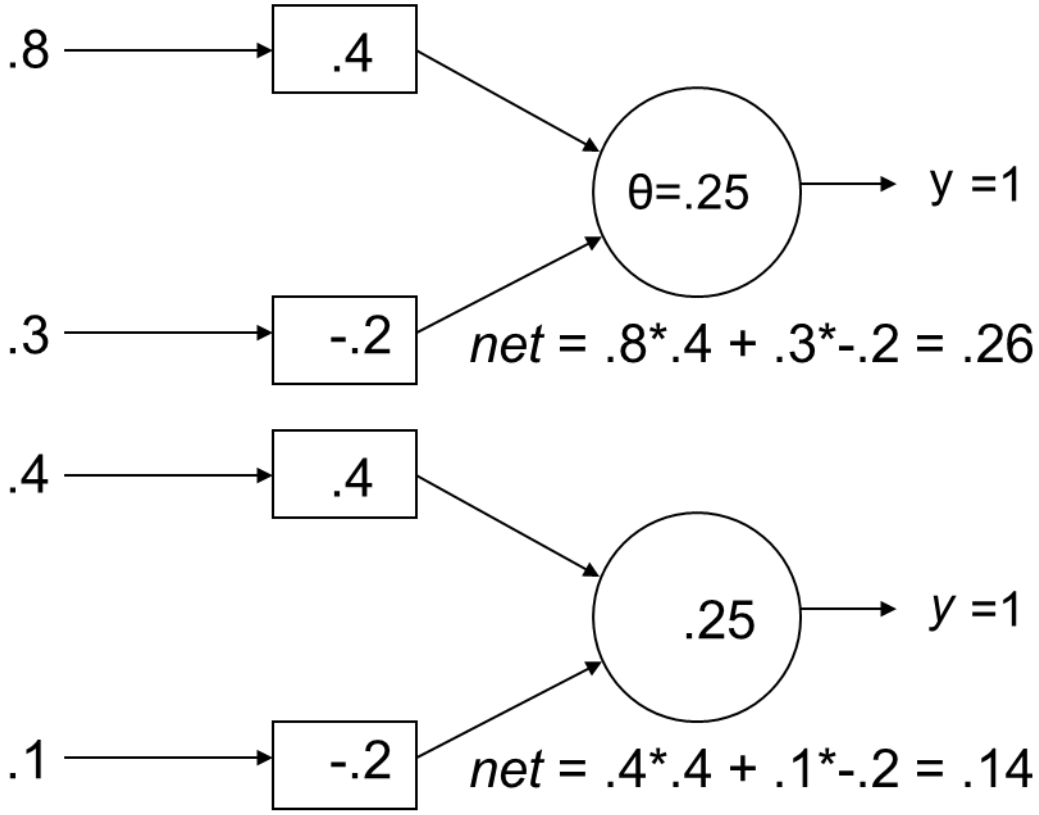


x_1	x_2	t
.8	.3	1
.4	.1	0

$$y = \begin{cases} 1 & if \ \sum_{i=1}^{n} x_i w_i \geq \theta \\ 0 & if \ \sum_{i=1}^{n} x_i w_i < \theta \end{cases}$$

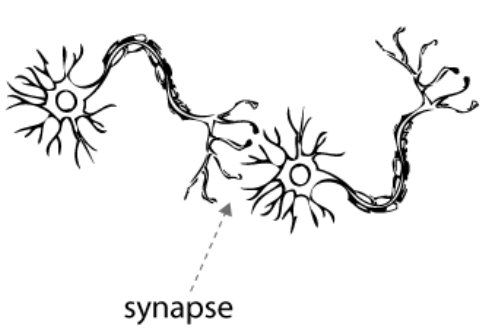


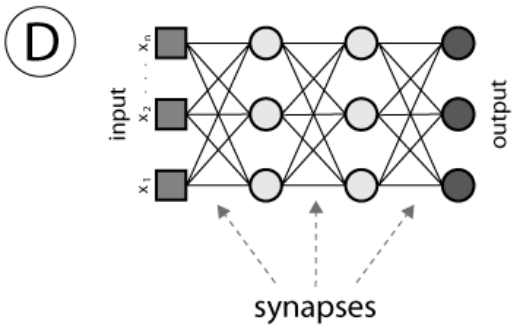
$$y = \begin{cases} 1 & if \ \sum_{i=1}^{n} x_i w_i \geq 0.25 \\ 0 & if \ \sum_{i=1}^{n} x_i w_i < 0.25 \end{cases}$$

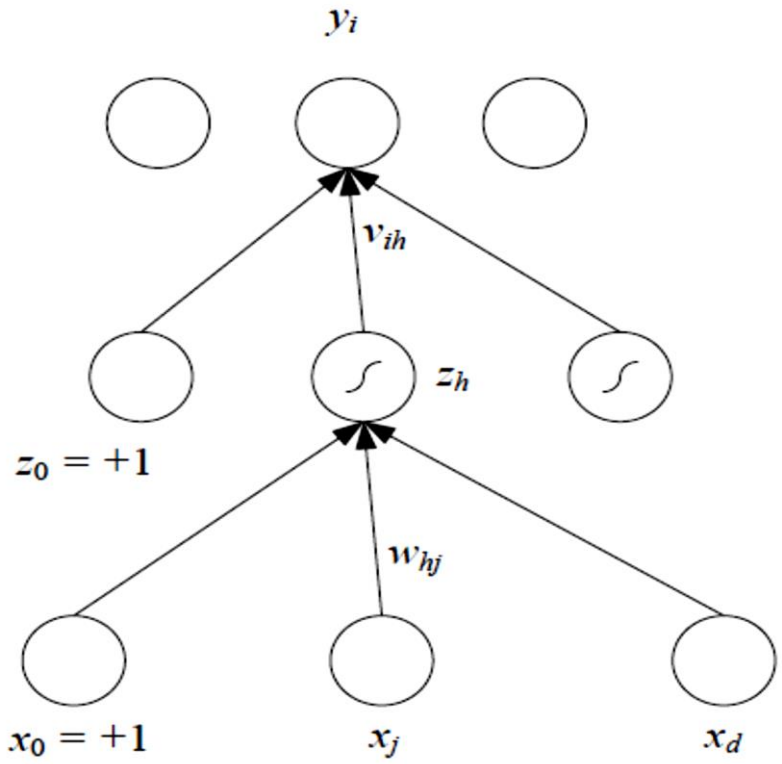


Multilayer perceptron

- A perceptron that has a single layer of weights can only approximate linear functions of the input and cannot solve problems like the XOR, where the discriminant to be estimated is nonlinear.
- These compute a series of transformations, the first layer is the input and the last layer is the output.
- It is used for classification, MLP can implement nonlinear discriminants.







- Input $\boldsymbol{x}$ is fed to the input layer (including the bias), the "activation" propagates in the forward direction, and the values of the hidden units z_h are calculated.
- Each hidden unit is a perceptron by itself and applies the nonlinear sigmoid function to its weighted sum:

$$z_h = \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}, \quad h = 1, \ldots, H$$

- The output *yi* are perceptrons in the second layer taking the hidden units as their inputs

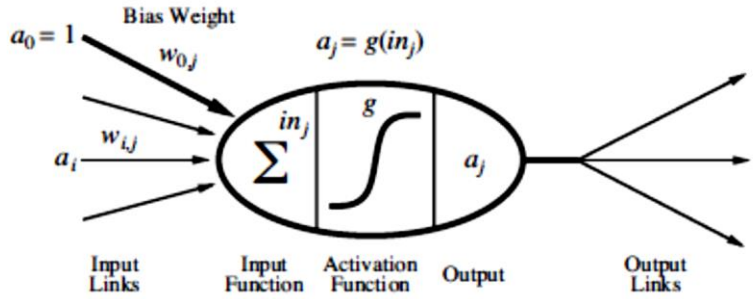
$$y_i = \boldsymbol{v}_i^T \boldsymbol{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

- where there is also a bias unit in the hidden layer, which we denote by *z*0, and *vi*0 are the bias weights. In a two-class discrimination task, there is one sigmoid output unit and when there are $K > 2$ classes, there are K outputs with softmax as the output nonlinearity.
- If the hidden units' outputs were linear, the hidden layer would be of no use: Linear combination of linear combinations is another linear combination.

