COMP3608 Notes

Lecture 1 - What is AI, History and State of the Art

What is AI?

- There are four views about how a system with AI should work:
  - Human-centred vs Rationality-centred approaches.
  - Thinking vs Acting.

Think Like Humans	Think Rationally
Act Like Humans	Act Rationally

- Rationality: ideal performance given a performance measure. Acting rationally means to select the best action at each step i.e. the one that maximises the given performance measure.

Acting Rationally: The Rational Agent Approach

- AI is a part of computer science which is concerned with the design of intelligent agent programs. An intelligent agent:
  - Has in-built knowledge and goals.
  - Perceives the environment.
  - Acts rationally to achieve its goals by using its knowledge and the percept sequence (history of all perceptual inputs that an agent has perceived to date) from the environment.
  - Acts rationally by doing the right thing - the thing that maximises a performance measure.
- In complex environments:
  - Always doing the right thing is not possible since computational demands (time, memory) are too high.
  - Limited rationality: acting appropriately when there is not enough time to make all computations for a perfect solution.

Acting Humanly - The Turing Test Approach

- Proposed by Alan Turing in 1950, he asked whether machines could think and how we could tell. Since thinking is difficult to define, we replace this question by stating that a machine is intelligent if it can pass a test of behaviour.
  - A human asks written responses and receives back written responses.
  - A computer passes the test if the human cannot tell if the responses come from a human or computer.
- ALICE chat bot developed by Richard Wallace won the Loebner Prize for best program on a restricted Turing test three times.

Irena: Thanks for talking with me.
ALICE: Don't mention it.
Irena: Ciao!
ALICE: Does CIAO mean hello or goodbye?
Irena: Both.
ALICE: What are both like?
Irena: Bye! It was nice talking to you.
ALICE: Goodbye. Thanks. It was nice chatting with you as well.

- **What capabilities does the computer need for the Turing test?**
  - Natural language processing - to communicate successfully in written English.
  - Knowledge representation - to store what it knows and hears.
  - Automated reasoning - to use the stored information to answer questions and draw conclusions.
  - Machine learning - to adapt to new situations, to detect patterns and apply them to new situations.
- The Total Turing test incorporates two more capabilities, testing the subject's perceptual abilities by showing video and passing physical objects.
  - Computer vision - to perceive objects.
  - Robotics - to move them around.
- Reverse Turing tests require humans to prove themselves to computers e.g. CAPTCHA tests.

Passing the Turing Test

- Within AI, little effort is used to pass the Turing test, since testing whether a machine resembles a human being (or simulated intelligence) is not very useful.
- It is more important to understand the principles of intelligent behaviour, and use them to build intelligent systems.
  - E.g. The goal of aeronautical engineering is not to make flying machine fly exactly like other pigeons, fooling them - it is to learn aerodynamic principles and apply them to a new machine.

Acting Humanly - ELIZA

- Joseph Weizenbaum wrote the Communications of the ACM (1966). It was one of the first chatbots, programmed to behave like a psychiatrist.
- It could communicate in natural language on any topic.

Visitor: The trouble is, my mother's ill.
ELIZA: How long has she been ill?
…

Visitor: The trouble is, my mother's Irish.
ELIZA: How long has she been Irish?
…

Visitor: The trouble is, my mother's poodle.
ELIZA: How long has she been poodle?

- ELIZA had no comprehension of language or context - it looked for key words and manipulated a set of

questions to produce a response.

Thinking Humanly: the Cognitive Modelling Approach

- Once we have a theory on how humans think, we can write a computer program that thinks like a human. Given a problem, it is not enough that it is solved correctly by the computer - the same reasoning steps as in humans should be followed.
- General Problem Solver (GPS) by Newell and Simon 1957: program for proving theorems and solving geometric problems using an order similar to humans in considering goals and sub-goals.
- Cognitive science: a separate field from AI, it constructs and tests theories of how the human mind works using methods from psychology and computer models.

Thinking Rationally: The Laws of Thought Approach

- Goal: use logic to build intelligent systems.
    - Take a description of a problem in logical notation. Find a solution (if one exists) using correct inference.
- Problems:
    - Taking informal knowledge and representing it in formal notation is difficult. This is especially difficult when knowledge is uncertain.
    - Limited time and memory - which reasoning step to apply first?

History of AI

Foundations of AI

- Philosophy (428BC): theories of reasoning, learning, rationality and logic.
- Mathematics (800): algorithms, formal representations and proof, computation, (un)decidability, (in)tractability, probability.
- Psychology (1879): adaptation, phenomena of perception and motor control, experimental techniques to investigate the human mind.
- Computer Engineering (1940): machines to make AI reality
- Linguistics (1957): structure and meaning of language, grammar, knowledge and representation.

Gestation (1943-1955)

- **1943:** McCulloch and Pitt's model of the artificial neuron (first work recognised as AI)
    - Neuron is on and off, on when there is sufficient stimulation from neighbouring neurone, off otherwise.
    - Mathematically: weighted sum of input signals is compared to a threshold to determine the outputs.
    - Showed that a network of connected neurons can compute any arithmetic function and that all logical connections (AND, OR, NOT) can be implemented by simple networks of connected neurons.
    - Parameters of the network had to be designed, but it was suggested that they could be learned from examples.
- **1950:** Hebb proposed a rule for modifying the connection strength between neurons - Hebbian learning; still used in neural networks.

- **1950:** Turing's article stated a vision for AI (Turing test, ML, genetic algorithms and reinforcement learning).
- 1951: Minsky and Edmonds built the first neural network computer (Snarc) - a neural network of 40 neurons using 3000 vacuum tubes.

Birth (1956)

- **1956:** John McCarthy's workshop; 'AI' adopted, birth of AI.
    - Organised a 2 month workshop at Dartmouth College for researchers interested in automata theory, neural nets and the study of intelligence.
    - No breakthroughs were made but ten main AI people were introduced to each other. Herbert Simon and Allen Newell presented the Logic Theorist, a computer program for proving theorems - introduced reasoning as search in a tree and use of heuristics to trim unpromising branches.

Early Enthusiasm (1950s - 60s)

- **1957:** Newell and Simon's thinking humanly General Problem Solver.
- 1958: McCarthy wrote Lisp.
- 1959 - 1975: Samuel wrote programs for playing checkers.
    - Invented alpha-beta pruning for pruning unnecessary branches of the game tree.
    - Disproved the idea that computers can do only what they are told to - the programs were able to learn from experience.
- **1960:** Rosenblatt, Widrow and Hoff proposed perceptrons and linear neural networks - learning rule to train neural networks to solve pattern recognition problems.
- 1965: Robinson proposed the resolution theory - theorem proving for 1st order logic that was the basis for Prolog.
- 1967 - 1968: STUDEN and ANALOGY program were made to solve microworld problems (algebra and geometric analogy problems respectively).
- **High Expectations**: great progress was made, yet early AI systems failed when they were tried on a wider selection of problems and on more difficult problems.

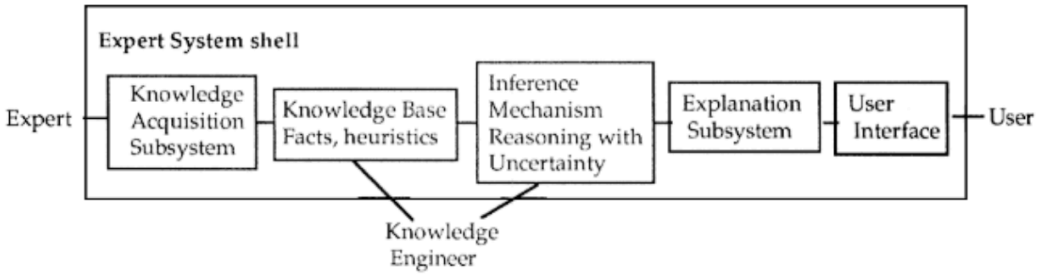
A Dose of Reality (1960s - 70s)

- Problem 1: Limitations of algorithms
- 1960: Minsky and Papert, book "Perceptrons" demonstrated limitations of existing neural networks and the need for more complex neural networks and a new learning rule.
- Rosenblatt and Widrow were aware of these limitations but were not able to modify their learning algorithm to train more complex net, despite proposing new neural networks.
- Problem 2: Need for deeper knowledge
- Syntactic manipulation was not sufficient, and the correct word depended on the context around it.
    - English "The spirit is willing but the flesh is weak" $\rightarrow$ Russian "The vodka is good but the meat is rotten".
- Problem 3: Many real problems are intractable
    - Early AI programs solved problems by using different combinations of steps until the solution was

found, which worked well for simple domains with small number of objects and possible actions.
    - Generalisation to larger problems proved difficult (and not only a matter of faster hardware and bigger memory).
    - Need to deal with combinatorial explosion and restrictions behind computing power.

The Come Back (from 1970s)

- Knowledge-based systems to solve specific problems in restricted domains.
- Expert systems industry provide expert quality advice, diagnoses and recommendation for real-world problems; rule based.
    - Should be able to provide explanation for its advice (a WHY explanation).



- 1980s: Rumelhart and McClelland brought neural networks back to popularity - backpropagation algorithm for training multilayer perceptrons was the answer to *Problem 1* Minsky and Papert.

Today

- Machine learning and data mining are the fastest growing areas of AI - Big Data (can be Big Garbage), Deep Learning.
- A lot of unique data is being collected - trillions of words, billions of images, billions of base pairs on genomic sequences, weather data etc.
    - Can be used to extract useful patterns used for different types of analytics.
    - Train machines to learn to perform different tasks e.g. translate languages or drive cars.

State of the Art

- Playing games: games test our intelligence with simple rules and a goal to win - has been used as mice labs for AI, where cutting-edge AI concepts have been produced by highly specialised tasks. Results:
    - 1997: Deep Blue (IBM) defeated world chess champion Gary Kasparov 2W3D1L.
    - 2016: AlphaGo (Google) defeated second in the world Go player Lee Sedol 4W1L using a CNN trained using data from previous games played by human experts.
- Maps and navigation: uses AI methods to plan the best route, evaluate traffic, realign itself when we take the wrong path.
- Google Translate: deep learning trained using a large corpora of texts for written, spoken and image text.
- Question answering: IBM's DeepQA project Watson won Jeopardy!, which answered questions posed in natural language.

- Brain-Computer Interaces (BCI): BCI is a system which allows a person to control a computer application by only using his or her thoughts - goal is to give paralysed people another way of communicating, a way which doesn't depend on muscle control but on their thoughts.
- Other examples: driving motor vehicles, recommender systems (books, TV shows, online dating).

Future Trends

- Transportation - smarter cars, remote controlled delivery vehicles, on-demand transportation (Uber) with AI algorithms matching drivers to passengers by location and reputation.
- Home robots - clean offices, deliver packages, interact with people at home.
- Healthcare - clinical decision supports, healthcare analytics for personalised diagnosis and treatment, patient risk prediction, mobile healthcare etc.
- Elder care - better hearing, visual aids, walkers, wheelchairs, in-home monitoring.
- Education - enhance education with personalisation, auto-grading assignments, automatic feedback to students, students at risk of failing.
- Public safety and security - white collar crime, spam, crime prevention and prediction, suspicious social data.
- Implications on the workforce - profound change but roles for humans will always remain.

Lecture 2 - Uninformed and Informed Search

Problem Solving and Search

- Many tasks can be formulated as search problems.

Task	**To get from an initial problem state to a goal state**
Initial State	Arad
Goal State (1 or more)	Stated explicitly e.g. Bucharest, or stated explicitly with a goal test e.g. checkmate.
Operators	Set of possible actions transforming one state to another e.g. driving between cities.
Path Cost Function	Assigns a numeric value to each path, reflecting the performance measure e.g. distance between cities.
Solution	A path from the initial to a goal state - quality measured by the path cost, with the optimal solution being the one with the lowest path cost.
State Space	All states reachable from the initial state by operators.

Choosing States and Actions with Abstraction

- Real problems are too complex. To solve them, we need to abstract them i.e. to simplify them by removing unnecessary details e.g. scenery.
    - Actions need to be suitably specified (the level of abstraction must be appropriate) e.g. turn the steering wheel 5 degrees to the left is not appropriate.
        - Abstract state = set of real states, abstract action = complex combination of real actions,

abstract solution = real path that is a solution in the real world.

The 8-Queens

- Place 8 queens on an 8 x 8 chessboard so that no queen can attack each other.

Task	
Goal State	8 queens on the board, none attacked
Path Cost	0 (only goal state matters)
States	1 board configuration of tiles
Operators	Put 1 queen on the board (or move 1)

- Different types of problem formulation:
  - Incremental - start with an empty board, add 1 queen at a time.
  - Complete-state - start with all 8 queens and move them around.

Incremental 1	
States	Any arrangement of 0 to 8 queens
Initial State	No queens on the board
Operators	Add a queen to any square
State Space	$64 \times 63 \times \ldots \times 57 = 1.8 \times 10^{14}$

- Huge state space cannot be reasonably solved. A better formulation prohibits placing a queen in any square that is already attacked.

Incremental 2	
States	Any arrangement of 0 to 8 queens, 1 in each column
Initial State	No queens on the board
Operators	Place a queen in the left-most empty column such that it is not attacked by any other queen
State Space	2057

- With 100 queens, formulation 1 has 10^{400} states, while formuation 2 has a hugely improved by not tractable 10^{52} states.

Searching for Solutions

- Solving the 8-queens puzzle or Romania example can be done by:
  - Searching the state space.
  - **Generating a search tree starting from the initial state and applying operators.**
  - Generating a search graph - the same state can be reached from multiple paths.

Tree-Search

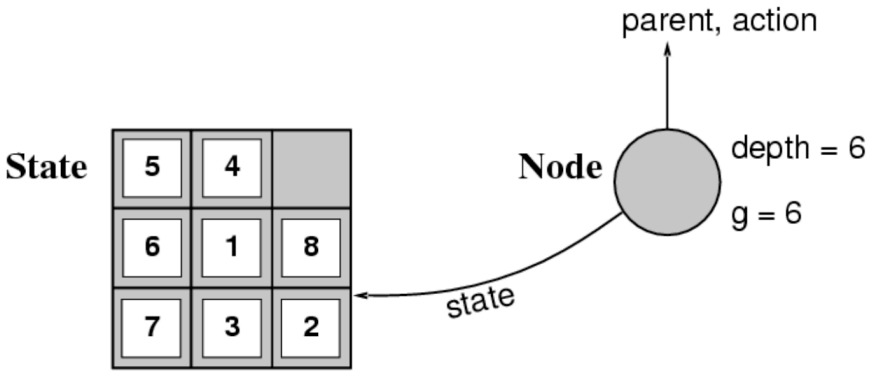
- Basic idea is offline exploration of the state space by generating successors of the explored states through expansion.
  - Choose a node for expansion based on a search strategy.
  - Check if it is a goal node.
    - If yes, return the solution.
    - If no, expand it, move it to the expanded list, generate its children, and put them in the fringe in the order defined by the search strategy.

Expanded and Fringe Lists

- We will keep two lists:
  - Expanded: nodes that have been expanded (children added to fringe, node explored).
  - Fringe: for nodes that have been generated but not yet expanded. We will keep the fringe ordered and always select the first node of the fringe for expansion. It is implemented using a priority queue (e.g. heap).

Nodes vs States

- Node: represents a state, is a data structure used in the search tree, and includes the parent, children and any other relevant information such as depth and path cost g.



Search Strategies

- A search strategy defines which node from the fringe is most promising and should be expanded next.
- Evaluation Criteria:
  - Completeness: is it guaranteed to find a solution if one exists?
  - Optimality: is it guaranteed to find an optimal (least cost path) solution?
  - Time complexity: how long does it take to find the solution? (measured as number of generated

nodes).
    - Space complexity: what is the maximum number of nodes in memory?
- Time and space complexity are measured in terms of:
    - b: max branching factor of the search tree (assumed as finite)
    - d: depth of the optimal (least cost) solution
    - m: maximum depth of the state space (can be infinite)

Uninformed (Blind) Search Strategies

- Do not use problem-specific heuristic knowledge.
- Generate children in a systematic way e.g. level by level, left to right.
- Know if a child node is a goal or non-goal node BUT do not know if one non-goal child is better/more promising than another. Exception of UCS, but this is not based on heuristic knowledge.
- In contrast, informed searches use heuristic knowledge to determine the most promising non-goal child.

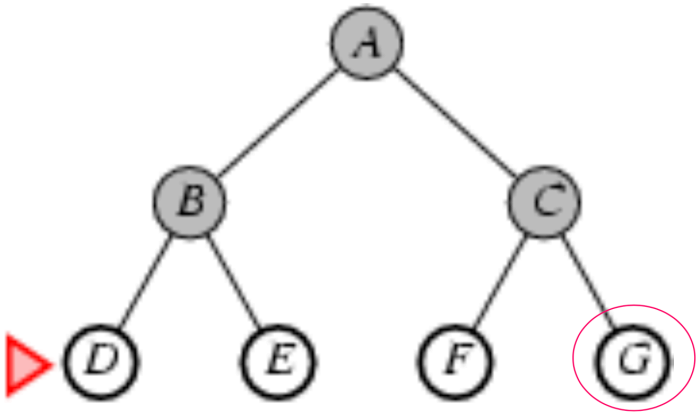
UCS and BFS

- UCS and BFS are equivalent when the step cost is 1, the same, $g(n) = \text{depth(n)}$, $g(n)$ is the same at each level and increasing as the level increases e.g. 5 for level 1, 7 for level 2.