- Sigmoid is the continuous, differentiable version of thresholding.

Activation Functions

Neural networks are composed of nodes or **units** connected by directed **links**. A link from unit i to unit j serves to propagate the **activation** a_i from i to j.8 Each link also has a numeric **weight** $w_{i,j}$ associated with it, which determines the strength and sign of the connection.



- A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$ where ai is the output activation of unit i and wi,j is the weight on the link from unit i to this unit.
- Just as in linear regression models, each unit has a dummy input a_0 =1 with an associated weight $w_{0,j}$. Each unit j first computes a weighted sum of its inputs:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i$$

- Then it applies an **activation function** g to this sum to derive the output

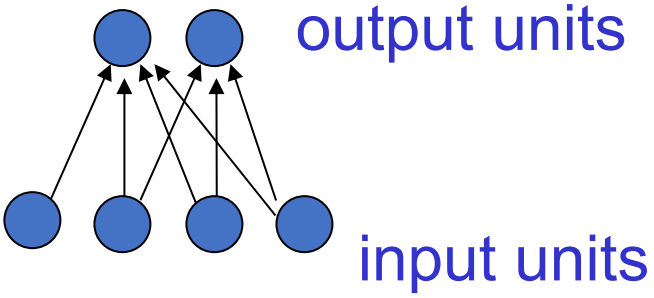
$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

- The activation function g is typically either a hard threshold, in which case the unit is called a **perceptron**,

Types of Networks

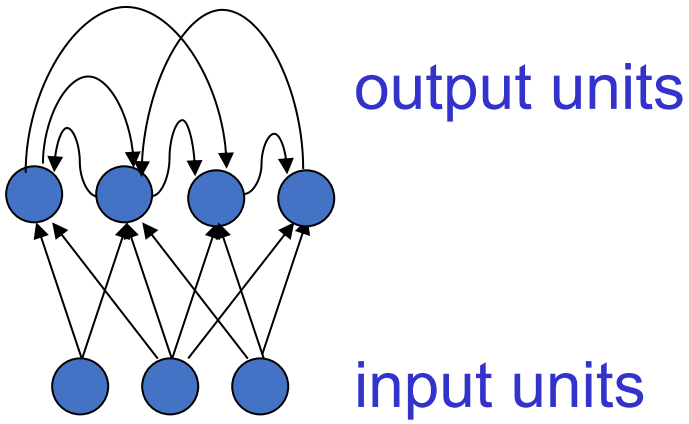
Feedforward networks

- These compute a series of transformations
- Typically, the first layer is the input and the last layer is the output.
- In feed-forward networks with intermediate or *hidden layers* between the input and the output layers.



Recurrent networks

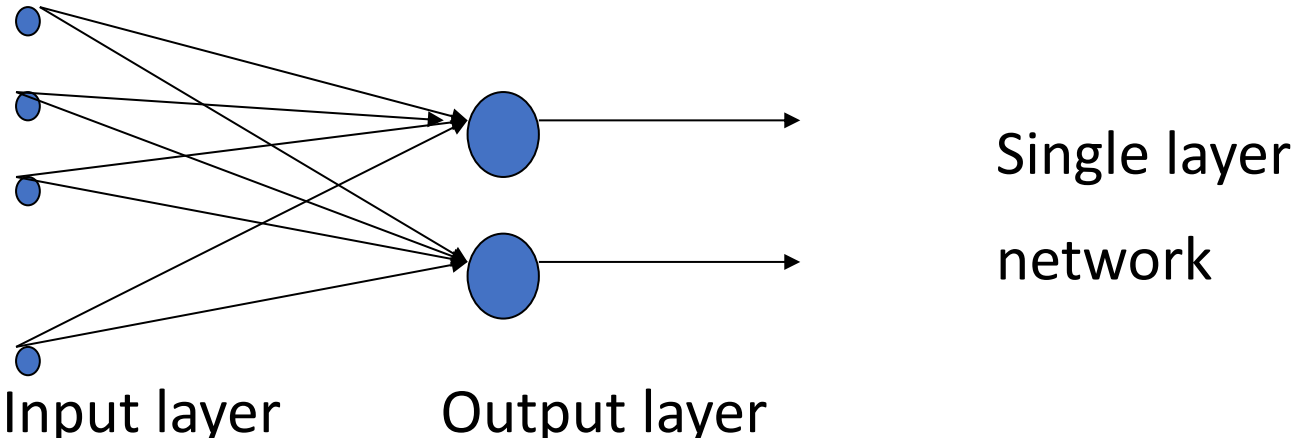
- These have directed cycles in their connection graph. They can have complicated dynamics.
- More biologically realistic.



Feed-Forward Networks

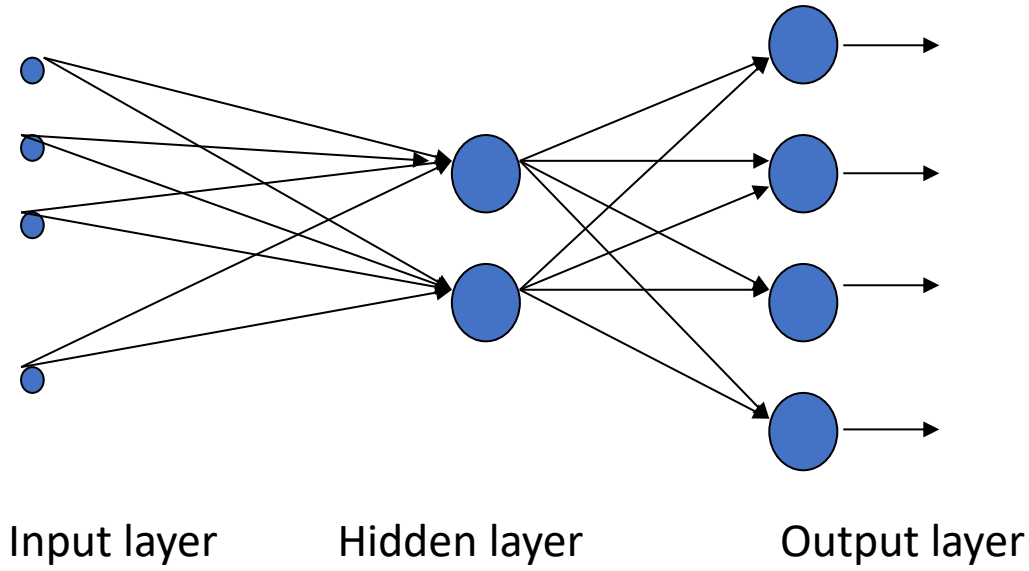
Single layer feed-forward networks

- Input layer projecting into the output layer



Multi-layer feed-forward networks

- One or more hidden layers. Input projects only from previous layers onto a layer.

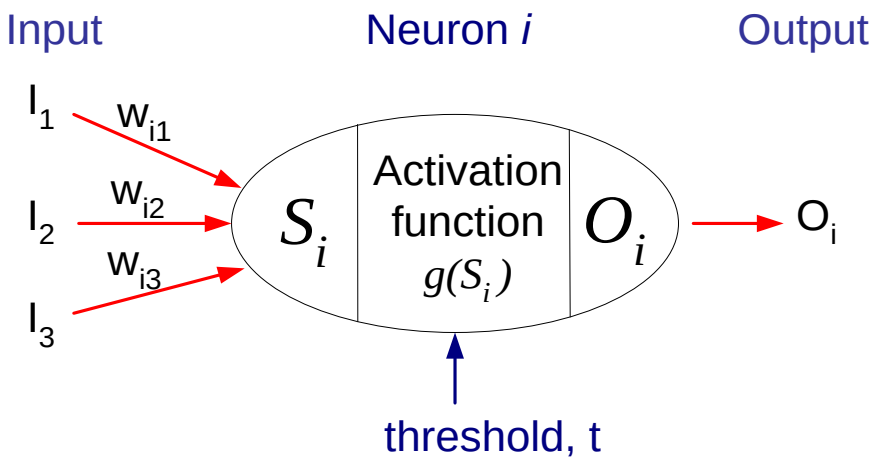


Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links
- Output node sums up each of its input value according to the weights of its links
- Compare output node against some threshold t

Perceptron Model

$$Y = I(\sum_i w_i x_i - t)$$

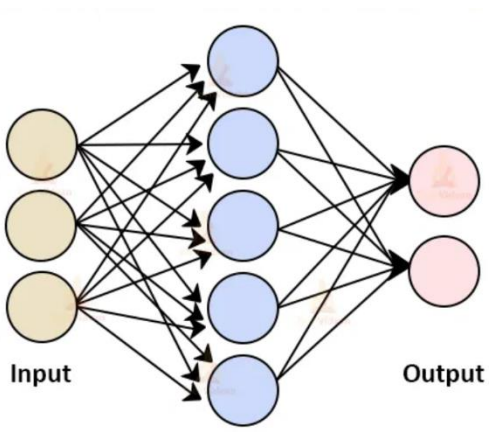


Algorithm for learning ANN

- Initialize the weights (w_0, w_1, …, w_k)
- Adjust the weights in such a way that the output of ANN is consistent with class labels of training examples
    - Error function:
    - Find the weights w_i's that minimize the above error function
      e.g., gradient descent, backpropagation algorithm

Type of Network

A neural network has many layers and each layer performs a specific function, and as the complexity of the model increases, the number of layers also increases that why it is known as the multi-layer perceptron.

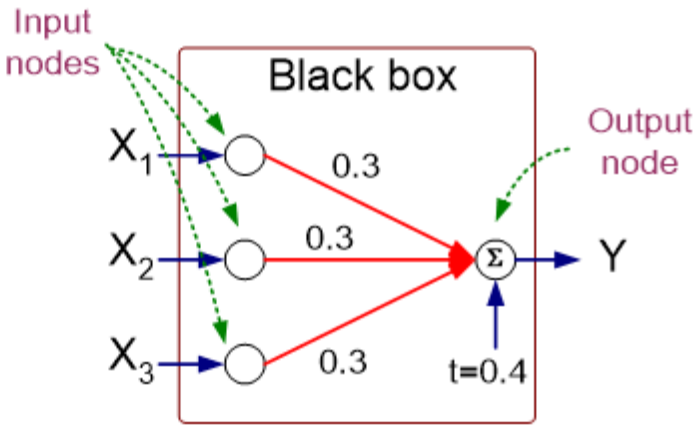


Hidden Layer

- At First, information is feed into the input layer which then transfers it to the hidden layers, and interconnection between these two layers assign weights to each input randomly at the initial point.
- and then bias is added to each input neuron and after this, the weighted sum which is a combination of weights and bias is passed through the activation function.
- Activation Function has the responsibility of which node to fire for feature extraction and finally output is calculated.
- This whole process is known as Forward Propagation. After getting the output model to compare it with the original output and the error is known and finally, weights are updated in backward propagation to reduce the error and this process continues for a certain number of epochs (iteration).

- Finally, model weights get updated and prediction is done.

X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0



$$Y = I(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4 > 0)$$

$$\text{where } I(z) = \begin{cases} 1 & \text{if } z \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Gradient descent optimization

The simplest approach to using gradient information is to choose the weight update in to comprise a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

where the parameter $\eta > 0$ is known as the *learning rate*.

- After each such update, the gradient is re-evaluated for the new weight vector and the process repeated.
- Note that the error function is defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate ∇E.
- Techniques that use the whole data set at once are called *batch* methods.
- At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as *gradient descent* or *steepest descent*.
- In order to find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times,
- At each time using a different randomly chosen starting point, and comparing the resulting performance on an independent validation set.
- Error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}).$$

Batch Perceptron gradient Algorithm

- *Randomly initialize weights* w_1, w_2, w_3
- Get your dataset
- *For i=1 to R* $\nabla_i = y_i - w^\tau x_i$
- *For j=1 to m* $W_j = W_j + \eta \sum_{i=1}^{r} \nabla_i x_{ij}$
- *If* $\sum_{i=1}^{r} \nabla_i$ stops improving stop else go to step 3

On-line gradient descent (*sequential gradient descent* or *stochastic gradient descent*)

- It makes an update to the weight vector based on one data point at a time,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$

- This update is repeated by cycling through the data either in sequence or by selecting points at random with replacement.
- There are of course intermediate scenarios in which the updates are based on batches of data points.
- One advantage of on-line methods compared to batch methods is that the former handle redundancy in the data much more efficiently.
- For example, consider an extreme case, in which we take a data set and double its size by duplicating every data point.
- Note that this simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function.
- Batch methods will require double the computational effort to evaluate the batch error function gradient, whereas online methods will be unaffected.
- Another property of on-line gradient descent is the possibility of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

Network Training Procedures

Improving Convergence

Gradient descent has various advantages. It is simple. It is local; namely, the change in a weight uses only the values of the presynaptic and postsynaptic units and the error (suitably backpropagated). When online training is used, it does not need to store the training set and can adapt as the task to be learned changes. Because of these reasons, it can be (and is) implemented in hardware. But by itself, gradient descent converges slowly. When learning time is important, one can use more sophisticated optimization methods. Bishop discusses in detail the application of conjugate gradient and second-order methods to the training of multilayer perceptrons. However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications.

Momentum

Let us say *wi* is any weight in a multilayer perceptron in any layer, including the biases. At each parameter update, successive Δwti values may be so different that large oscillations may occur and slow convergence. *t* is the time index that is the epoch number in batch learning and the iteration number in online learning. The idea is to take a running average by incorporating the previous update in the current change as if there is a *momentum* due to previous updates:

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1}$$

α is generally taken between 0.5 and 1.0. This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence. The disadvantage is that the past $\Delta wt-1i$ values should be stored in extra memory.

Adaptive Learning Rate

In gradient descent, the learning factor η determines the magnitude of change to be made in the parameter. It is generally taken between 0.0 and 1.0, mostly less than or equal to 0.2. It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta\eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^t \\ -b\eta & \text{otherwise} \end{cases}$$

Thus we increase η by a constant amount if the error on the training set decreases and decrease it geometrically if it increases. Because E may oscillate from one epoch to another, it is a better idea to take the average of the past few epochs as Et .

Overtraining

A multilayer perceptron with d inputs, H hidden units, and K outputs has $H(d+1)$ weights in the first layer and $K(H+1)$ weights in the second layer. Both the space and time complexity of an MLP is $O(H \cdot (K + d))$. When e denotes the number of training epochs, training time complexity is $O(e \cdot H \cdot (K + d))$.