Breadth-First Search (BFS)

Fringe: D, E, F, G

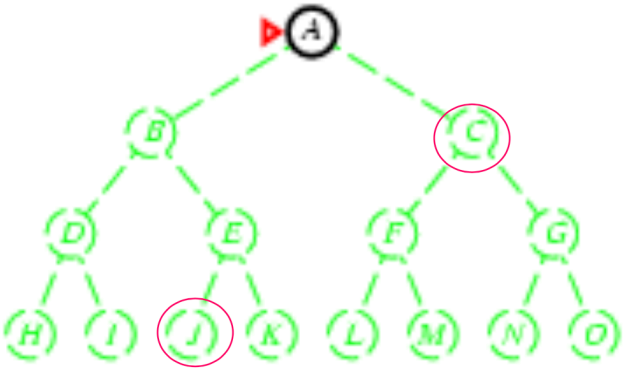
Expanded: A, B, C



- Expands the shallowest unexpanded node.
- Implementation: insert children at the end of the fringe.
    - Is the first node in the fringe a goal node?
        - Yes: stop and return the solution
        - No: expand it, move it to the expanded list. Generate its children and put them in at the end of the fringe.
- There may be loops (repeated states), and thus a repeated state checking mechanism may be needed.

Properties of BFS

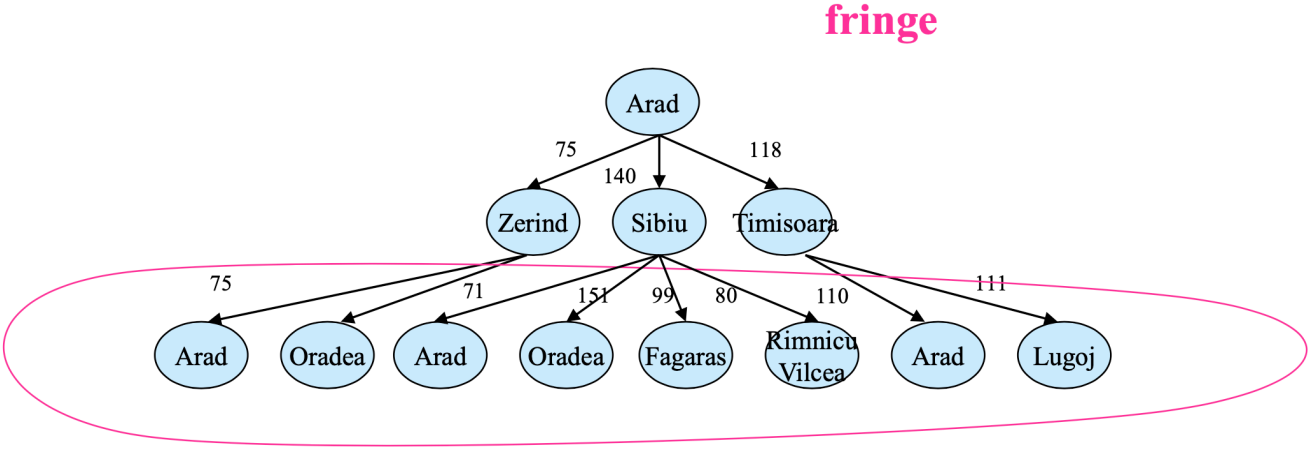
- **Complete**: Yes (since we assume branching factor b is finite).
- **Optimal**: If all the step costs are the same/not the same, BFS is/isn't optimal (it doesn't consider the step costs regardless).
    - Suppose C and J are the goal nodes, but J is a better solution. C is chosen by BFS since it is the shallowest goal node.



- **Time**: Generated notes = $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, exponential.
- **Space**: $O(b^d)$ since it keeps every node in memory.
- Both time and space are problems as they grow exponentially with depth, but space is the bigger problem.
    - Search problems with exponential complexity can NOT be solved by uninformed search except for small depth and branching factor.

Uniform Cost Search (UCS)

Fringe: (Oradea,146), (Arad, 150), (Riminicu Vilcea, 220), (Arad,228), (Lugoj, 229), (Fagaras, 239), (Oradea , 291)
Expanded: Arad, (Zerind, 75), (Timisoara, 118), (Sibiu, 140)



- UCS considers the step cost. It expands the least-cost unexpanded node i.e. the node n with the lowest path cost $g(n)$ from the intial state.
- Implementation: insert nodes int he fringe in order of increasing path cost from the root.

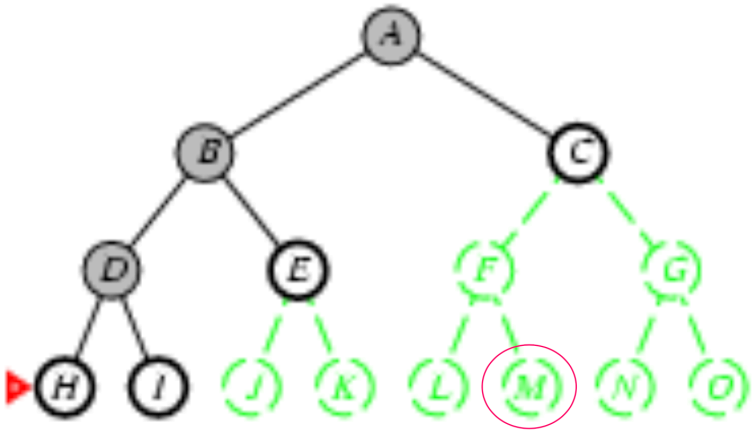
- Is the first node in the fringe a goal node?
  - Yes: stop and return the solution
  - No: expand it, move it to the expanded list. Generate its children and put them in the fringe in based on the path cost from the root.

Properties of UCS

- **Complete**: Yes (if the step costs are greater than 0).
- **Optimal**: Yes
- **Time**: Depends on path costs not depths. $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, typically $O(b^d)$ since UCS may explore long paths of small steps before exploring path with large steps (which may be more useful).
- **Space**: Depends on path costs not depths. $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ where C^* is the cost of the optimal solution, and ϵ is the smallest step cost.

Depth-First Search (DFS)

Fringe: H, I, E, C
Expanded: A, B, D



- Expands the deepest unexpanded node.
- Implementation: insert children at the front of the fringe.
  - Is the first node in the fringe a goal node?
    - Yes: stop and return the solution
    - No: expand it, move it to the expanded list. Generate its children and put them in at the front of the fringe.
- DFS can perform infinite cycle explorations $\rightarrow$ needs a finite, non-cyclic search space or a repeated state checking mechanism.

Properties of DFS

- **Complete**: No, since it fails in infinite-depth spaces where $m = \infty$, since it can get stuck going down infinite paths e.g. unbounded left sub-tree with no goal node on sub-tree. Yes, in finite spaces.
- **Optimal**: No, since it may find a solution longer than the optimal e.g. goal node found first on left subtree with cost 5, despite a goal node on the right subtree with cost 1.
- **Time**: $1 + b + b^2 + b^3 + \ldots + b^m = O(b^m)$, similar to BFS although higher since $m > d$.
- **Space**: $O(bm)$, linear - excellent. This is because once a node has been expanded, it can be removed

from memory as soon as all its descendents have been fully explored (tree is trimmed).

Iterative Deepening Search (IDS)

Depth-Limited Search

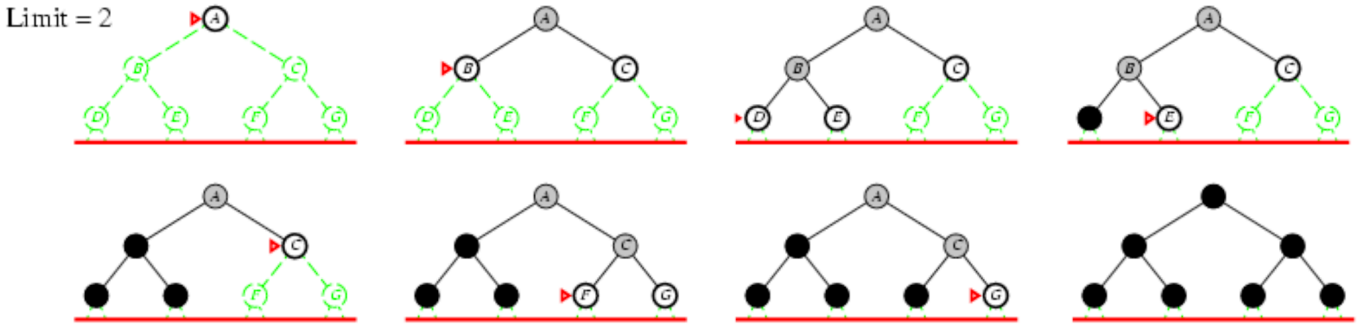
- DFS with depth limit l - imposed cutoff on the maximum depth.
  - Selecting a good limit l is based on domain knowledge e.g. 20 Romanian cities, l=19 is ok. We can see that any city can be reached by any other city in at most 9 steps, so $l = 9$ is best (diameter of state space). The diameter of the state space is not known in advance for most problems however.
- **Complete**: Yes, since the search depth is always finite.
- **Optimal**: No.
- **Time**: $1 + b + b^2 + b^3 + \ldots + b^l = O(b^l)$.
- **Space**: $O(bl)$

Back to IDS

- Sidesteps the issue of choosing the best depth limit by trying all possible depth limits in turn and applying DFS.
- For depth $= 0$ to ∞, do depth-limited search(depth):

```
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
```

- Tries to combine the benefits of DFS (low memory) and BFS (completeness if b is finite, and optimality if step costs are the same) by repeated DFS searches with increased depth.
- The nodes close to the root will be expanded multiple times but the number of these nodes is small - the majority of the nodes are close to the leaves, not the root (small overhead).
- Implementation: insert children at the front of the fringe.
  - Is the goal found in this level?
    - Yes: Stop and return the solution
    - No: Increase l and repeat depth-limited search.



Expanded: A, B, D, E, C, F, G

Properties of IDS

- **Complete**: Yes, as BFS. This is as opposed to DFS which is yes, if and only if m is finite.
- **Optimal**: No, in general. Yes, if the step cost = 1, as BFS. This is as opposed to DFS which is no, even if step cost = 1.
- **Time**: $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$ as BFS, as opposed to DFS $O(b^m)$. Each term represents the number of times each node is explored (e.g. b^0, the root, is explored d+1 times).
- **Space**: $O(bd)$, linear as DFS.
- Can also be modified to explore uniform-cost trees.

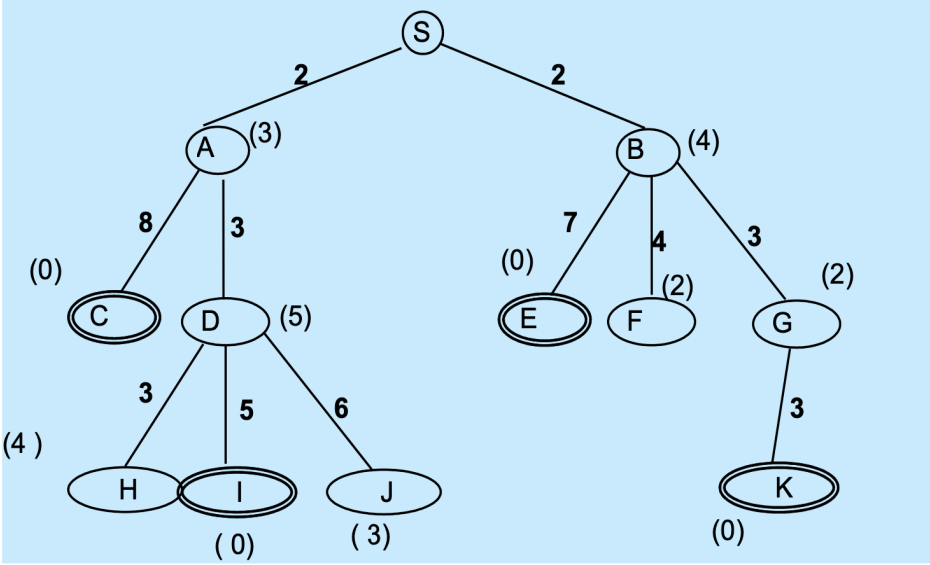
Real-World Path Finding Problems

- Planning trips from A to B - Google Maps, public transport trips.
  - Initial and goal states: specified by user.
  - Action: take any public transport method, leaving after current time with enough transfer time.
  - Path cost: monetary cost, transport time, waiting time etc.
- VLSI layout - for microchip assembly to minimise area and circuit delays.
- Assembling objects by a robot.
- Protein design - find a sequnce of amino acids that will fold into a 3-dimensional protein with useful properties.

Informed (Heuristic) Search Strategies (Best-First Search)

- Informed search strategies use problem-specific heuristic knowledge to select the order of node expansion. They:
  - can compare non-goal nodes - they know if one non-goal node is better than another.
  - are typically more efficient.
- Domain specific knowledge is used to devise an evaluation function which estimates how good each node is, assigning a value to each node.
  - Called best-first search, we expand the one with the best value (although we don't really know which is the best node since evaluation functions are educated estimations - best-first searches expand the node that appears best).
- Fringe: insert children in decreasing order of desirability

Greedy Search (GS)



Fringe: (C,0), (B,4), (D, 5) **//C and D are added and the fringe is ordered**

Expanded: S, (A,3)

- Greedy searches use h values as an evaluation function (as in $h$euristic).
  - $h(n)$ for a node n is the estimated cost from n to a goal node.
  - GS expands the node with the smallest h i.e. the node that appears closest to the goal. Generated children are added to the fringe based on desirability, as a tuple with their h value.
    - In the Romania example, we can use $h(n) = SLD(n, Bucharest)$ i.e. the straight-line distance from a node to Bucharest.
- If a search algorithm uses $h(n) = g(n)$, this greedy search is equivalent to UCS.

Properties

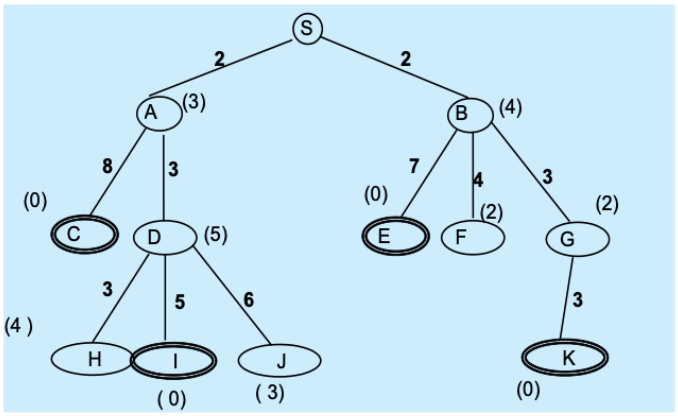
- **Complete**: As DFS, yes in finite space (m is finite), but no in finite spaces (and can get stuck in loops).
- **Optimal**: No. In the above case, SAC is found first, despite SBE and SBGK being better.
- **Time**: $O(b^m)$, but a good heuristic can give a dramatic improvement.
- **Space**: $O(b^m)$, since it keeps every node in memory.

Lecture 3 - A* and Local Search Algorithms

A* Search

- UCS minimises the cost so far $g(n)$.
- GS minimises the estimated cost to the goal $h(n)$.
- A* combines UCS and GS, using the evaluation function $f(n) = g(n) + h(n)$. $f(n)$ is thus the **estimated total cost of the path through n to the goal**.

  - **Fringe: (K, 8), (F, 8), (E, 9), (C, 10), (D, 10)**
  - **Expanded: S, (A, 5), (B, 6), (G, 7)**



Comparison to Other Algorithms

- UCS is a special case of A* when $h(n)$ is equal to 0.
- BFS is a special case of A* when $f(n) = \text{depth(n)}$, and when among nodes with the same priority, the left most is expanded first.
- BFS is also a special case of UCS when $g(n) = \text{depth(n)}$.

Properties of A*

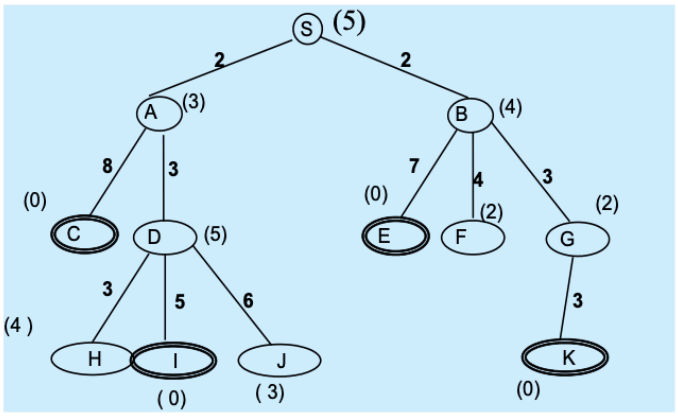
- **Complete**: Yes. This is unless there are infinitely many nodes with $f \leq f(G)$.
- **Optimal**: Yes, with an admissible heuristic.
- **Time**: $O(b^d)$, exponential.
- **Space**: Exponential, keeps all nodes in memory.
- For most problems, the number of nodes which have to be expanded is exponential. Both time and space are problems for A* but space is a bigger problem.
- We can solve this using Iterative Deepening A* (IDA*), Simplified Memory-Bounded A* (SMA*), simply using a non-admissible heuristic that works well in practice or a loacl search algorithm $\rightarrow$ completeness and optimality are no longer guaranteed.
- Although dominant heuristics are better, they may need a lot of time to compute - may be better to expand more nodes using a faster, simpler heuristic.

Admissable Heuristic

- A heuristic $h(n)$ is admissible if for every node n:
  - $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost to reach a goal from n i.e. the estimate to reach a goal is smaller than or equal to the true cost to reach a goal.
- Admissible heuristics are optimistic - they think that the cost of solving the problem is less than it actually is e.g. the straight line distance heuristic never overestimates the actual road distance.
- **Theorem**: If h is an admissable heuristic, then A* is complete and optimal.

Admissable Heuristic Example

- No need to check goal nodes ($h = 0$ for them) and nodes that are not on a goal path.

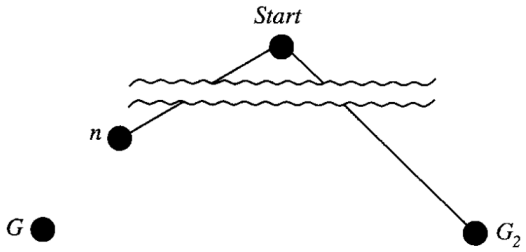


- $h(S) = 5 \leq 8, h(B) = 4 \leq 6, h(G) = 2 \leq 3$ etc. It is admissable.

Optimality of A* with an Admissible Heuristic Proof

- **Optimal solution**: the lowest cost path to a goal node.
- **Idea**: Suppose that some sub-optimal goal G_2 has been generated and it is in the fringe. We will show that G_2 can not be selected from the fringe.
- **Given**: G - optimal goal, G_2 - sub-optimal goal, h is admissable.
- **Prove**: G_2 can not be selected from the fringe for expansion.

Let n be an unexpanded node in the fringe such that n is on the optimal (shortest) path to G (there must be such a node). We will show that $f(n) < f(G_2)$ i.e. n will be expanded, not G_2.



$$f(G_2) = g(G_2) + h(G_2) = g(G_2) \text{ since } h(G_2) = 0 \ (1)$$
$$f(G) = g(G) + h(G) = g(G) \text{ since } h(G_2) = 0 \ (2)$$
$$g(G_2) > g(G) \text{ since } G_2 \text{ is suboptimal } (3)$$
$$f(G_2) > f(G) \ [(1) \text{ and } (2) \text{ into } (3)] \ (4)$$

Comparing $f(n)$ and f(G):

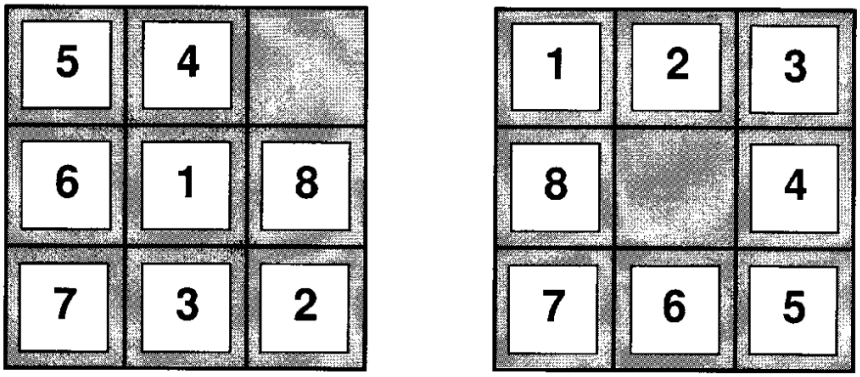
$$f(n) = g(n) + h(n) \ (5)$$
$$h(n) \leq h*(n) \text{ where h*(n) is the true cost from n to G, h is admissible } (6)$$
$$f(n) \leq g(n) + h^*(n) \ [(5) \text{ and } (6)] \ (7)$$
$$g(n) + h^*(n) = g(G) = f(G) \text{ since its the path cost from S to G via n } (8)$$
$$f(n) \leq f(G) \ [(6) \text{ and } (7) \text{ and } (8)] \ (9)$$

Therefore:

$$f(n) \leq f(G) \leq f(G_2) \ (9) \text{ and } (4)$$

- Since $f(n)$ is less than or equal to $f(G_2)$, n will be expanded, not G_2.

Heuristics for the 8-Puzzle



Start State **Goal State**

- $h_1(n)$ = number of misplaces tiles
- $h_1(Start) = 7$ since 7 of 8 tiles are out of position.
- h_1 is thus admissible becuase admissible heuristics are optimistic - they never overestimate the number of steps to the goal.
- h_1 essentially states that any tile out of place can be moved once to its spot.
- The true cost is higher, since any tile out of place must be moved at least once.

A Different Heuristic

- $h_2(n)$ = the sum of the distances of the tiles from their goal positions (Manhattan distance), with only horizontal and vertical movement.
- $h_2(Start) = 18(2 + 3 + 3 + 2 + 4 + 2 + 0 + 2)$.
- h_2 is admissible because with Manhattan distance, with each step you move a tile to an adjacent position so that it is one step closer to its goal position, reaching the solution in h_2 steps.
- The true cost is higher, since moving to an adjacent position is not always possible (depends on the position of the blank tile).

How to Invent Admissible Heuristics?

- By formulating a **relaxed version** of the problem (problem with fewer restrictions on the actions) and finding the exact solution. This solution is an admissable heuristic.
- Theorem: The optimal solution to a relaxed proble is an admissible heuristic for the original problem.
    - The optimal solution to the original problem is also a solution to the relaxed version (by definition) $\rightarrow$ it must be at least as expensive as the optimal solution to the relaxed version$\rightarrow$ the solution to the relaxed version is less or equally expensive than the solution to the original problem $\rightarrow$i t is an admissible heuristic for the original problem
- Relaxed problems can be constructed automatically if the problem definition is written in a formal language.
    - A tile can move from square A to square B if A is adjacent to B and B is blank. 3 relaxed problems can be generated by removing conditions:
        - A tile can move from square A to square B if A is adjacent to B.
        - A tile can move from square A to square B if B is blank.
        - A tile can move from square A to square B.

Learning Heuristics from Experience

- Often we can't find a single heuristic that is clearly the best i.e. dominant. We have a set of heuristics h_1 , h_2,...,hm, but none of them dominates any of the others. Which should we choose?
- Solution: define a composite heuristic $h(n) = \max\{h_1(n), h_2(n), \ldots, hm(n)\}$. At a given node, it uses whichever heuristic is dominant. $h(n)$ is admissible because the individual heuristics are admissible.

8-Puzzle Classifier

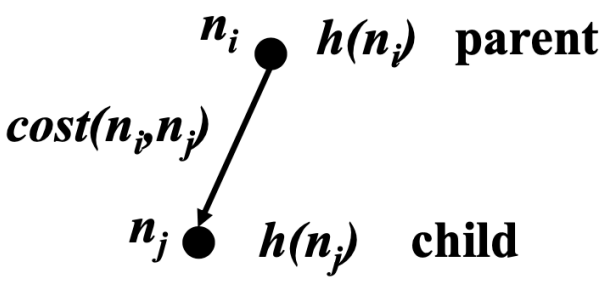
- Experience: many 8-puzzle solutions (paths from A to B). Each previous solution provides a set of examples to learn h.
- Each example is a pair (state, associated h). Since h is known for each state, we have a labelled dataset.
- The state is suitably represented as a set of useful features e.g. f_1 = number of misplaced tiles, f_2 = number of adjacent tiles that should not be adjacent. h is a function of the features, but we don't know exactly how it exactly depends on them (it learns this relationship from the data).
- We can generate e.g. 100 random 8-puzzle configurations and record the values of f_1, f_2, h to form a trainng set, and from that, a classifier.
- We use this classifier on new data i.e. given f_1 and f_2, predict h. No guarantee that the learned heuristic is admissible or consistent.

Ex.#	f1	f2	h
Ex1	7	8	14
…			
Ex100	5	2	5

Dominance

- **Given two admissible heuristics h_1 and h_2, h_2 dominates h_1 if for all nodes n, $h_2(n) \geq h_1(n)$.** Dominant heuristics give a better estimate of the true cost to a goal G.
- Theorem:
  - All n with $f(n) < f^*$ will be expanded (f^* is the cost of optimal solution path).
  - Therefore, all n with $h(n) < f^* - g(n)$ will be expanded. But since $h_2(n) \geq h_(n)$:
  - All n expanded by A* using h_2 will also be expanded by h_1, and h_1 may also expand other nodes.
- Typical search costs for 8-puzzle with $d = 14$: IDS = 3 473 941 nodes, A* (h_1) = 539 nodes, A* $(h_2) = 113 nodes.

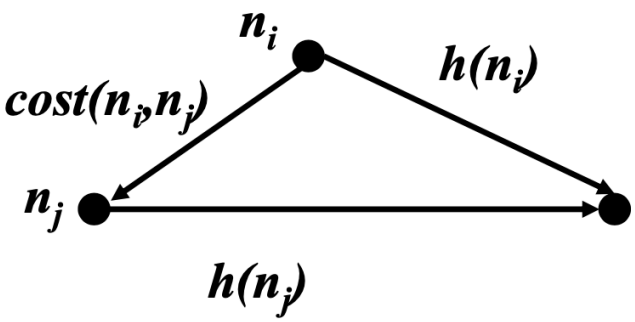
Consistent (Monotonic) Heuristic

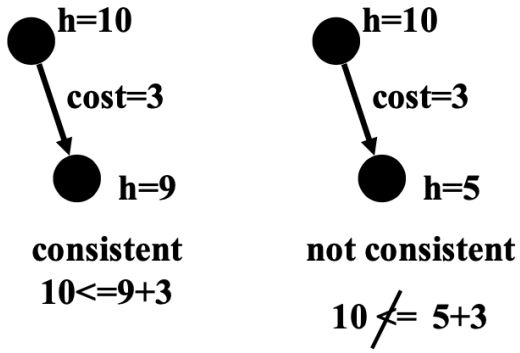


- Consider a pair of nodes n_i and n_j, where n_i is the parent of n_j.

- h is a consistent (monotonic) heuristic if for all such pairs in the search graph, the following triangle inequality is satisfied:

$$h(n_i) \leq \text{cost}(n_i, n_j) + h(n_j) \text{ for all n}$$



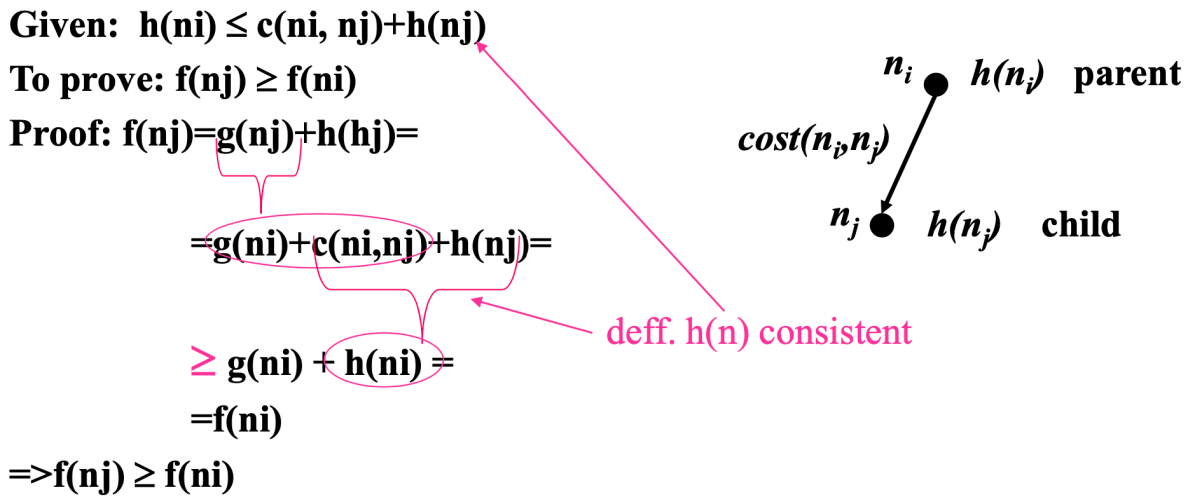


- Another interpretation of the triangle inequality is that along any path, our estimate of the remaining cost to the goal cannot decrease by more than the arc cost.

$$h(n_j) \geq h(n_i) - \text{cost}(n_i, n_j)$$

Consistency Theorems

- Theorem 1: If $h(n)$ is consistent, then $f(n_j) \geq f(n_i)$ i.e. f is non-decreasing along any path.



- Theorem 2: If $f(n_j) > f(n_i)$ i.e. f is non-decreasing along any path, then $h(n)$ is consistent.

f-contours with Consistent Heuristic

- A* uses the f-cost to select nodes for expansion. If h is consistent, the f-costs are non-decreasing $\rightarrow$ we can draw f-contours in the state space.
- A* expands nodes in order of increasing f-value i.e. it gradually adds f-contours, and nodes inside a contour have f-cost less than or equal to the contour value.

Completeness of A* with Consistent Heuristic

- As we add f-contour bands of increasing f, we must eventually reach a band where $f = h(G) + g(G) = h(G)$.

Optimality of A* with Consistent Heuristic

- A* finds the optimal solution i.e. the one with the smallest path cost $g(n)$ among all solutions.
- The first solution must be the optimal one, as the subsequent contours will have a higher f-cost, and thus a higher g-cost since $h(n) = 0$ for all goal nodes.

A* with Consistent Heuristic is Optimally Efficient

- **Theorem**: If h is a consistent heuristic, then A* is optimally efficient among all optimal search algorithms using h. No other optimal algorithm using h is guaranteed to expand fewer nodes than A*.

Admissibility and Consistency

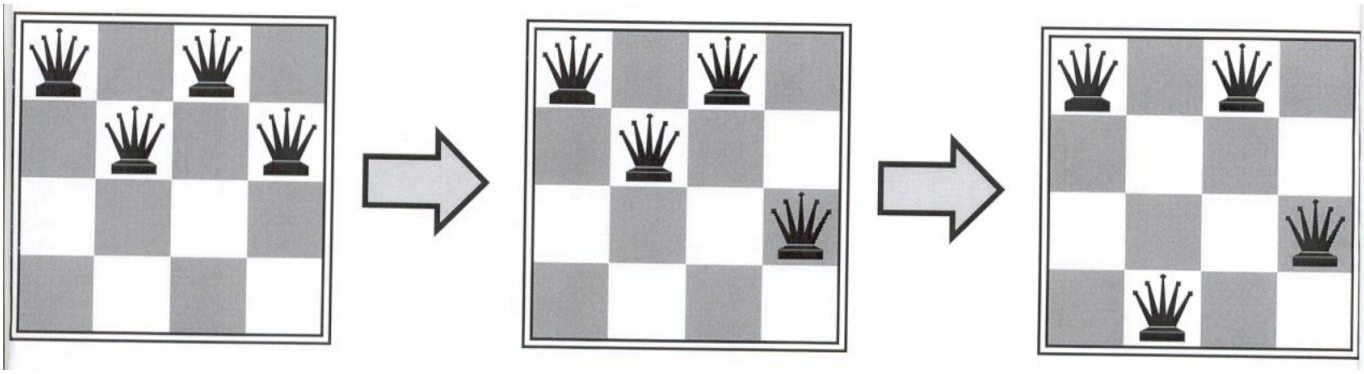
- Consistency is the stronger condition. Theorem: Consistent $\rightarrow$ Admissible.
  - If a heuristic is consistent, it is also admissible.
  - If a heuristic is admissible, there is no guarantee that it is consistent.