In an application, d and K are predefined and H is the parameter that we play with to tune the complexity of the model. For example, we have previously seen this phenomenon in the case of polynomial regression where we noticed that in the presence of noise or small samples, increasing the polynomial order leads to worse generalization. Similarly in an MLP, when the number of hidden units is large, the generalization accuracy deteriorates, and the bias/variance dilemma also holds for the MLP, as it does for any statistical estimator. A similar behavior happens when training is continued too long: As more training epochs are made, the error on the training set decreases, but the error on the validation set starts to increase beyond a certain point. Remember that initially all the weights are close to 0 and thus have little effect. As training continues, the most important weights start moving away from 0 and are utilized. But if training is continued further on to get less and less error on the training set, almost all weights are updated away from 0 and effectively become parameters.

Thus as training continues, it is as if new parameters are added to the system, increasing the complexity and leading to poor generalization. Learning should be *stopped early* to alleviate this problem of *overtraining*. The optimal point to stop training, and the optimal number of hidden units, is determined through cross-validation, which involves testing the network's performance on validation data unseen during training. Because of the nonlinearity, the error function has many minima and gradient descent converges to the nearest minimum. To be able to assess expected error, the same network is trained a number of times starting from different initial weight values, and the average of the validation error is computed.

Back Propagation

$$\mathrm{Out}(\mathrm{x}) = g\left(\sum_j W_j g\left(\sum_k w_{jk} x_k\right)\right)$$

Find the set of weights Wj, wjk

to minimize

$$\sum_i \left(y_i - \mathrm{Out}(\mathrm{x}_i)\right)^2$$

By gradient descent,

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to minimize the mean squared error between the network's prediction and the actual target value
- Modifications are made in the "backwards" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "backpropagation"

Steps

- Initialize weights (to small random #s) and biases in the network
- Propagate the inputs forward (by applying activation function)
- Backpropagate the error (by updating weights and biases)
- Terminating condition (when error is very small, etc.)

Error Backpropagation

- The goal is to find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network.
- It can be achieved using a local message passing scheme in which information is sent alternately forwards and backwards through the network and is known as *error backpropagation*, or sometimes simply as *backprop*.

In the first stage, the derivatives of the error function with respect to the weights must be evaluated. The important contribution of the backpropagation technique is in providing a computationally efficient method for evaluating such derivatives. Because it is at this stage that errors are propagated backwards through the network, If the evaluation of derivatives is used it is termed as backpropagation. the propagation of errors backwards through the network in

order to evaluate derivatives, can be applied to many other kinds of network and not just the multilayer perceptron. It can also be applied to error functions other that just the simple sum-of-squares, and to the evaluation of other derivatives such as the Jacobian and Hessian matrices.

In the second stage, the derivatives are then used to compute the adjustments to be made to the weights. The simplest such technique, and the one originally considered, involves gradient descent. The weight adjustment using the calculated derivatives can be tackled using a variety of optimization schemes, many of which are substantially more powerful than simple gradient descent. It is important to recognize that the two stages are distinct.

Error Backpropagation Algorithm

1. Apply an input vector **x***n* to the network and forward propagate through the network using

$$a_j = \sum_i w_{ji} z_i$$

and

$$z_j = h(a_j).$$

to find the activations of all the hidden and output units.

2. Evaluate the δk for all the output units using

$$\delta_k = y_k - t_k$$

3. Backpropagate the δ's using

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

to obtain δj for each hidden unit in the network.

4. Use

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i.$$

to evaluate the required derivatives.

For batch methods, the derivative of the total error *E* can then be obtained by repeating the above steps for each pattern in the training set and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}.$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $h(\cdot)$. The derivation is easily generalized, however, to allow different units to have individual activation functions, simply by keeping track of which form of $h(\cdot)$ goes with which unit.

Unit Saturation (aka the vanishing gradient problem)

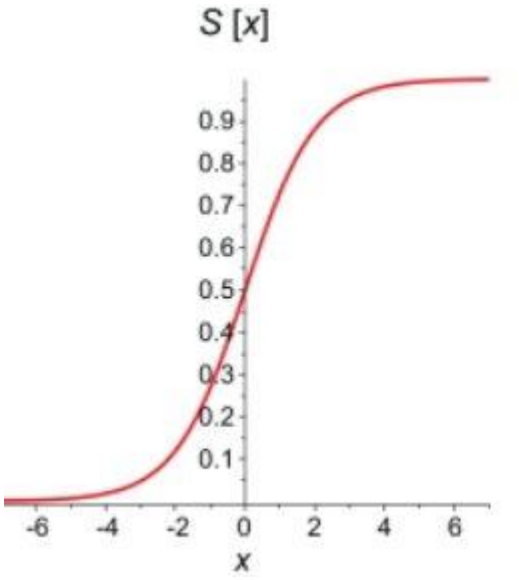
The sigmoid function is one of the most popular activations functions used for developing deep neural networks. The use of sigmoid function restricted the training of deep neural networks because it caused the vanishing gradient problem. This caused the neural network to learn at a slower pace or in some cases no learning at all.

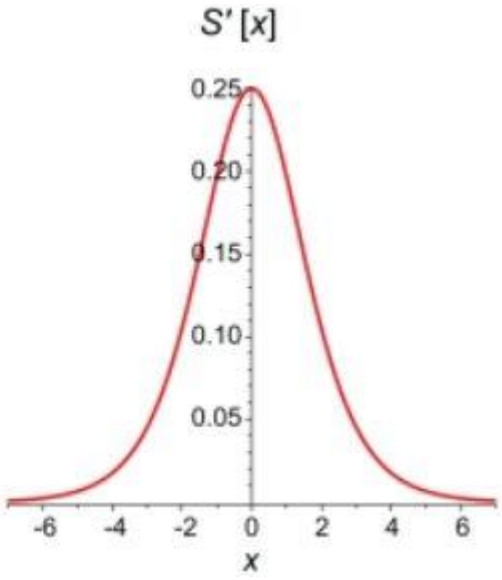
Sigmoid function

Sigmoid functions are used frequently in neural networks to activate neurons. It is a logarithmic function with a characteristic S shape. The output value of the function is between 0 and 1. The sigmoid function is used for activating the output layers in binary classification problems.

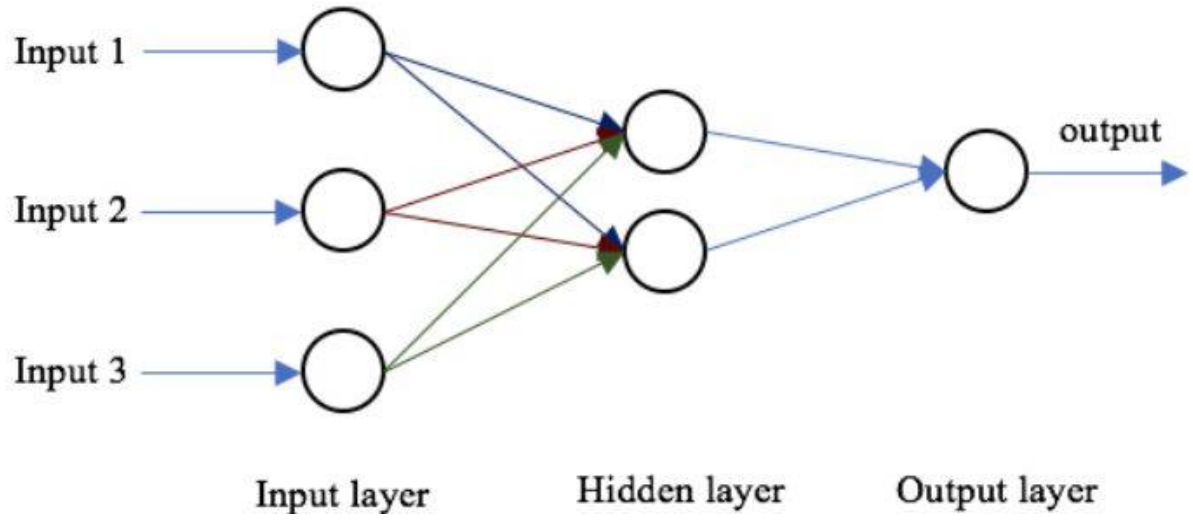
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

On the graph below you can see a comparison between the sigmoid function itself and its derivative. First derivatives of sigmoid functions are bell curves with values ranging from 0 to 0.25.





Our knowledge of how neural networks perform forward and backpropagation is essential to understanding the vanishing gradient problem.



Back Propagation

As the network generates an output, the loss function(C) indicates how well it predicted the output. The network performs back propagation to minimize the loss. A back propagation method minimizes the loss function by adjusting the weights and biases of the neural network. In this method, the gradient of the loss function is calculated with respect to each weight in the network.

In back propagation, the new weight(w_{new}) of a node is calculated using the old weight(w_{old}) and product of the learning rate(η) and gradient of the loss function Vanishing Gradient Problem, $(\frac{\partial C}{\partial w})$ Explained.

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$$

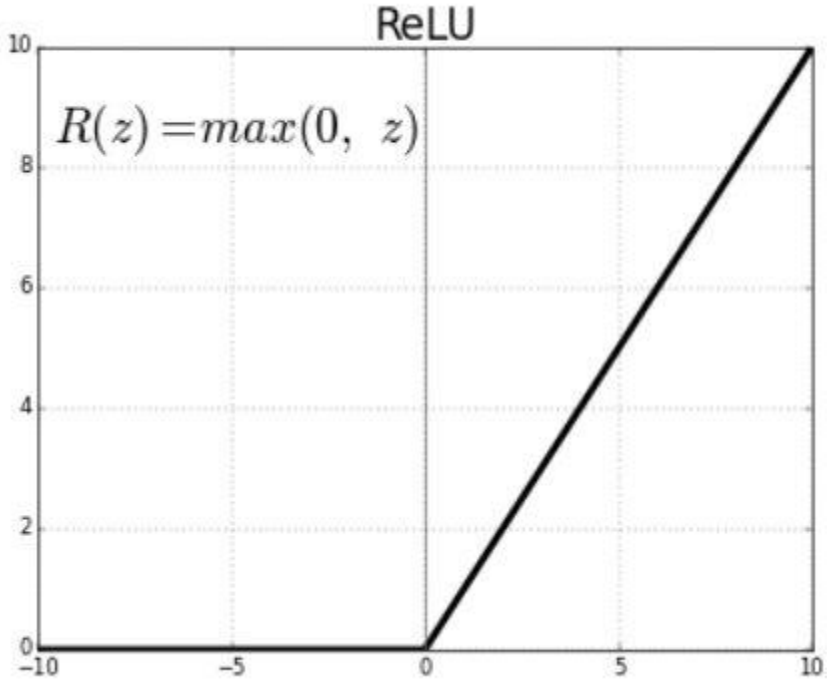
With the chain rule of partial derivatives, we can represent gradient of the loss function as a product of gradients of all the activation functions of the nodes with respect to their weights. Therefore, the updated weights of nodes in the network depend on the gradients of the activation functions of each node.

For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25. When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes. We call this the vanishing gradient problem.

With shallow networks, sigmoid function can be used as the small value of gradient does not become an issue. When it comes to deep networks, the vanishing gradient could have a significant impact on performance. The weights of the network remain unchanged as the derivative vanishes. During back propagation, a neural network learns by updating its weights and biases to reduce the loss function. In a network with vanishing gradient, the weights cannot be updated, so the network cannot learn. The performance of the network will decrease as a result.

Method to overcome the problem

The vanishing gradient problem is caused by the derivative of the activation function used to create the neural network. The simplest solution to the problem is to replace the activation function of the network. Instead of sigmoid, use an activation function such as ReLU. Rectified Linear Units (ReLU) are activation functions that generate a positive linear output when they are applied to positive input values. If the input is negative, the function will return zero.

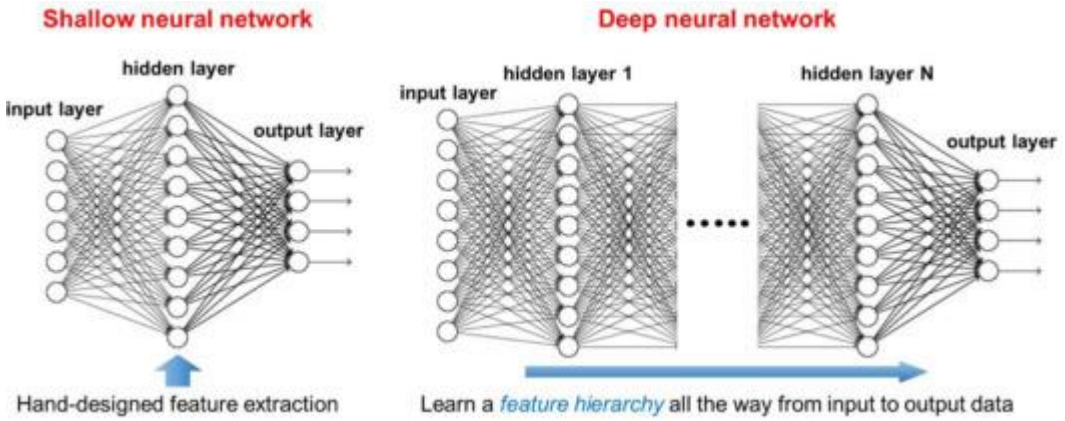


The derivative of a ReLU function is defined as 1 for inputs that are greater than zero and 0 for inputs that are negative. The graph shared below indicates the derivative of a ReLU function

Shallow Network & Deep Network

Shallow Network	Deep Network
A shallow neural network has only one hidden layer between the input and output layers, while a deep neural network has multiple hidden layers. One way to think about it is like a hierarchy of decision-making. Just like how a human brain makes decisions by processing information in layers, a deep neural network learns to make decisions by processing information through multiple hidden layers. On the other hand, a shallow network is like having just one layer of decision-making, which might not be enough to capture the complexity of the problem at hand.	A deep network, on the other hand, can capture more complex patterns in the data and potentially achieve higher accuracy, but it is more computationally expensive to train and may require more data to avoid overfitting. Additionally, deep networks can be more challenging to design and optimize than shallow networks. When it comes to deep networks, the vanishing gradient could have a significant impact on performance. The weights of the network remain unchanged as the derivative vanishes. During back propagation, a neural

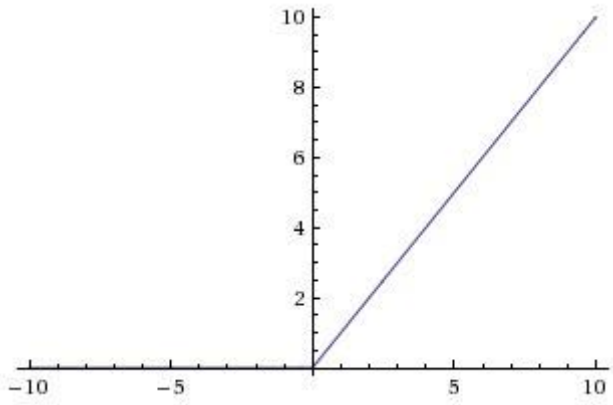
A shallow network might be used for simple tasks like image classification, while a deep network might be used for more complex tasks like image segmentation or natural language processing. The main advantage of a shallow network is that it is computationally less expensive to train, and can be sufficient for simple tasks. However, it may not be powerful enough to capture complex patterns in the data.	network learns by updating its weights and biases to reduce the loss function. In a network with vanishing gradient, the weights cannot be updated, so the network cannot learn. The performance of the network will decrease as a result.