Optimisation Problems

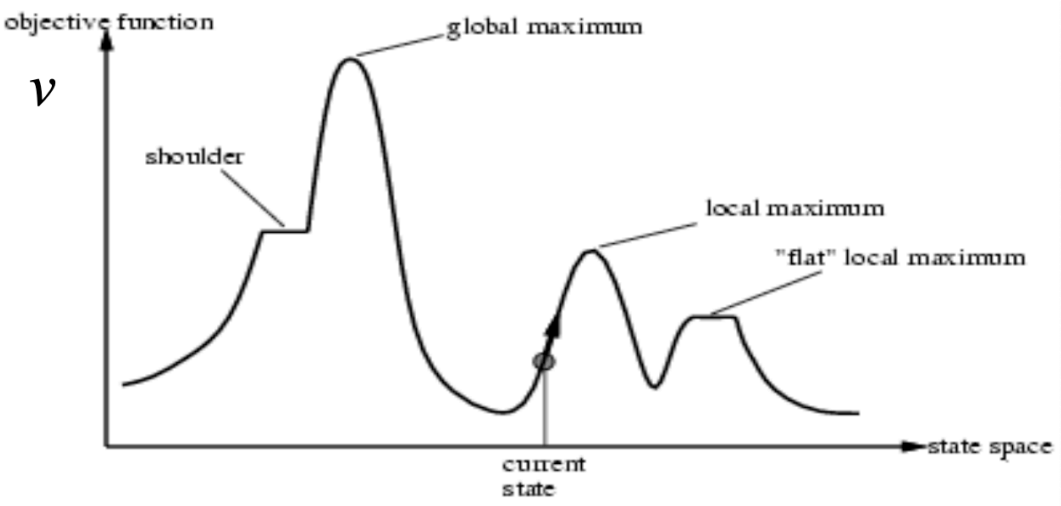
- Each state has a value v.
- **Goal**: Find the optimal state = the state with the highest or lowest v score (depending on what is desirable, max or min).
- **Solution**: the state, the path is not important.
- There are a large number of states such that it can't be enumerated. We can't apply the previous algorithms since they're too expensive.

Optimisation Problem Example

- n-queens problem: the solution is the goal configuration, not the path to it.
- Non-incremental formulation:
  - **Initial State**: n-queens on the board, given or randomly chosen.
  - **States**: any configuration with n-queens on the board.
  - **Goal**: no queen is attacking each other.
  - **Operators**: 'move a queen' or 'move a queen to reduce the number of attacks'.



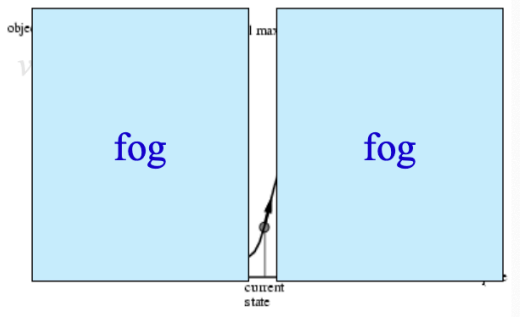
V-value Landscape



- Each state has the value v that we can compute. This value is defined by a heuristic evaluation function (also called objective function).
- Goal: 2 variations depending on the task:
  - Find the state with the highest value (global maximum).
  - Find the state with the lowest value (global minimum).
- **Complete** local search: finds the goal state if one exists.
- **Optimal** local search: find the best goal state, the state associated with the global maximum or minimum.

Hill-Climbing Algorithm

- Also called iterative improvement algorithm.
- **Idea**: Keep only a single state in memory, try to improve it.
- **Two variations**:
  - Steepest ascent - the goal is the maximum value
  - Steepest descent - the goal is the minimum value



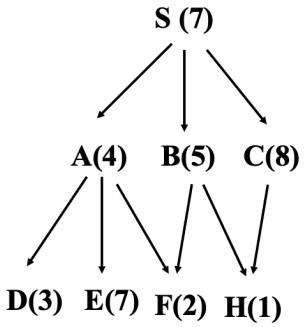
- Assuming hill-climbing ascent, the idea is to move around trying to find the highest peak.
    - Store only the current state.
    - Do not look ahead beyond the immediate neighbours of the current state (cannot see the whole landscape).
    - If a neighbouring state is better, move to it and continue, otherwise stop.
    - "Like climbing Everest in thick fog with amnesia".

Implementation

- Set current node n to the initial state s, which can be given or randomly selected.
- Generate the succesors of n. Select the best successor n_{best} i.e. the highest score for ascent.
- If $v_{\text{best}} < v(n)$, return n #compare the child with the adult, if the child is lower, stop, since a local or global max has been found.
- Else set n to n_{best} and go to step 2.

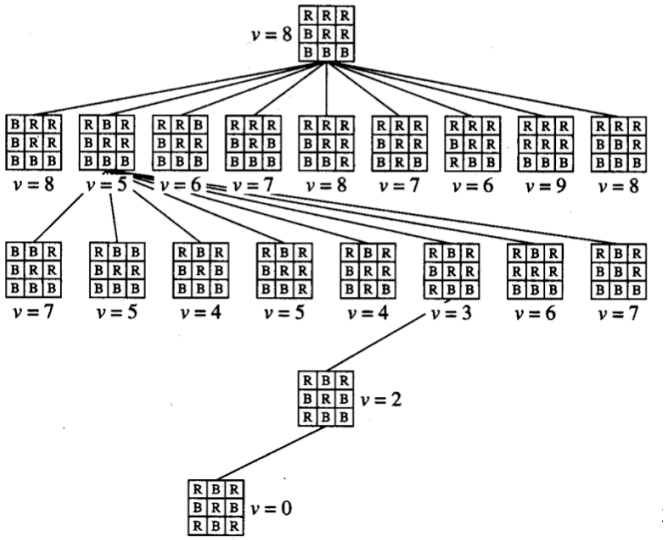
Example 1

- Lower the better, descent. Expanded nodes: SAF.



Example 2

- Given a 3x3 grid, each grid is coloured either red or blue.
- The aim is to find the colouring with minimum number of pairs of adjacent cells with the same colour .
- Descending search.

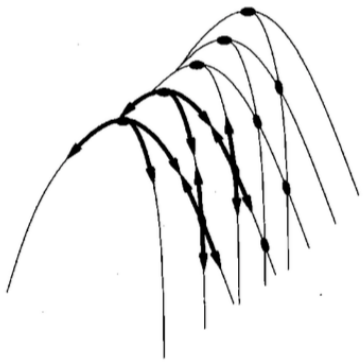


Hill-Climbing Search Pros and Cons

- Pros:
  - Good choice for hard, practical problems.
  - Uses very little memory (keeps just one node in memory).
  - Finds reasonable solutions in large spaces where systematic algorithms are not useful.
- Cons:
  - Not a very clever algorithm - can easily get stuck in a local optimum.
  - However, not all local maxima/minima are bad - some may be reasonably good even though not optimal.

Escaping Bad Local Optima

- The solution that is found depends on the initial state. When the solution found is not good enough, have a random restart:
  - Run the algorithm several times starting from different points, select the best solution found i.e. the best local optimum.
  - Applicable for tasks without a fixed initial state.
- Plateaus (flat areas): no/little change in v. Our version of hill-climbing does not allow visiting states with the same v as it terminates if the best child's v is the same as the parents.
  - Other version keep searching even if values ar ethe same, although it may to endless walking and node revisiting.
    - **Solution**: Keep track of the number of times v is the same and do not allow revisiting of nodes with the same v.



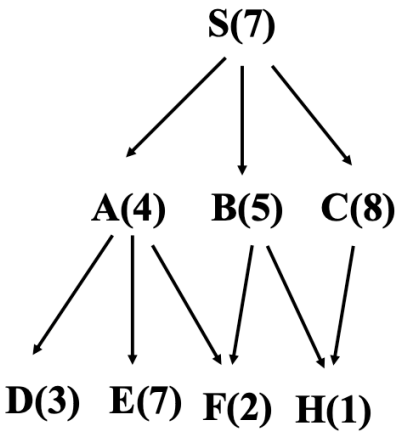
- Ridges: the current local maximum is not good enough; we would like to move up but all moves go down.
    - Example: dark circles are states. A sequnce of local maxima that are not connected with each other.
    - From each of them, all available actions point downhill.
        - **Solution**: combine 2 or more moves in a macro move that can increase the height or allow a limited number of look-ahead search.

Beam Search

- Keeps track of k states rather than just 1.
- **Version 1**:
    - Start with 1 given state.
    - At each level: generate all succesors o fthe given state.
    - If any one is a goal state, stop; else, select the k best succesors from the list and go to the next level.
- **Version 2**:
    - Starts with k randomly generated states.
    - At each level, generate all successors of all k states.
    - If any one is a goal state, stop; else, select the k best successors from the list and go to the next level.

Example

- Consider version 1. Run a beam search with $k = 2$, with the smaller values the better.



- S, generate ABC, select AB, generate DEFH, select FH, expanded ndoes: SABFH.

Beam Search and Hill Climbing Search

- Beam search with 1 initial state and hill-climbing with 1 initial state:
  - Beam: 1 start node, at each step keeps k best nodes.
  - Hill-climbing: 1 start node, at each step keeps 1 best node.
- Beam search with k random initial states and hill-climbing with k random initial states.
  - Beam: k starting positions, k threads run in parallel, passing of useful information among them as at each step k children are selected.
  - Hill-climbing: k starting positions, k threads run individually, no passing of information.

Beam Search with A*

- Memory was a big problem with A*. **Idea**: keep only the best k nodes in the fringe e.g. use a priority queue of size k.
- Advantages: memory efficient. Disadvantages: neither complete nor optimal.

Simulated Annealing

- Annealing in metallurgy is when a material's temperature is gradually decreased (very slowly), allowing its crystal structure to reach a minimum energy state.
- Similar to hill-climbing but selects a random successor instead of the best successor.
  - Simulated annealing combines a hill-climbing step (accepting the best child) with a random walk step (accepting bad children with some probability). The random walk step can help escape bad local minima.

Implementation

- Set current node n to initial state s.
- Randomly select m, one of n's successors.
- If $v(m)$ is better than $v(n)$, $n = m$. Else: $n = m$ with a probability p.
- Go to step 2 until a predefined number of iteratioins is reached or the state reached is good enough.

Probability p

- Assume we are looking for a minimum, with parent n and child m.

$$p = e^{\frac{v(n)-v(m)}{T}}$$

- Main ideas:
  - p decreases exponentially with the badness of the childs move.
  - Bad children (moves) are more likely to be allowed at the beginning than at the end.
- Numerator shows how good the child m is:
  - For a bad move, $v(n) < v(m)$.
  - Case 1: $v(n) = 10, v(m) = 20, p_1 = e^{-10/T}$
  - Case 2: $v(n) = 10, v(m) = 11, p_2 = e^{-1/T}$
  - m in case 1 is worse than in case 2, $p_1 < p_2$ as T is positive $\rightarrow p$ exponentially decreases with the badness of the move.

- Denominator is a parameter T that decreases (anneals) over time based on a schedule e.g. $T = 0.8T$.
  - High T - bad moves are more likely to be allowed.
  - Low T - bad moves are less likely, becoming more like hill-climbing.
  - T decreases with time and depends on the number of iterations completed i.e. until "bored".
- Some versions have an additional step (Metropolis criterion for accepting child).
  - p is compared with p^*, a randomly generated number [0, 1].
  - If $p > p*$, accept the child, otherwise reject it.

Simulated Annealing Theorem

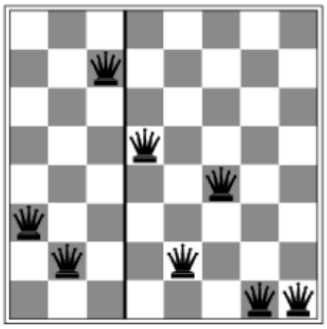
- What is the correspondence to metallurgy?
  - v is the total energy of the atoms in the material.
  - T is the temperature.
  - Schedule is the rate at which T is lowered.
- Theorem: if the schedule lowers T slowly enough, the algorithm will find global optimum.
  - i.e. complete and optimal given a long enough cooling schedule $\rightarrow$ annealing schedule is very important.
- It is widely used to solve VLSI layout problems, factory scheduling and other large-scale optimisations.
- It is easy to implement but a 'slow enough' schedule may be difficult to set.

Genetic Algorithms

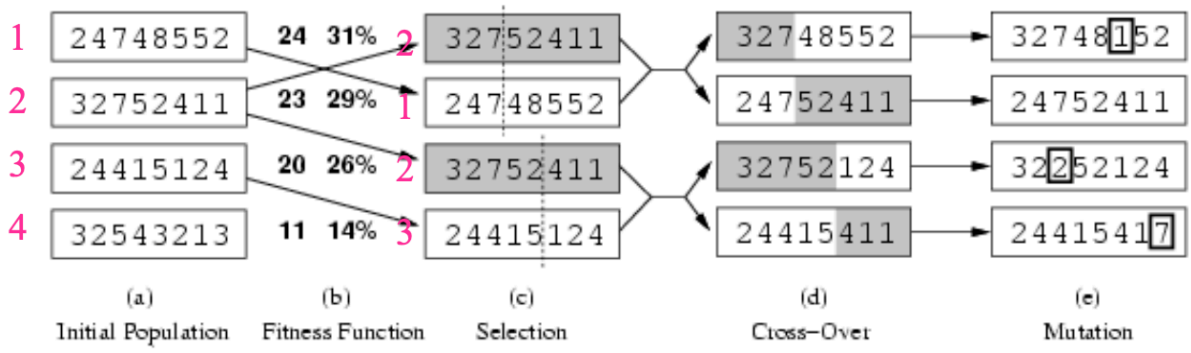
- Inspired by mechanisms used in evolutionary biology e.g. selection, crossover, mutation.
  - Similar to beam search, in fact a variant of stochastic beam search.
- Each state is called an individual. It is coded as a string. Each state n has a fitness score $f(n)$ (evaluation function). The higher the value, the better the state.
- **Goal**: starting with k randomly generated individuals, find the optimal state.
- Successors are produced by selection, crossover and mutation. At any time, keep a fixed number of states.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

8-Queens Problem Example

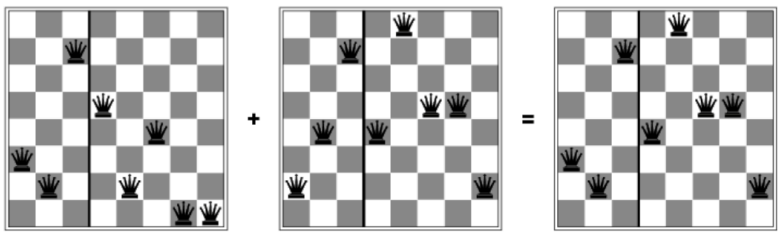


- Starting from bottom left, 1 to 8, upwards being the position, sideways being the columns.
  - One possible encoding is (3 2 7 5 2 4 1 1)
- Suppose that we are given 4 individuals (initial population) with their fitness values.
  - **Fitness Values**: number of non-attacking pairs of queens (given 8 queens, how many different pairs of queens are there? Max value is 28).
  - Let the probability for reproduction be proportional to the fitness (expressed as a %).



- Method:

- **Select**: 4 individuals for repoduction based on fitness function (the higher, the higher the probability to be selected). In the above case, 2 1 2 3.
- **Crossover**: random selection of crossover point; crossing over the parents strings.
- **Mutation**: random change of bits (in this case, 1 bit was changed in each individual).



(3 2 7 5 2 4 1 1) + (2 4 7 4 8 5 5 2) = (3 2 7 4 8 5 5 2)

- When the two states are different, crossover produces a state which is a long way from either parents.
- Given that the population is diverse at the beginning of the search, crossover takes big steps in the state space early in the process and smaller later, when the individuals are more similar.

Discussion

- Combines uphill tendency (based on fitness function) and random exploration (based on crossover and mutation).
- Exchange information among parallel threads - the population consists of several individuals.
- The main advantage comes from crossover.
- Success depends on the representation (encoding).
- Easy to implement, not complete, not optimal.

Lecture 4 - Game Playing

- **Why are games an appealing target for AI research?**
  - Games are too hard to solve:
    - Very large search space for common games. Chess: $b = 35, m = 100, 35^{100}$ nodes in the search tree, although there are only 10^{40} legal positions.
    - Require decisions in limited time.
    - No time to calculate the exact consequences of each move; need to make best guess based on previous experience, heuristics.
  - More like the real world than the toy search problems:
    - Optimal decisions are infeasible, yet some decision is required.
    - There is an unpredictable opponent to be considered.
  - Easy to represent as search problems:
    - A state is a board configuration.
    - Moves are typically a small number and well defined.
    - Clear criterion for success.

Characteristics of Games

- Deterministic vs Chance
  - Deterministic is when there is no chance element (no dice rolls, coin flips).
- Perfect vs Imperfect Information
  - Perfect is when there is no hidden information, there is a fully observable game and each player can see the complete game.
  - Imperfect is when some information is hidden e.g. player's cards are hidden from other players.
- Zero-sum vs Non Zero-sum
  - Zero-sum (adversarial): one player's gain is the other player's loss.