Rectified Linear Units (ReLU) in Deep Learning

The Rectified Linear Unit is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. So it can be written as f(x)=max(0,x).

Graphically it looks like this



It's surprising that such a simple function (and one composed of two linear pieces) can allow your model to account for non-linearities and interactions so well. But the ReLU function works great in most applications, and it is very widely used as a result.

Interactions and Non-linearities

Activation functions serve two primary purposes: 1) Help a model account for **interaction effects**. What is an interactive effect? It is when one variable A affects a prediction differently depending on the value of B. For example, if my model wanted to know whether a certain body weight indicated an increased risk of diabetes, it would have to know an individual's height. Some bodyweights indicate elevated risks for short people, while indicating good health for tall people. So, the **effect of body weight on diabetes risk depends on height**, and we would say that **weight and height have an interaction effect**.

2) Help a model account for **non-linear effects.** This just means that if I graph a variable on the horizontal axis, and my predictions on the vertical axis, it isn't a straight line. Or said another way, the effect of increasing the predictor by one is different at different values of that predictor.

ReLU's captures Interactions and Non-Linearities

Interactions: Imagine a single node in a neural network model. For simplicity, assume it has two inputs, called A and B. The weights from A and B into our node are 2 and 3 respectively. So the node output is f(2A+3B). We'll use the ReLU function for our f. So, if 2A+3B is positive, the output value of our node is also 2A+3B. If 2A+3B is negative, the output value of our node is 0.

For concreteness, consider a case where A=1 and B=1. The output is 2A+3B, and if A increases, then the output increases too. On the other hand, if B=-100 then the output is 0, and if A increases moderately, the output remains 0. So A might increase our output, or it might not. It just depends what the value of B is.

This is a simple case where the node captured an interaction. As you add more nodes and more layers, the potential complexity of interactions only increases. But you should now see how the activation function helped capture an interaction.

Non-linearities: A function is non-linear if the slope isn't constant. So, the ReLU function is non-linear around 0, but the slope is always either 0 (for negative values) or 1 (for positive values). That's a very limited type of non-linearity.

But two facts about deep learning models allow us to create many different types of non-linearities from how we combine ReLU nodes.

First, most models include a **bias** term for each node. The bias term is just a constant number that is determined during model training. For simplicity, consider a node with a single input called A, and a bias. If the bias term takes a value of 7, then the node output is f(7+A). In this case, if A is less than -7, the output is 0 and the slope is 0. If A is greater than -7, then the node's output is 7+A, and the slope is 1.

So the bias term allows us to move where the slope changes. So far, it still appears we can have only two different slopes.

However, real models have many nodes. Each node (even within a single layer) can have a different value for it's bias, so each node can change slope at different values for our input.

When we add the resulting functions back up, we get a combined function that changes slopes in many places.

These models have the flexibility to produce non-linear functions and account for interactions well (if that will giv better predictions). As we add more nodes in each layer (or more convolutions if we are using a convolutional model) the model gets even greater ability to represent these interactions and non-linearities.

However researchers had great difficulty building models with many layers when using the tanh function. It is relatively flat except for a very narrow range (that range being about -2 to 2). The derivative of the function is very small unless the input is in this narrow range, and this flat derivative makes it difficult to improve the weights through gradient descent. This problem gets worse as the model has more layers. This was called the **vanishing gradient problem**.

The ReLU function has a derivative of 0 over half it's range (the negative numbers). For positive inputs, the derivative is 1.

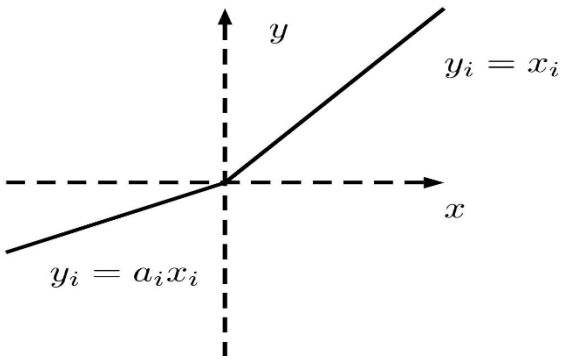
When training on a reasonable sized batch, there will usually be some data points giving positive values to any given node. So the average derivative is rarely close to 0, which allows gradient descent to keep progressing.

Leaky ReLU

There are many similar alternatives which also work well. The Leaky ReLU is one of the most well known. It is the same as ReLU for positive numbers. But instead of being 0 for all negative values, it has a constant slope (less than 1.).

That slope is a parameter the user sets when building the model, and it is frequently called α. For example, if the user sets α=0.3=0.3, the activation function is f(x) = max(0.3*x, x). This has the theoretical advantage that, by being influenced by x at all values, it may be make more complete use of the information contained in x.

Their are other alternatives, but both practitioners and researchers have generally found insufficient benefit to justify using anything other than ReLU.



Hyperparameter Tuning

A Machine Learning model is defined as a mathematical model with a number of parameters that need to be learned from the data. By training a model with existing data, we are able to fit the model parameters. However, there is another kind of parameter, known as *Hyperparameters*, that cannot be directly learned from the regular training process. They are usually fixed before the actual training process begins. These parameters express important properties of the model such as its complexity or how fast it should learn.

Some examples of model hyperparameters include:

1. The penalty in Logistic Regression Classifier i.e. L1 or L2 regularization
2. The learning rate for training a neural network.
3. The C and sigma hyperparameters for support vector machines.
4. The k in k-nearest neighbors.

The aim of this article is to explore various strategies to tune hyperparameters for Machine learning models.

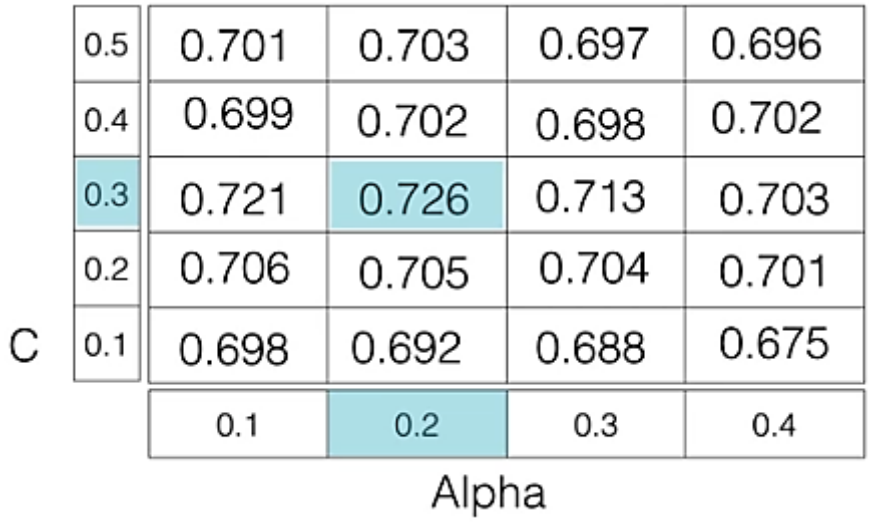
Models can have many hyperparameters and finding the best combination of parameters can be treated as a search problem. The two best strategies for Hyperparameter tuning are:

GridSearchCV

In GridSearchCV approach, the machine learning model is evaluated for a range of hyperparameter values. This approach is called GridSearchCV, because it searches for the best set of hyperparameters from a grid of hyperparameters values.

For example, if we want to set two hyperparameters C and Alpha of the Logistic Regression Classifier model, with different sets of values. The grid search technique will construct many versions of the model with all possible combinations of hyperparameters and will return the best one.

As in the image, for C = [0.1, 0.2, 0.3, 0.4, 0.5] and Alpha = [0.1, 0.2, 0.3, 0.4]. For a combination of **C=0.3 and Alpha=0.2**, the performance score comes out to be **0.726(Highest)**, therefore it is selected.



RandomizedSearchCV

RandomizedSearchCV solves the drawbacks of GridSearchCV, as it goes through only a fixed number of hyperparameter settings. It moves within the grid in a random fashion to find the best set of hyperparameters. This approach reduces unnecessary computation.

Batch Normalization

One of the most common problems of data science professionals is to avoid over-fitting. Have you come across a situation when your model is performing very well on the training data but is unable to predict the test data accurately. The reason is your model is overfitting. The solution to such a problem is regularization.
The regularization techniques help to improve a model and allows it to converge faster. We have several regularization tools at our end, some of them are early stopping, dropout, weight initialization techniques, and batch normalization. The regularization helps in preventing the over-fitting of the model and the learning process becomes more efficient.

Here, in this article, we are going to explore one such technique, batch normalization in detail.

Normalization

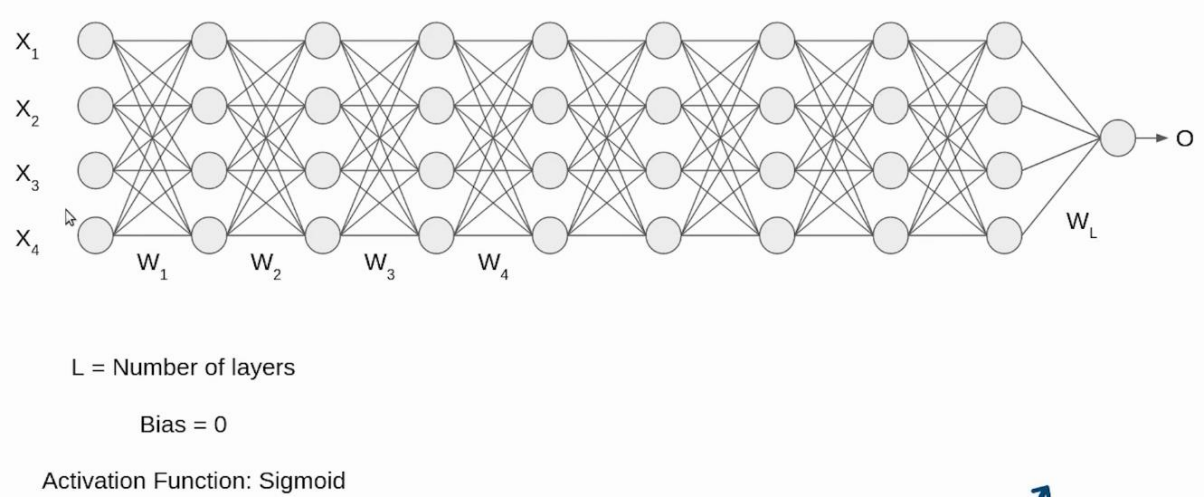
Normalization is a data pre-processing tool used to bring the numerical data to a common scale without distorting its shape.

Generally, when we input the data to a machine or deep learning algorithm we tend to change the values to a balanced scale. The reason we normalize is partly to ensure that our model can generalize appropriately.

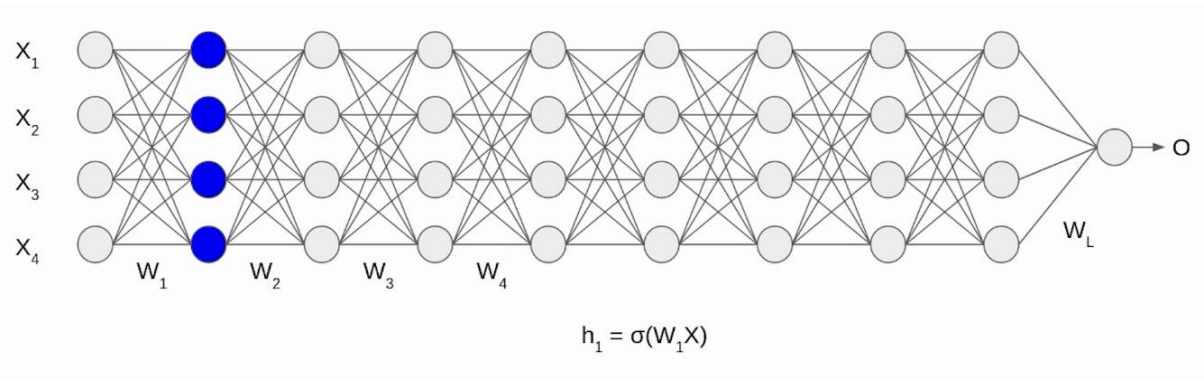
Now coming back to Batch normalization, it is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.

But what is the reason behind the term “Batch” in batch normalization? A typical neural network is trained using a collected set of input data called **batch**. Similarly, the normalizing process in batch normalization takes place in batches, not as a single input.

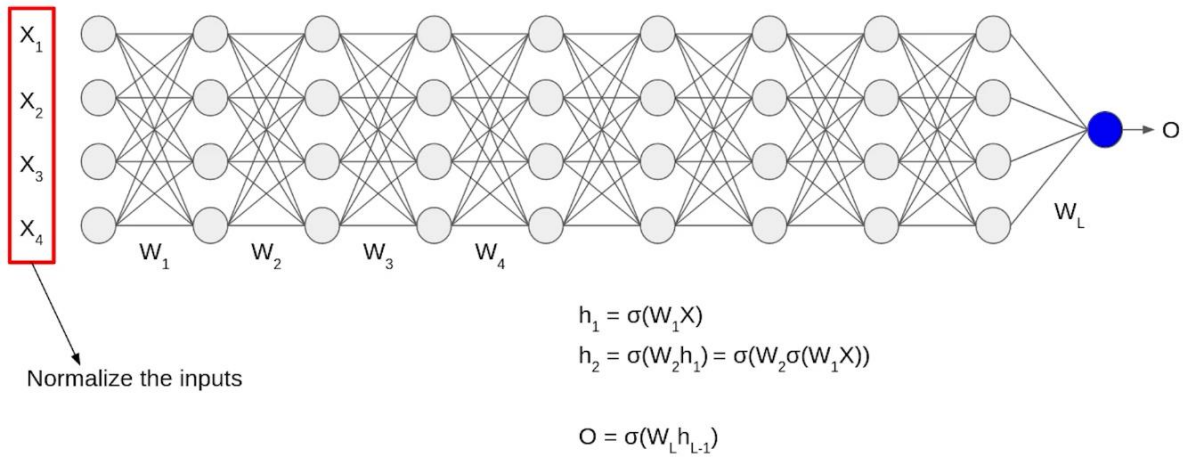
Let’s understand this through an example, we have a deep neural network as shown in the following image.