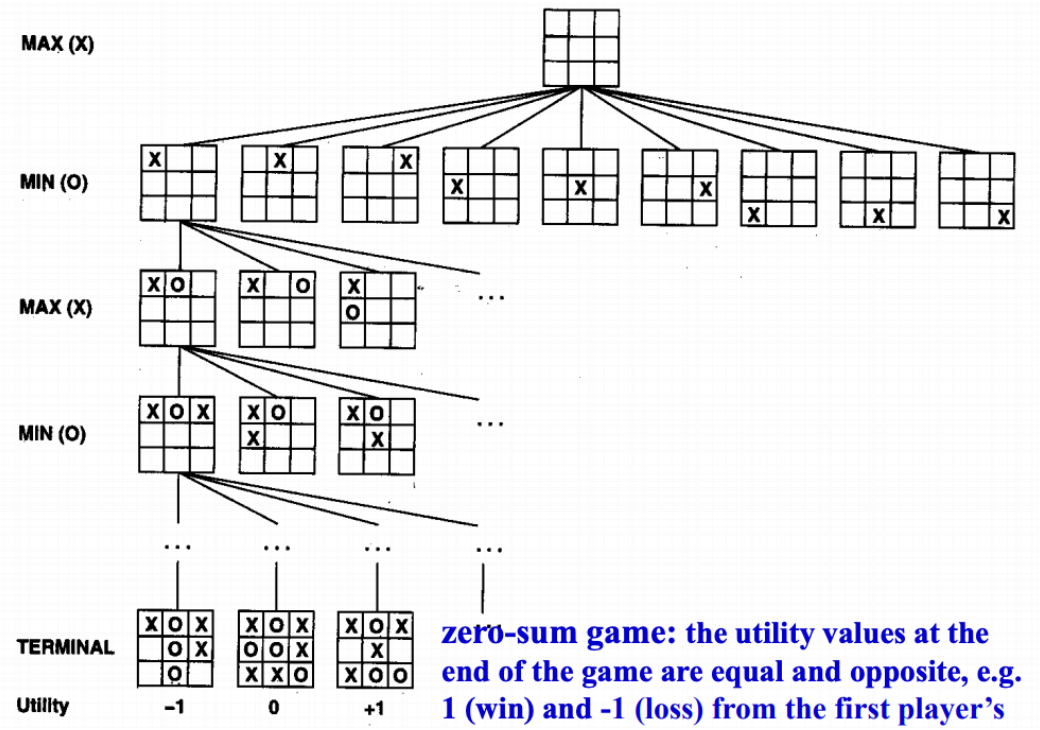
Types of Games

	deterministic	chance
perfect information	chess, checkers, connect-4	backgammon, monopoly
imperfect information		bridge, poker, scrabble

- Consider the specific setting where there are: 2 players max and min, players take turns, no chance is involved, there is perfect information, and it's a zero-sum game e.g. chess, checkers.
- A possible way to play would be to consider all legal moves that can be made from the current position, compute all new positions resulting from these moves, evaluate each new position and determine the best move. The key steps are:
  - Representing a position.
  - Evaluating a position.
  - Generating all legal moves from a given position.
- This can be represented as a search problem.

Game Playing as Search

Task	**Play a game by searching a game tree.**
State	Board configuration
Initial State	Initial board configuration and who goes first
Terminal States	Board configurations where the game is over
Terminal Test	When is the game over?
Operators	Legal moves a player can make
Utility Function (Payoff Function)	Numeric value associated with terminal states representing the game outcome e.g. positive = win for MAX, negative = win for MIN, 0 = draw (higher the better).
Evaluation Function	Numeric value associated with non-terminal states - shows how good the state is e.g. how close it is to a win
Game Tree	Represents all possible game scenarios



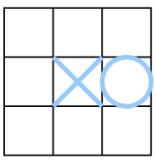
Evaluation Function

- A numeric value $e(s)$ associated with each non-terminal state s.
- Similar to the heuristic functions we studied before (e.g. an estimate of the distance to the goal).
- It gives the expected outcome of the game from the current position for a given player.
- Assuming that we are playing as MAX:
  - State $s \rightarrow$ number $e(s)$

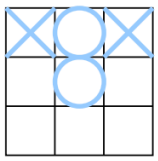
        - $e(s)$ is a heuristics that estimates how favourable s is for MAX.
        - $e(s) > 0$ means that s is favourable for MAX (the larger the better).
        - $e(s) < 0$ means that s is favourable for MIN.
        - $e(s) = 0$ means that s is neutral.
    - More generally, the higher the value $e(s)$, the more favourable the position s is for MAX.

Example for Tic-Tac-Toe

- MAX: cross, MIN: circle.
- $e(s)$ = number of rows, columns and diagonals open for MAX minus the number of rows, columns and diagonals open for MIN.



6–4 = 2



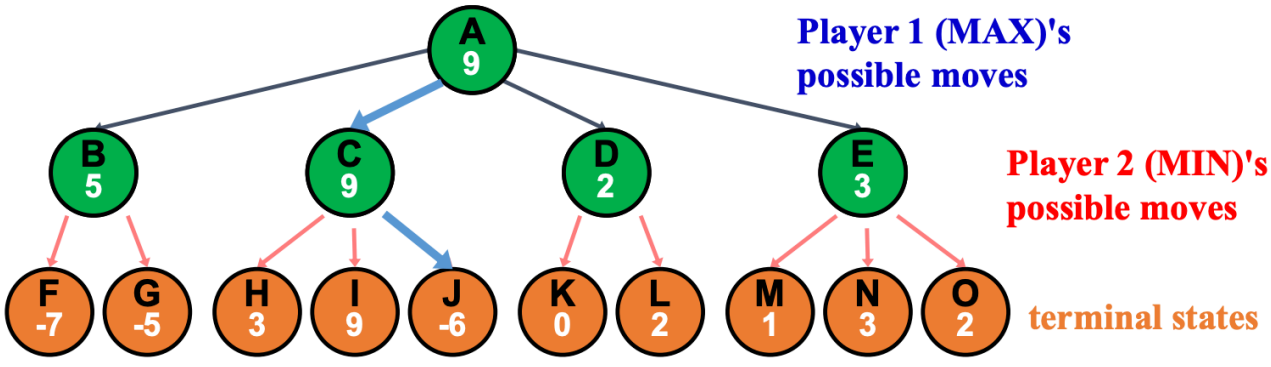
3–3 = 0

Game Playing as Normal Search

- Consider the following: Player 1 (MAX) is the first player. They search for a sequence of moves leading to a terminal state that is a winner (according to the utility function).
- Let's apply a normal search strategy e.g. greedy search.
    - Expand each to a terminal state i.e. build the game tree.
    - Choose the move with the best heuristic value for each player.

Using Greedy Search

- Player 1 chooses C (max value), Player 2 choose J (min value). J is a terminal node and the game ends, and since the node's value is negative, player 2 wins.
- The issue with this is that it does not look ahead any plays, and guarantees nothing for either player.



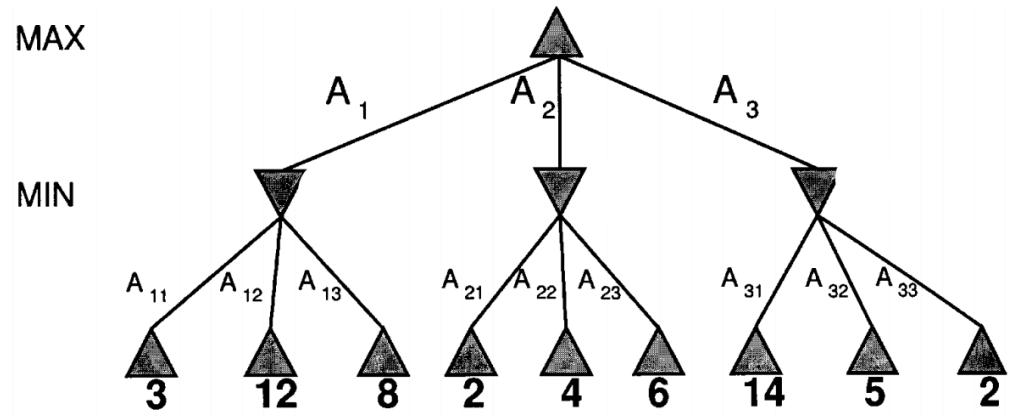
Minimax Algorithm

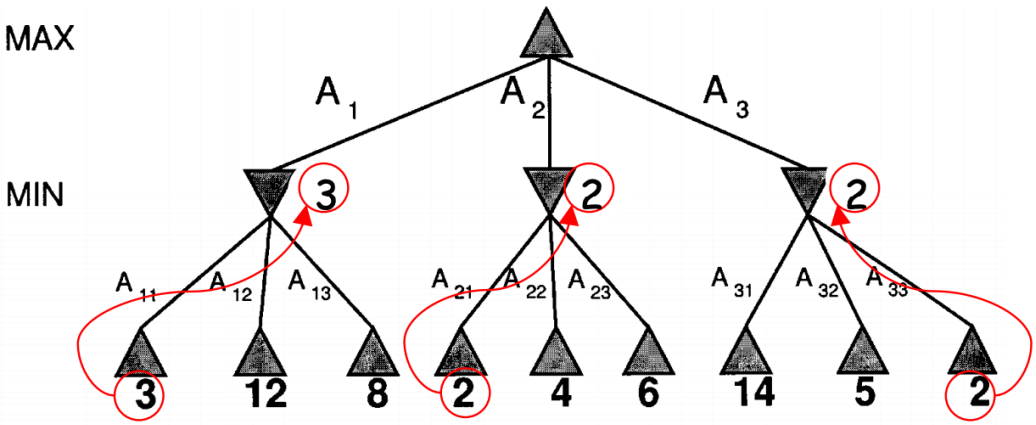
- Given a 2-player, deterministic, perfect information game:
    - **Minimax gives the perfect (best, optimal) move for a player, assuming that its opponent plays optimally** i.e. assuming the worst case scenario that the opponent also plays optimally.
    - A player plays optimally if he/she always selects the best move based on the evaluation function.

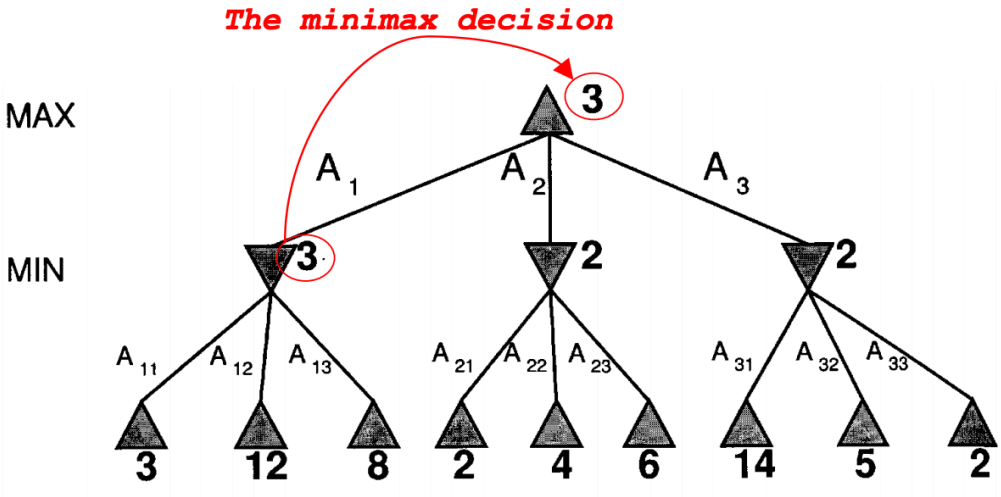
        - MAX - always selects the node with the max evaluation value
        - MIN - always selects the node with the min evaluation value.
    - Minimax computes a value for each non-terminal node, called a minimax value.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

- Minimax works the following way:
    - Generate the game tree
        - Create start node as a MAX node with the current board configuration.
        - Expand nodes down the terminal states (or to some depth if there is resource limitation, whether it be time or memory).
    - Evaluate the utility of the terminal states (lead nodes).
    - Starting at the leaf nodes and going back to the root of the tree, compute recursively the minimax value of each node (each player will select differently).
    - When the root node is reached, select the best move for MAX (max of the minimax values of children).







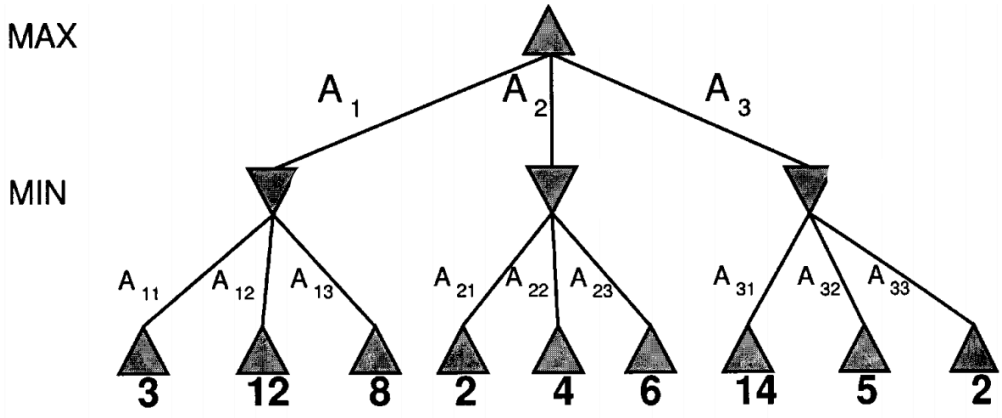
- MAX then chooses the move with the highest evaluation function, i.e. the left most.
    - If MAX follows the left/middle/right branch and MIN plays optimally, the utility is 3/2/2.

Implementation of Minimax

- For each move by MAX:
    1. Perform DFS to a terminal state.
    2. Evaluate each terminal state.
    3. Propagate the minimax values upwards (if MIN, min value, if MAX, max value of children backed up).
    4. Choose move with the maximum of minimax values of children.
- Note:
    - Minimax values generally propagate upwards as DFS proceeds in a left-to-right fashion.
    - Minimax values for sub-tree backed up as we go, so only $O(bd)$ nodes need to be kept in memory at any time.

What if MIN Does Not Play Optimally?

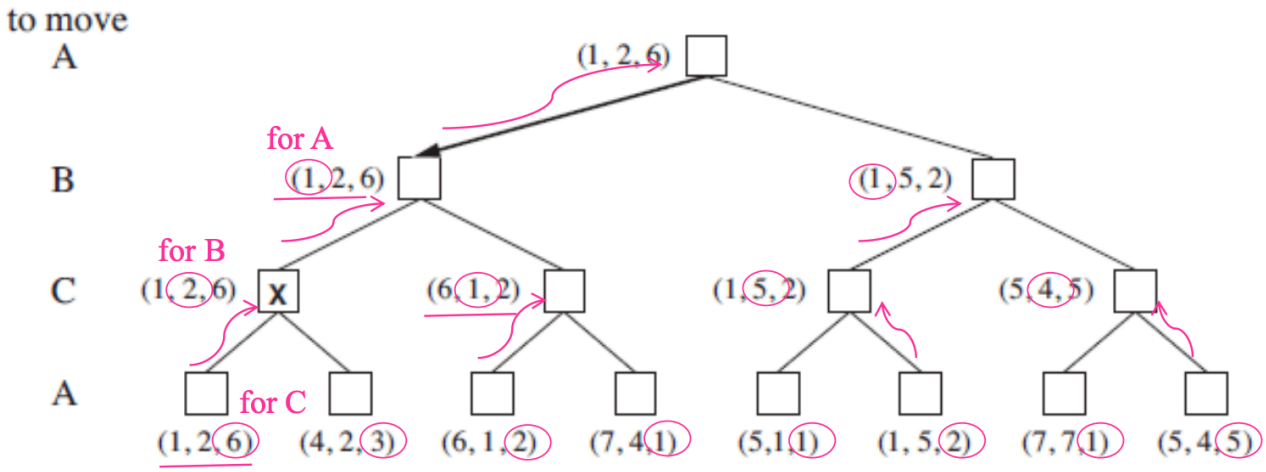
- If MIN does not play optimally, MAX will do even better.



- If MAX follows left/middle/right and MIN does not play optimally, the utility is 12 or 8/4 or 6/14 or 5, which is better for MAX than the result if MIN plays optimally 3/2/2.

Multiplayer Games

- Many games allow more than two players. In this case, single minimax values become vectors with a value for each player.
- Each player is maximising its own value.



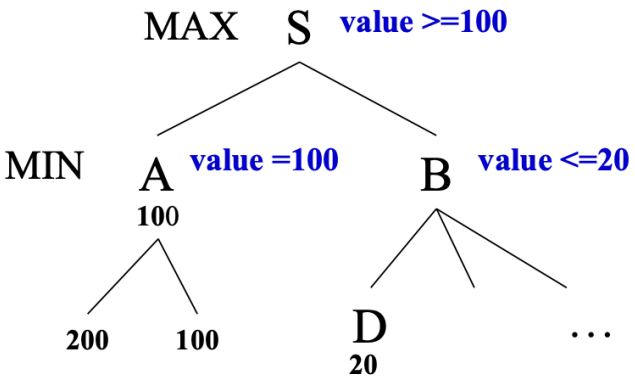
- In this case, A randomly selects either path since the value is both 1. Assuming that it follows down the left branch, B will select the leftmost branch since $2 > 1$, and C will then select the leftmost branch since $6 > 3$. The game will then end there, with C as the winner.