Initially, our inputs X1, X2, X3, X4 are in normalized form as they are coming from the pre-processing stage. When the input passes through the first layer, it transforms, as a sigmoid function applied over the dot product of input X and the weight matrix W.



Similarly, this transformation will take place for the second layer and go till the last layer L as shown in the following image.



Although, our input X was normalized with time the output will no longer be on the same

scale. As the data go through multiple layers of the neural network and L activation functions are applied, it leads to an internal co-variate shift in the data.

Normalization working

Since by now we have a clear idea of why we need Batch normalization, let's understand how it works. It is a two-step process. First, the input is normalized, and later rescaling and offsetting is performed.

Normalization of the Input

Normalization is the process of transforming the data to have a mean zero and standard deviation one. In this step we have our batch input from layer h, first, we need to calculate the mean of this hidden activation.

$$\mu = \frac{1}{m} \sum h_i$$

Here, m is the number of neurons at layer h.

Once we have meant at our end, the next step is to calculate the standard deviation of the hidden activations.

$$\sigma = \left[\frac{1}{m} \sum (h_i - \mu)^2 \right]^{1/2}$$

Further, as we have the mean and the standard deviation ready. We will normalize the hidden activations using these values. For this, we will subtract the mean from each input and divide the whole value with the sum of standard deviation and the smoothing term (ε).

The smoothing term(ε) assures numerical stability within the operation by stopping a division by a zero value.

$$h_{i(norm)} = \frac{(h_i - \mu)}{\sigma + \varepsilon}$$

Rescaling of Offsetting

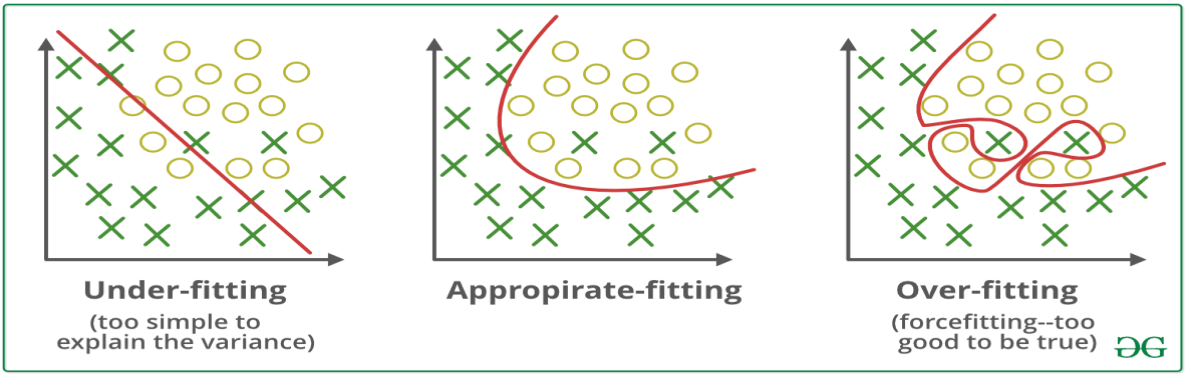
In the final operation, the re-scaling and offsetting of the input take place. Here two components of the BN algorithm come into the picture, γ(gamma) and β (beta). These parameters are used for re-scaling (γ) and shifting(β) of the vector containing values from the previous operations.

$$h_i = \gamma h_{i(norm)} + \beta$$

These two are learnable parameters, during the training neural network ensures the optimal values of γ and β are used. That will enable the accurate normalization of each batch.

Regularization

Overfitting is a phenomenon that occurs when a Machine Learning model is constraint to training set and not able to perform well on unseen data.



Regularization is a technique used to reduce the errors by fitting the function appropriately on the given training set and avoid overfitting.
The commonly used regularization techniques are :

1. L1 regularization
2. L2 regularization
3. Dropout regularization

This article focus on L1 and L2 regularization.

A regression model which uses technique is called

- **L1 Regularization LASSO (Least Absolute Shrinkage and Selection Operator) regression**.

  Lasso Regression adds *"absolute value of magnitude"* of coefficient as penalty term to the loss function(L).

$$||\mathbf{w}||_1 = |w_1| + |w_2| + ... + |w_N|$$

- **L2 Regularization Ridge regression**.

Ridge regression adds "*squared magnitude*" of coefficient as penalty term to the loss function(L).

$$\|\mathbf{w}\|_2 = \left(|w_1|^2 + |w_2|^2 + \ldots + |w_N|^2\right)^{\frac{1}{2}}$$

during Regularization the output function(y_hat) does not change. The change is only in the loss function.

The output function:

$$\hat{y} = w_1 x_1 + w_2 x_2 + ... + w_N x_N + b$$

The loss function before regularization:

$$Loss = Error(y, \hat{y})$$

The loss function after regularization:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

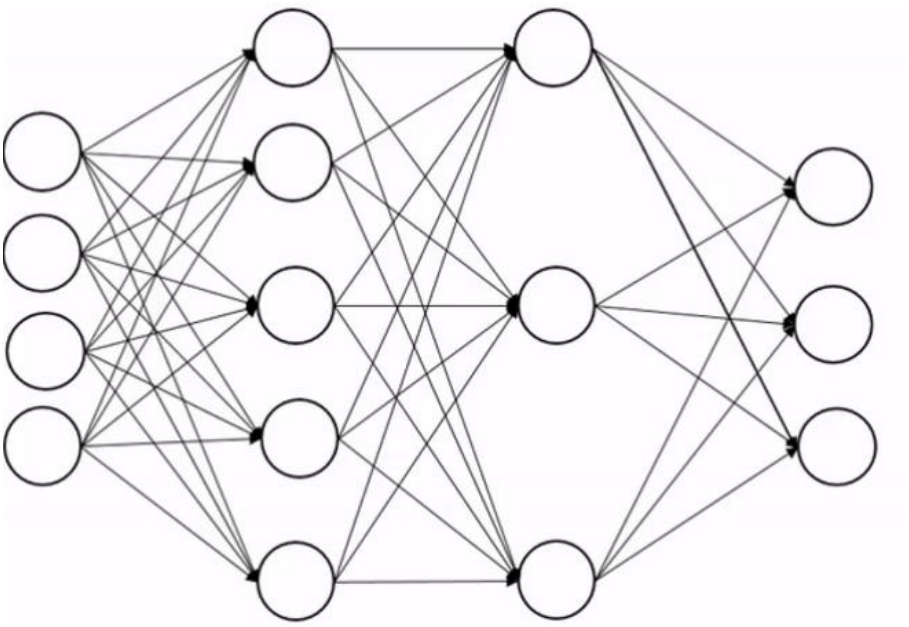
- **Dropout regularization**

When you have training data, if you try to train your model too much, it might overfit, and when you get the actual test data for making predictions, it will not probably perform well. **Dropout regularization** is one technique used to tackle overfitting problems in deep learning.

In the below image, we are applying a dropout on the second hidden layer of a neuron network.

Dropout

In machine learning, “dropout” refers to the practice of disregarding certain nodes in a layer at random during training. A dropout is a regularization approach that prevents overfitting by ensuring that no units are codependent with one another.



Training with Drop-Out Layers

Dropout is a regularization method approximating concurrent training of many neural networks with various designs. During training, some layer outputs are ignored or dropped at random. This makes the layer appear and is regarded as having a different number of nodes and connectedness to the preceding layer. In practice, each layer update during training is carried out with a different perspective of the specified layer. Dropout makes the training process noisy, requiring nodes within a layer to take on more or less responsible for the inputs on a probabilistic basis.