Properties of Minimax

- Implemented as DFS, assumptions are that the branching factor is b, and all terminal nodes at depth d.
- **Optimal**: will the optimal score be reached, where the optimal score is the score of the terminal node that will be reached if both players play optimally?
    - Yes, against an optimal opponent.
    - Against a suboptimal player, the score of the terminal node will never be lower than the optimal score.
- **Time**: $O(b^m)$ as in DFS.
    - This is the major problem, since moves need to be chosen in a limited time.
    - In most cases, we can only look forward so far before being forced into making a move. Suppose that:
        - We have 100 seconds per move.
        - In 1 second, we can explore 10 000 nodes.
        - At each move, we can explore 10^6 nodes. $b^m = 10^6 \text{ where b} = 35 , m = 4$ i.e. 4 ply ahead.
        - 4 plys = human novice, 8 ply = human master, 12 ply = Kasparov and Deep Blue.
- **Space**: $O(bm)$ as in DFS.

Not All Nodes are Worth Exploring

- There is no need to evaluate the other children of B as we already know that MAX will not choose B, it will choose A.

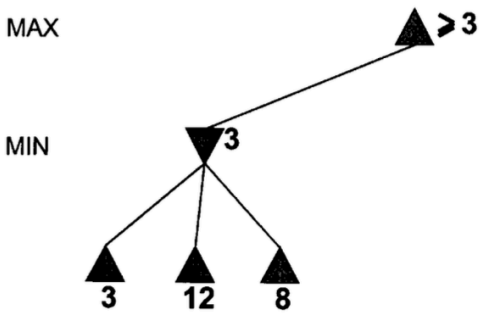


$$\text{Let the values of the 2 non-evaluated children of B be } x \text{ and } y$$
$$\text{minimax}(S) = \max(A, B)$$
$$= \max(\min(200, 100), \min(20, x, y))$$
$$= \max(100, \leq 20) = 100$$

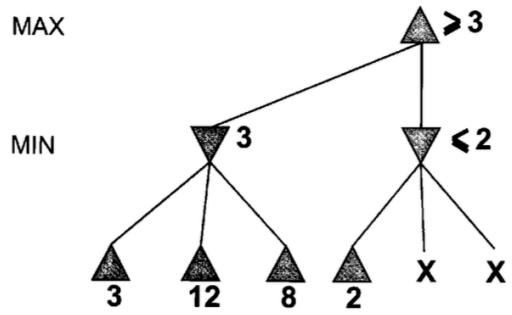
- The value of the root is independent of the values of the leaves x and y, we can prune x and y since there is no need to generate and evaluate them.

Alpha-Beta Pruning

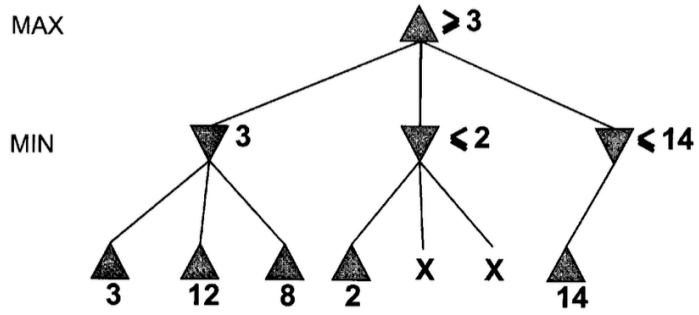
- If alpha-beta pruning is applied to a standard minimax tree, it returns the same move as minimax, but prunes away branches that cannot influence the final decision.
- While doing DFS of the game tree, along the path, keep 2 values:
  - **Alpha value** α - the best value for MAX so far along the path (initialised to -∞).
  - **Beta value** β - the best value for MIN so far along the path (initialised to ∞).
- If a MAX node exceeds beta, prune the sub-tree below.
- If a MIN node is below alpha, prune the sub-tree below.



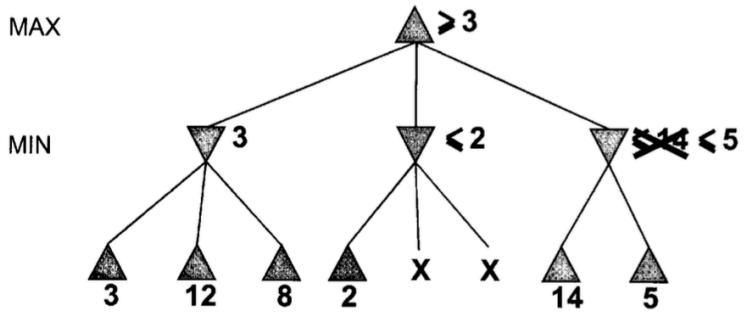
- **We always need to evaluate all children of the first branch – it is not possible to prune them**



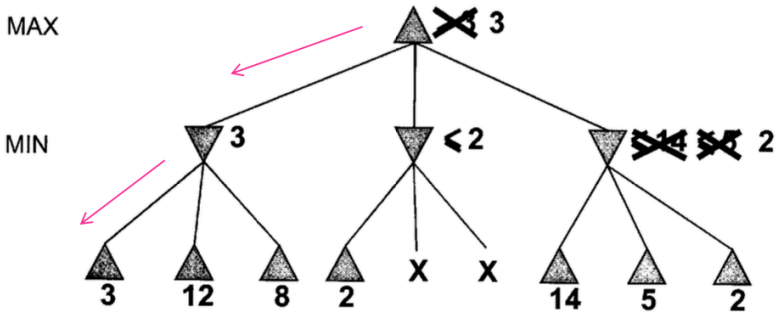
- **The intervals <=2 and >=3 do not overlap, so no need to evaluate the 2 children**



- **The intervals <=14 and >=3 overlap, so the evaluation of children should continue**



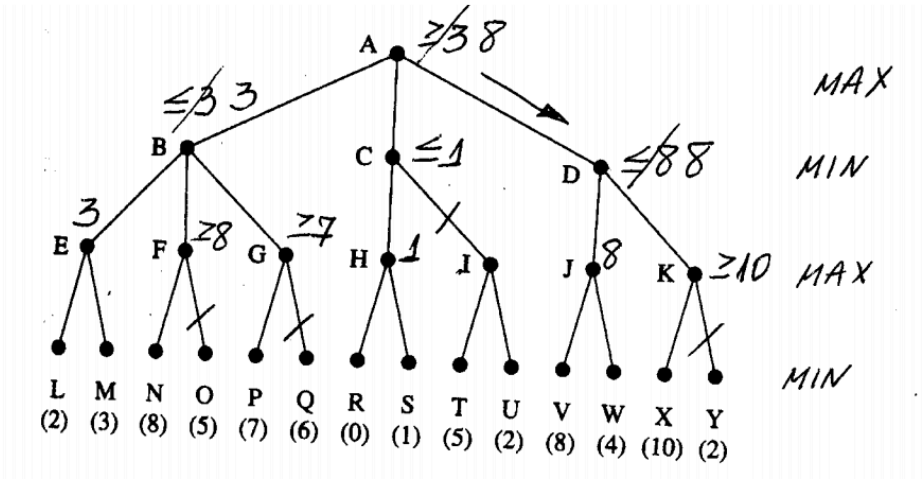
- **The intervals <=5 and >=3 overlap, so we should evaluate the remaining children (no pruning is possible)**



- **MAX should choose the left most path (value =3)**
- **MIN also should choose the left most path (value =3)**

Another Example

- Evaluate all children of E. Set bound on B. Find bound for F based on N. Check with bound on B. In this case, prune. Then evaluate with G.
- Once all children of B have been evaluated, set a hard value for B, and a bound for A. Continue onwards.

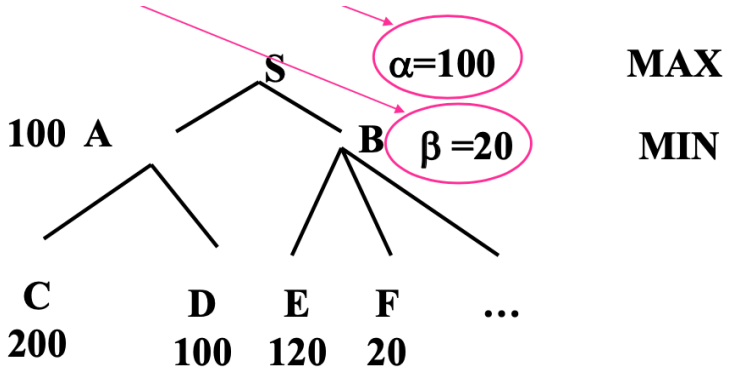


Implementation of Alpha-Beta Pruning

- Traverse the search tree in depth-first order.
- Apply utility function at each leaf node that is generated.
- Compute values backwards.
- At each non-leaf node, store a value indicating the best backed up value so far.
  - MAX nodes: α = best (max) value found so far.
  - MIN nodes: β = best (min) value found so far.
- Given a node n, cutoff the search below n if:
  - n is a MAX node and $\alpha(i) \geq \beta(i)$ for some MIN node i that is the ancestor of n.
  - n is a MIN node and $\beta(n) \leq \alpha(i)$ for some MAX node i that is the ancestor of n.

Alpha Cut Off

- If a MIN node is below alpha, prune it i.e. if child's beta $\leq$ parent's alpha.
- There is no need to expand B further as MAX can make a better move.

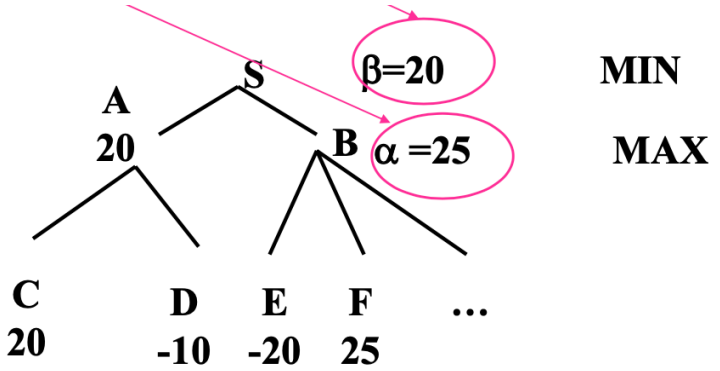


Alpha cutoff

β(B)=20<α(S)=100

Beta Cut Off

- If a MAX node exceeds beta, prune it i.e. if child's alpha $\geq$ parent's beta.
- There is no need to expand B further as MIN will not allow MAX to take the move.

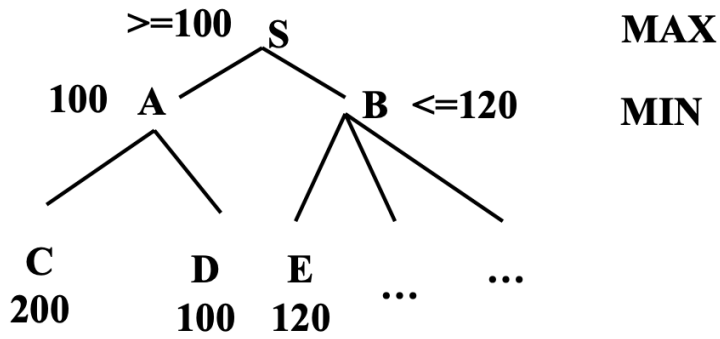


Beta cutoff

α(B)=25>β(S)=20

Pruning with Bounds

- Keep alpha and beta bounds, not values.
- If the intervals they define overlap, pruning is not possible.
- If the intervals they define do not overlap, pruning is possible.



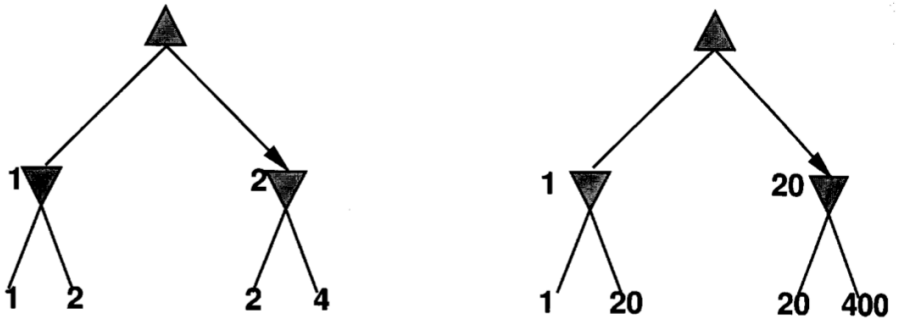
In this case, we need to evaluate the children of B, since the two bounds overlap.

Properties of Alpha-Beta Pruning

- **Pruning does not affect the final result.**
  - Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less of equal number of evaluated nodes.
- **Good move ordering improves effectiveness of pruning.**
  - Worst case: no pruning, examine b^d nodes (equal to minimax).
  - Best case ("perfect ordering"), examine only $b^{d/2}$ nodes, meaning double the depth of look-ahead search, for chess we can easily reach depth of 8 and play good chess.

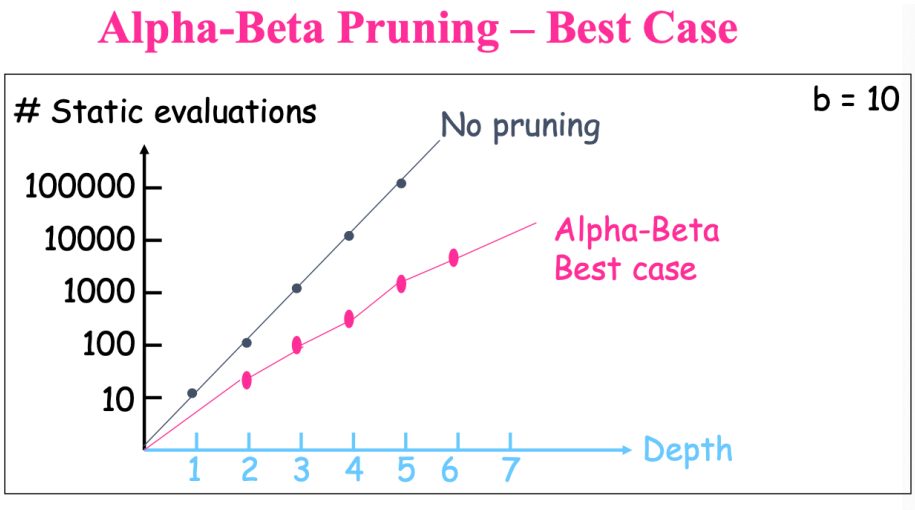
Exact Values Don't Matter Only the Relative Values Matter

- In deterministic games, the evaluation functions need not to be precise as long as the comparison of values is correct.
  - The relative values are important not the actual values.
- Behaviour is preserved under any monotonic transformation of the evaluation function.



Imperfect Real-Time Decisions

- Both minimax and alpha-beta require too many evaluations.
  - Minimax: the full game tree must be generated to maximum depth.
  - Alpha-beta: at least some parts of the game tree must be generated to maximum depth (the others may be pruned).
  - Example: It is impossible to generate the complete game tree for chess.
    - With 10^{40} nodes in the complete game tree, it would take 10^{22} centuries to generate it assuming that a child node could be generated in $1/3$ nano seconds.
    - The universe itself is 10^{8} centuries old.



- Both algorithms may be impractical and a move needs to be made in minutes at most.

Cutting Off Search Earlier

- Expand the tree only to a given depth (as opposed to maximum depth). The fixed depth is selected so that the amount of time will not exceed what the rules of the game allow.
- Apply heuristic evaluation function to the lead nodes, treating them as terminal nodes.
- Return values to the parents of the lead nodes as in minimax and alpha-beta.

- This means that changing minimax or alpha-beta in 2 ways:
  - Replace the terminal test by a cutoff test.
  - Replace the utility function by a heuristic evaluation function (=estimated desirability of a position).
- Pseudocode change in minimax and alpha-beta search:

```
if TERMINAL-TEST(state), then return UTILITY(state) becomes
if CUTOFF-TEST(state, depth) then return EVAL(state)
```

- Instead of fixed depth search (alpha-beta uses depth first search), iterative deepening (depth-first) search can be used.
  - When the time runs out, the program returns the move selected by the deepest completed search.

More on Evaluation Functions (For Chess)

- Typical evaluation function is a weighted linear sum of features:
  $EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$, where $f_i - \text{feature i}$, $w_i - \text{weight of feature i}$.
- Example:
  - f is the number of different pieces on the board e.g. f_1 for number of pawns, f_2 for number of bishops.
  - w is the material value of the piece e.g. $w = 1$ for pawn, $w = 3$ for bishop, $w = 9$ for queen.
- Other features:
  - Combinations of number of pieces e.g. #white queens $-$ #black queens.
  - Good pawn structure, king safety etc.

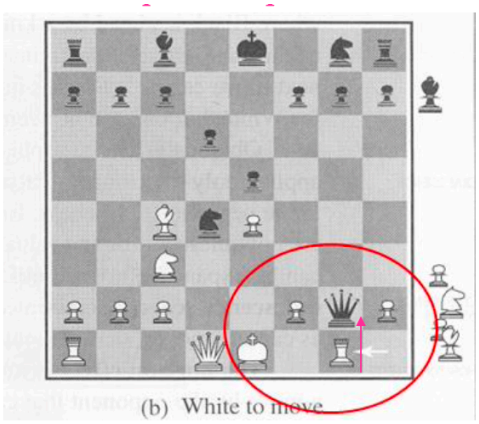
Some Problems with these Evaluations Functions

- A linear combination assumes that the pieces are independent from each other and have the same value during the game.
  - A pawn is more powerful when there are many other pawns on the board (good pawn structure) - the higher the number, the higher the value of the pawn.
  - Bishops are more powerful at the end of the game when they have a lot of space to move than at the beginning.
    - The bishop value correlates with the number of other pieces.
    - The bishop value does not have a constant value of 3 during the game.

Horizon Effect

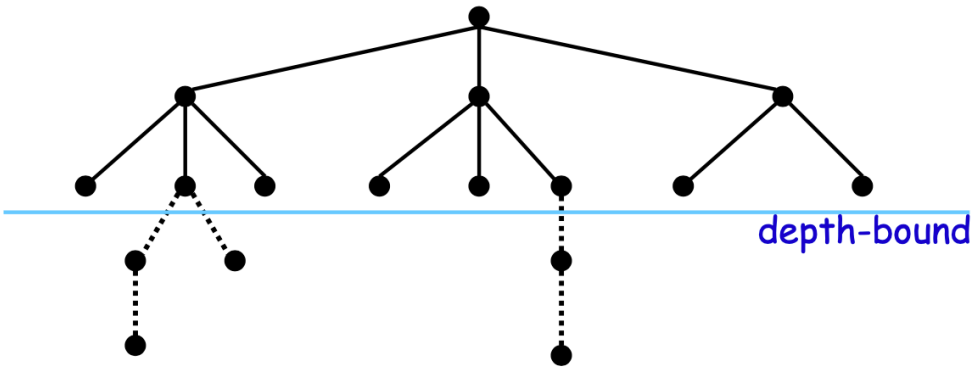
- Hidden pitfall not captured by the heuristic is a sudden damaging change just outside the search window that we can't see. It is one of the main difficulties in game playing programs.
  - Example: Black has material advantage but if White can advance his or her pawn, it becomes a queen and White will win.
  - We should not stop the search here, as we will not be able to see this sudden change.
- We have a heuristic that counts material advantage e.g. pieces won, and Black has an advantage. White is to move, and he/she will capture the queen and the heuristic value will drastically change.

- We can only see this is we look 1 more move ahead i.e. we should not stop the search at this position as it is not a quiescent position.



Solution

- A better cut-off test is needed.
- The evaluation function should be applied only to positions which are quiescent i.e. unlikely to change extremely in the near future.
    - Non-quiescent positions include: a capture can be made after it, a pawn becomes a queen after it.
- Solution: secondary search extends the search for the selected move to make sure that there is no hidden pitfall i.e. non-quiescent positions need to be expanded further until reaching a quiescent position. This is typically restricted to certain moves.
- In general, and especially in strategically crucial situations:
    - Select the most interesting nodes and extend search for them beyond the depth limit.
    - Check if king in danger, pawn to become queen, piece to be captured etc.

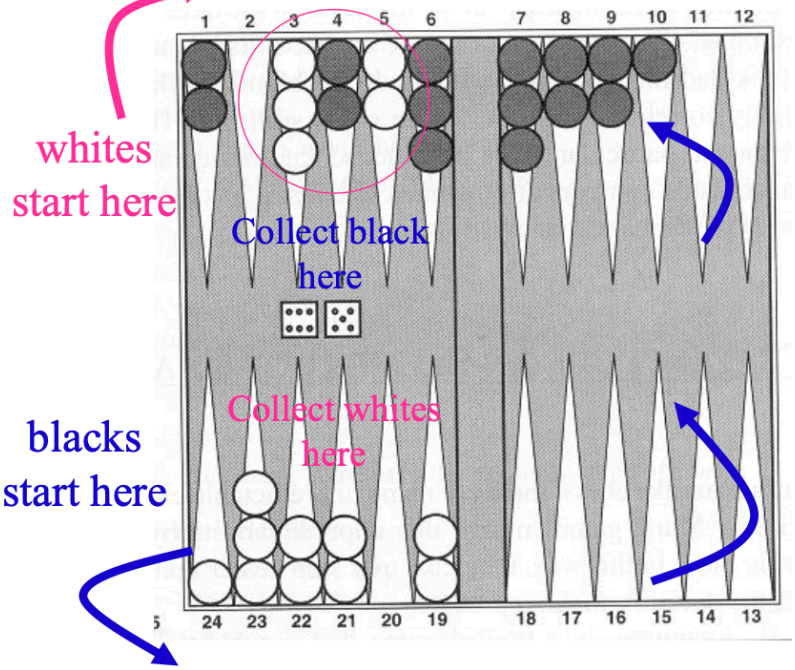


Other Refinements

- IDS with alpha-beta pruning.
- Book moves: catalogue of pre-computed sequences of moves for a particular board configuration e.g. chess opening sequence and endgame strategies, checkers endgame strategies.
- Heuristic pruning to reduce branching factor.
    - Ordering of the search tree based on how plausible the moves are (more plausible first).
- Alternative to minimax.
    - Risking a bad move may lead to a much better board configuration.

Non-Deterministic Games (Games of Chance)

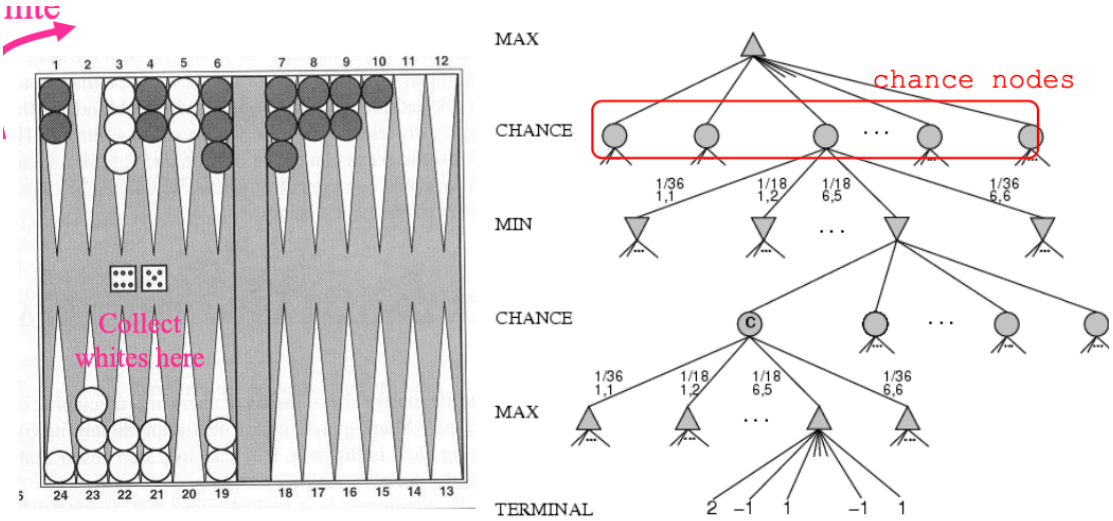
- Can we apply minimax and alpha-beta in games of chance?
  - **backgammon - the dice determines the legal move**



• White knows what his/her legal moves are
• But White doesn't know what Black will roll and, thus, what Black's legal moves will be
• => White cannot construct a standard game tree

Game Tree for Non-Deterministic Games

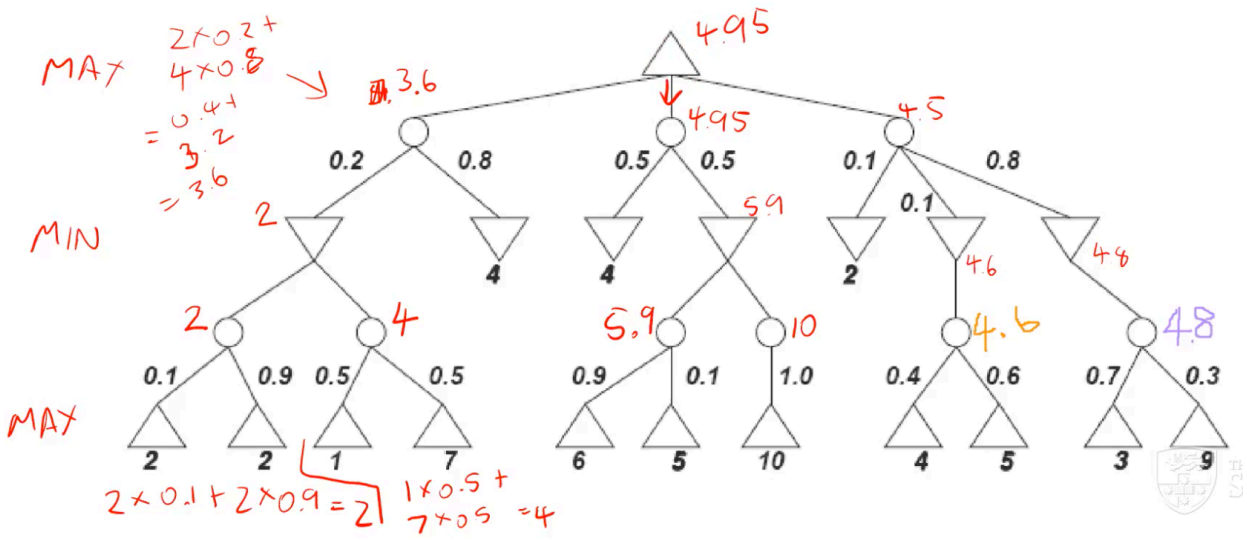
- The game tree includes chance nodes, corresponding to dice values. Because of the chance nodes, we cannot calculate definite minimax values, only expected value over all dice rolls.



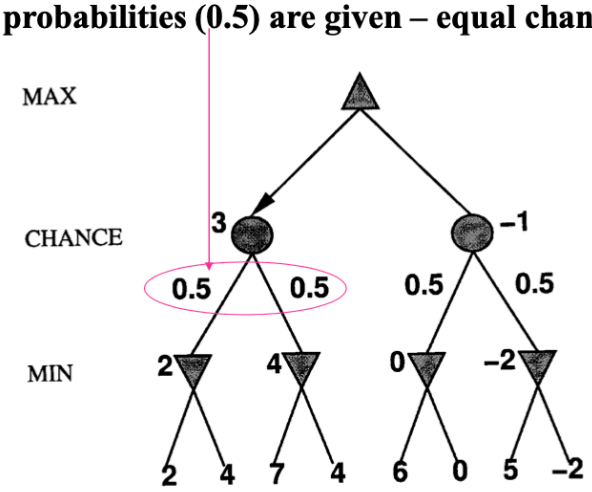
- Dice is (6, 5): so possible moves for white are: (5-10, 5-11), (5-11, 19-24), (5-10, 10-16) and (5-11, 11-16).
- When rolling 2 dice, how many possible outcomes are there? Are they equally likely or are some more likely?
  - How many distinct ways are there? Probability for each of them?
    - $[1,1], [2,2] \ldots [6,6]$ chance = $1/36$. All others have the chance $2/36$.

Expectiminimax Algorithm for Non-Deterministic Games

- As an extension of minimax for non-deterministic games, it gives the perfect play for non-deterministic games.
- Just like MINIMAX, except we must also handle chance nodes.
  - At terminal state nodes: use utility function.
  - At MIN nodes: minimum of the children node values.
  - At MAX nodes: maximum of the children node values.
  - At CHANCE nodes: weighted average of EXPECTIMINIMAX values resulting from all possible dice rolls.
- A version of alpha-beta pruning is possible but only if the leaf values are bounded.
- Time complexity: $O(b^m n^m)$, n being the number of distinct dice rolls.



- **Simplified example with coin-flipping instead of dice-rolling**
- **The probabilities (0.5) are given – equal chance for heads and tails**



Expectiminimax evaluates chance nodes by taking the average utility of all children, weighted by the probability of each child

3=2*0.5+4*0.5

Exact Values Do Matter

- Evaluation functions for games with chance should be carefully constructed.
- it is possible to avoid this - this evaluation function needs to satisfy a certain condition.
    - Be a positive linear transformation of the probability of winning from a position.

Learning Evaluation Functions (Example)

- **TD-Gammon (Tesauro 1992-1995) learns evaluation functions for backgammon from examples**
    - **Learner (classifier): neural network trained with the backpropagation algorithm**
        - **Maps input to output, and this mapping is learned from previous examples**
        - **Once trained, predicts the output for a new input (i.e. predicts the evaluation function for a new position)**
    - **Input: backgammon position**
        - **198 dimensional input vector**
    - **Output: value of the position**
        - **4 dimensional vector p**
        - **values are then combined, e.g. $v=p_1+2p_2-p_3-2p_4$**
    - **Training during actual play**
    - **Near championship level**

Game Playing - State of the Art

- AI algorithms have successfully won in:
    - Chess
    - Checkers
    - Othello
    - Go
    - Poker
- Further notes can be found in Lecture 4, from page 72 onwards.

Summary

- Many game programs are based on alpha-beta + iterative deepening + extended + transposition tables (store evaluation of repeated positions) + huge databases + ...
- For instance, Chinook searched all checkers configurations with 8 pieces of less and created an endgame database of 444 billion board configurations.
- Other methods (Go): deep learning (huge training set of previous games) and Monte Carlo search.
- The methods are general, but their implementation is dramatically improved by many specifically tuned up enhancements e.g. the evaluation function.

Lecture 5 - Introduction to Machine Learning, kNN, Rule-Based Algorithms: 1R

Introduction to Machine Learning

- Machine Learning (ML) is the area of AI that is concerned with writing computer programs that can learn from:
  - examples
  - domain knowlege
  - user feedback
- ML is the core of AI - without an ability to learn, a system cannot be considered intelligent.
- Operational Definiton: Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.
  - Computers learn when they change their behaviour in a way that makes them perform better in the future. Ties learning to performance rather than knowledge.
  - Any definition of learning with computers has to address questions of whether a machine has knowledge or become aware (or if it even can at all). It also has to go past 'committing to memory' or 'receiving instructions', since this is trivial for computers, and gives no indication if any benefit has been derived from the information.

Types of ML

Supervised ML

- Given: a set of pre-classified (labelled) examples $\{x, y\}$ where x is the input vector and y is the target output.
- Task: learn a function (classifier, model) which encapsulates the information in these examples (i.e. maps $x \rightarrow y$) and can be used predictively.
  - i.e. to predict the value of one variable (y) from the known values of other variables (x).
- **Two Types of Supervised Learning**:
  - Classification: the variable to be predicted is categorical i.e. values belong to a pre-specified, finite set of possbilities.
    - Given handwritten digits and their corresponding label (class), build a classifier that can recognise new handwritten digits - MNIST classification.
  - Regression: the variable to be predicted is numeric.
    - Given data from previous years (economic indicators, political events) and their corresponding exchange rate, build a classifier to predict the exchange rate for future days.
- Examples: 1R, kNN, DTs, NB, neural networks (perceptron, backpropagation, deep learning), SVM.

Examples of Classification

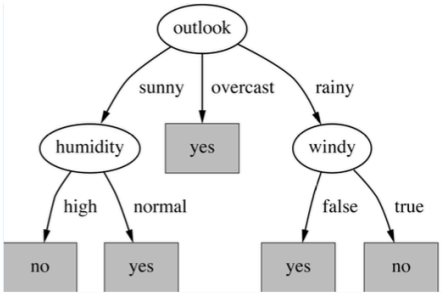
input data

input vector, with 4 features — target class

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

model (classifier)

model 1: decision tree



model 2: rules

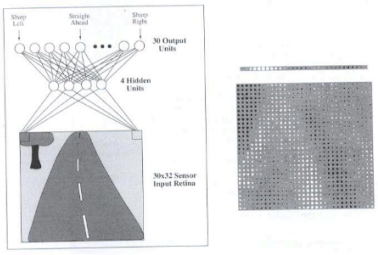
if `outlook=sunny` then `play=no`
elseif `outlook=overcast` then `play=yes`
elseif `outlook=rainy` then `play=yes`

model 3: …

We can learn different types of models

- Driving Motor Vehicles:

  - **ALVINN, Pomerleau et al., 1993**
  - **Driving a van along a highway**
  - **Uses a neural network classifier**
  - **Data**
    - **Input vectors: derived from the 30x32 image (black and white values)**
    - **Outputs (classes): 32 classes, corresponding to the turning directions - left, straight, right; different degrees**
    - **1 labelled example is: input vector + class label (turning direction)**



Examples of Regression

input

CPU performance data

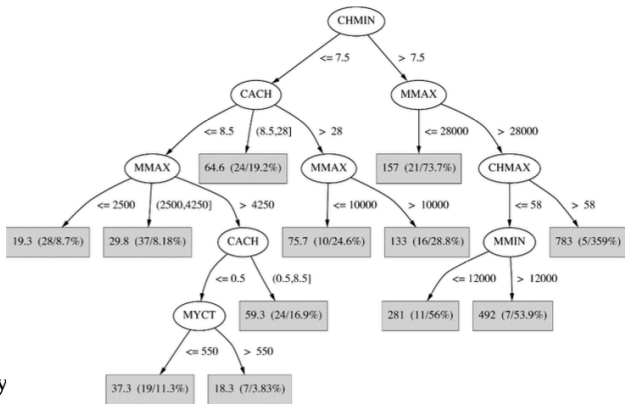
	cycle time (ns)	main memory (Kb)		cache	channels		perfor-mance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
3	29	8000	32000	32	8	32	220
4	29	8000	32000	32	8	32	172
5	29	8000	16000	32	8	16	132
...							
207	125	2000	8000	0	2	14	52
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

model

linear regression

```
PRP =  - 56.1 + 0.049 MYCT  + 0.015
MMIN
          + 0.006 MMAX + 0.630 CACH
          - 0.270 CHMIN + 1.46 CHMAX
```

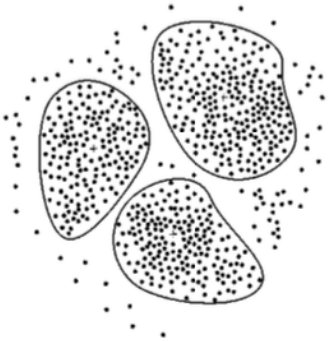
regression tree



Task: Predict computer performance

Irena Koprinska, irena.koprinska@sydney

Unsupervised ML (Clustering)



- Given: a collection of input vectors x, no target outputs y are given.
- Task: group (cluster) the input examples into a finite number of clusters so that the examples from each cluster are similar to each other, and examples from different clusters are dissimilar to each other.
- Examples: k-means, nearest neighbour, hierarchical clustering.
    - Customer profiling: a department store wants to segment its customers into groups and create a special catalog for each group. The attributes for the grouping included customer's income, location and physical characteristics (age, height, weight, etc.).
        - Clustering was used to find clusters of similar customers. A catalogue was created for each cluster based on the cluster characteristics and mailed to each customer.

Reinforcement Learning

- Each example has a score (grade) instead of correct output.
- Much less common than supervised learning.
- Most suited to control systems applications.

Associations Learning

- Developed with the database community in the early 90s. It finds relationships in data:

Market-Basket Analysis

- Find combinations in items that occur typically together. It uses the information about what customers buy to give us insights into who they are and why they make certain purchases.
    - A grocery store owner is trying to decide if they should put bread on sale. He generates association rules and finds what other products are typically purchased with bread. A particular type of cheese is sold 60% of the time the bread is sold and a jam is sold 70% of the time. Based on these findings, he decides:
        - 1) to place some cheese and jam at the end of the aisle where the bread is.
        - 2) not to place either of these 3 items on sale at the same time.

Sequential Analysis

- Goal: given a sequence of events, find frequent sub-sequences. These patterns are similar to market-basket analysis but the relationship is based on time.
    - Example: the webmaster at a company X periodically analyses the web log data to determine how users of X browse them. He finds that 70% of the time, users of page A follow one of three patterns.
        - A B C
        - A D B C
        - A E B C
    - A C is a frequent pattern, so a link is created for these pages.
    - Also used to find sub-sequences in DNA data for particular species.

Why is ML Important?

- Some tasks cannot be defined well, except by examples e.g. recognising the gender of people, handwritten digits etc.
- The amount of knowledge available about certain tasks is too big for explicit encoding into rules or difficult to extract from experts) e.g. it is easier to learn the relationship between symptoms and diagnosis from cases for medical diagnoses.
- Need for adaptation. Humans often produce machines that do not work as well as desired in the environments in which they are used, especially if these environments change over time e.g. spam email filter.
- Relationships and correlations can be hidden within large amounts of data (in this case ML and Data Mining may be able to help).

More ML Applications

- Fraud detection
  - Medical insurance fraud, inappropriate medical treatment.
  - Credit card services, phone card and retail fraud.
  - Historical transactions and other data.
- Sport - analyzing game statistics (shots blocked, assists and fouls) to gain competitive advantage.
  - “When player X is on the floor, player Y’s shot accuracy decreases from 75% to 30%”.
- Astronomy
  - JPL and the Palomar Observatory discovered 22 quasars using ML.
- Web applications
  - Mining web logs to discover customer preferences and behavior, analyze effectiveness of web marketing, improve web site organization.

Data Mining

- Data Mining: search for hidden patterns in large datasets, which should be meaningful, useful and actionable.
- Most of the techniques used for Data Mining have been developed in ML.
  - DM deals with large and multidimensional data, which is not necessarily true for ML.
  - DM is applied ML.
- Motivation for DM: Data Explosion - huge databases.
  - Due to automated data collection tools and mature database technoloy e.g. supermarket transaction data, credit card usage, Wikipedia, molecular databases etc.
  - We are drowning in data but starving for knowledge!

Classification

- **Given**: a set of pre-labelled examples.
  - 14 examples, 4 attributes: outlook, temperature, humidity, and windy, with the class being play (yes, no).
- **Task**: build a model (classifier) that an be used to predict the class of new (unseen) examples given a combination of attributes.
- Examples used to build the model are called training data.
- Success is measured empirically on another set called test data.
  - Test data hasn't been used to build the classifier; it is also labelled.
  - Performance measure: accuracy - proportion of correctly classified test examples.

attributes (features, variables) class

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

- Two types of attributes (features):
  - Numeric (continuous): their values are numbers.
  - Nominal (categorical): their values belong to a pre-specified, finite set of possibilities.

Commonly Used Distance Measures

- A, B are examples with attribute values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ e.g. $A = [1, 3, 5]$ and $B = [1, 6, 9]$.
- Euclidian Distance - most frequently used

$$D(A, B) = \sqrt{(a_1 - b_2)^2 + (a_2 - b_3)^2 + \ldots + (a_n - b_n)^2}$$

- Manhattan Distance

$$D(A, B) = |a_1 - b_1| + |a_2 - b_2| + \ldots + |a_n - b_n|$$

- Minkowski Distance - generalisation of Euclidian and Manhattan

$$D(A, B) = (|a_1 - b_1|^q + |a_2 - b_2|^q + \ldots + |a_n - b_n|^q)^{1/q}$$

- Minkowski distance and Manhattan distance are equal when $q = 1$.

Nearest Neighbour Algorithm (1-Nearest Neighbour)

- Remember all of your training data. To classify a new unlabelled example, find the nearest training data example and return the answer (class) associated with it.
- 'Nearest' is determined based on a distance measure.
- Given is the following dataset where $years\_experience$ is the only attribute and $salary$ is the class. What will be the prediction for the new example $years\_experince = 5$ using the Nearest Neighbor algorithm with the Manhattan distance?

	years_experience	salary	D(new, current)
1	4	low	1
2	9	medium	4
3	8	medium	3
4	15	high	10
5	20	high	15
6	7	medium	2
7	12	medium	7
8	23	medium	18
9	1	low	4
10	16	medium	11

- The closest neighbour is number 1 with a distnace of 1. The nearest neighbour will predict that $salary = low$.

Normalisation

Need for Normalisation

- Different attributes are measured on different scales.
- When calculating the distance between two examples, the effect of the attributes with the smaller scale will be less significant than those with the larger scale.
- Example: predict the car petrol consumption based on car weight (kg) and number of cylinders,
  - Example 1: 6000, 4
  - Example 2: 1000, 2
  - $D(\text{Ex1}, \text{Ex2}) = 5002$. In this case, the effect of the number of cylinders is lost as car weight dominates the calculation.

Normalisation Process

- Solution is to normalise the attribute values between 0 and 1.

$$a_i = \frac{v_i - \min v_i}{\max v_i - \min v_i}$$

where v_i is the actual value of attribute i and $\min v_i$ and $\max v_i$ is the minimum and maximum value of v_i taken over all examples in the training set (for attribute i).

Ex.: Consider a dataset with 4 examples
values of the attribute *car weight*:
original: 1000, 6000, 8000, 10000
normalized: 0, 0.55, 0.77, 1

values of the attribute *number of cylinders*:
original: 2, 4, 6, 4
normalised: 0, 0.5, 1, 0.5

original -> normalised
ex1: 1000, 2 0, 0
ex2: 6000, 4 0.55, 0.5
ex3: 8000, 6 0.77, 1
ex4: 10000, 4 1, 0.5
dist(ex1,ex2)=|0-0.55|+|0-0.5|=1.05
Both attributes are contributing,
***car weight* doesn't dominate**

Lazy Learning

- Nearest neighbour is also called instance-based learning, and is an example of lazy learning - stores all training examples and does not build a classifier until a new (unlabelled) example needs to be classified).
- Opposite to eager learning, which constructs a classifier before receiving new examples to classify e.g. 1R, decision trees, Naive Bayes, neural networks, SVM.
- Lazy classifiers are faster at training but slower at classification - training = memorising, with all computations delayed to classification time.

Computation Complexity - Time and Space

- Training is fast since no model is built; just storing training examples.
- **Time Complexity**:
  - During the classification phase (prediction for a new example), we need to compare each new example with each training example.
  - If m training examples with dimensionality n, the lookup for 1 new example takes $m \times n$ computations i.e. $O(mn)$.
- **Memory Requirements**:
  - $O(mn)$, we need to remember each training example. This is impractical for large datasets due to time and memory requirements.
  - Variations use more efficient data structures (KD and ball trees).

k-Nearest Neighbour Algorithm

- Using the k closest examples is called the k-Nearest Neighbour.
  - Majority voting is used to take the decision e.g. 3-Nearest Neighbour - circle, circle, square would be classified as circle.
  - Increases the robustness of the algorithm to noisy examples.
  - Increases the computational time (slightly).
- kNN is very sensitive to the value of k.
  - Rule of thumb: $k \leq \sqrt{\#\text{training\_examples}}$. Commercial packages typically use $k = 10$.
- Can be used also for regression tasks. The classifier returns the average value of the target variable associated with the k nearest neighbours of the new example
  - i.e. prediction of new example:
    $X_{\text{new}} = (\text{target\_value\_neighbour}_i + ... + \text{target\_value\_neighbour}_k)/k$
- In the $years\_experience$ example above, 3-Nearest Neighbour will predict $salary = medium$ based on neighbours 1, 6, 3.

Distance for Nominal Attributes

- Consider:
  - Example 1: (red, new)
  - Example 2: (blue, new)
  - $\text{Distance}(\text{Ex1}, \text{Ex2}) = ?$
- Simple solution is to assign 1 for the difference between two attribute values that are not the same, and 0 for the difference between two attribute values that are the same (no need for nomralisation).

- In this case, red and blue differ but new and new do not so $\text{Manhattan\_Difference}(\text{Ex1}, \text{Ex2}) = 1 + 0 = 1$.

Distance for Numeric Attributes with Missing Values

- Principle: a missing value is maximally different from any other value.
  - Example 1: (0.2, ?)
  - Example 2: (0.7, 0.1)
  - $D(\text{Ex1}, \text{Ex2})$ = ?
- Firstly, normalise the data. If two values are missing, the difference between them is 1. If only one of them is missing, the difference is either the other value or 1 minus that value - whichever is larger.
- $\text{Manhattan\_Difference}(\text{Ex1}, \text{Ex2}) = |0.2 - 0.7| + 0.9 = 1.4$

Variations: Weighted Nearest Neighbour

- **Idea**: closer neighbours count more than distant neighbours.
- Distance-weighted nearest-neighbour algorithm.
  - Find the k nearest neighbours.
  - Weight their contribution based on their distance to the new example.
    - Bigger weight if they are closer.
    - Smaller weight if they are further.

$$w_i = \frac{1}{D(X_{\text{new}}, X_i)}$$

- More often used for regression tasks e.g. $X_{\text{new}} = w_i \times \text{target\_value\_neighbour}_i + ... + \text{w\_k} \times \text{target\_value\_neighbour}_k$
- Another variation: instead of using only the k nearest neighbours, use all training examples.
  - The very distant will have very little effect since the weight is too small. However, it is a slower algorithm.

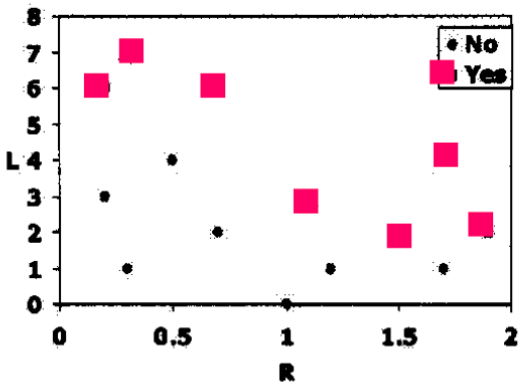
Curse of Dimensionality

- High dimensional data causes problems for all classifiers with overfitting.
- Nearest neighbours algorithms are great in low dimensions up to 6, but as the dimensionality increases, they become ineffective.
- In high dimensions, most examples are:
  - Far from each other.
  - Close to the boundaries. Imagine sprinkling data points uniformly within a 10-dimensional unit hypercube (cube whose sides are of length 1), 90% of the data are outside a cube with sides > 0.63.
- The notion of nearness that is very effective in low dimensional space becomes ineffective in a high dimensional space.
- The nearest neighbour classification cannot be trusted for high-dimensional data.

- **Solution**: feature selection to reduce dimensionality.

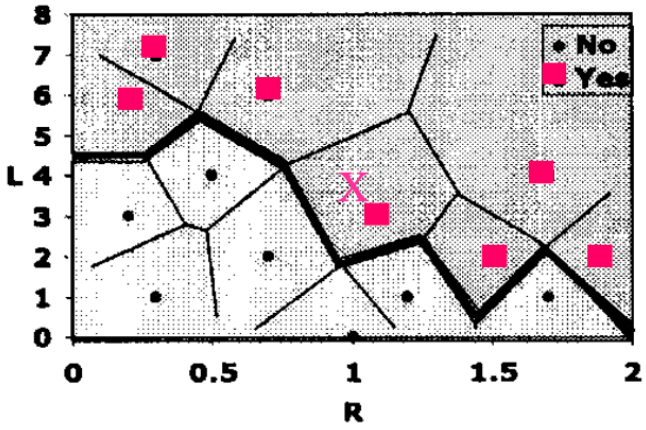
1-Nearest Neighbour Decision Boundary

Bankruptcy Example



- **Predicting bankruptcy**
- **2 attributes:**
  on x: R – ratio of earnings to expenses
  on y: L – number of late payments on credit cards over the past year
- **2 classes:**
  yes – high risk for bankruptcy
  no - low risk for bankruptcy

- The training defines a partioning in the feature space called a Voronoi partition (tessellation).
  - Each training example has an associated Voronoi region.
  - Voronoi region for a point (training example) is its nearby area.
  - More precisely, if a new example Y falls in the Voronoi region of the training example X, then X is the closest training example to Y.
- In 1NN, a decision boundary is represented by the edges in the Voronoi space that separate the points of the two classes.



kNN Discussion and Summary

- Simple method; works well in practice (like 1R and Naive Bayes).
- Slow for big databases as the entire database has to be compared with each testing example.
  - Requires efficient indexing + there are more efficient variations.
- Curse of dimensionality – “nearness” is ineffective in high dimensions.
  - Solution: feature selection to reduce dimensionality.
- Very sensitive to irrelevant attributes. Solutions:
  - Weight each attribute when calculating the distance.
  - Feature selection to select informative attributes.
- Produces arbitrarily shaped decision boundary defined by a subset of the Voronoi edges.

- High variability of the decision boundary depending on the composition of training data and number of nearest neighbors.
- Sensitive to noise.
- Local vs Global information.
    - The standard k-nearest neighbor algorithm (which considers only a few neighbours) makes predictions based on local information.
    - Other ML algorithms, e.g. 1R, decision trees and neural networks try to find a global model that fits the training set.

Rule Learning

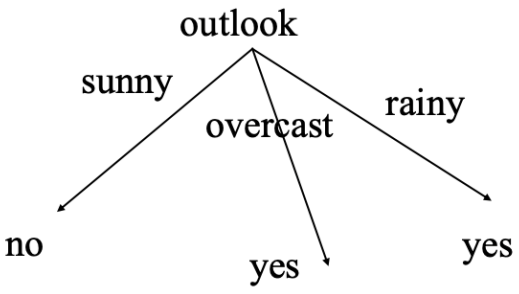
1R Algorithm

- Stands for '1-rule' in that it generate one rule that tests the value of a single attribute e.g. a rule that tests the value of `outlook`.

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

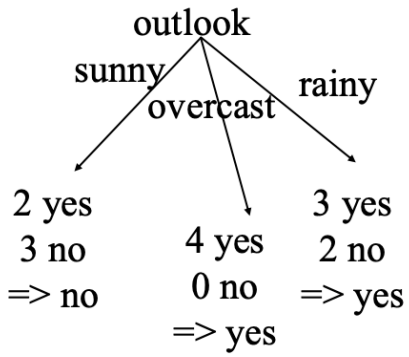
if outlook=sunny then class=no
elseif outlook=overcast then class=yes
elseif outlook=rainy then class=yes

- The rule can be represented as a 1-level decision tree (decision stump).
    - At the root, test the attribute value.
    - Each brach corresponds to a value.
    - Each leaf corresponds to a class.



How to Determine the Class for the Leaves?

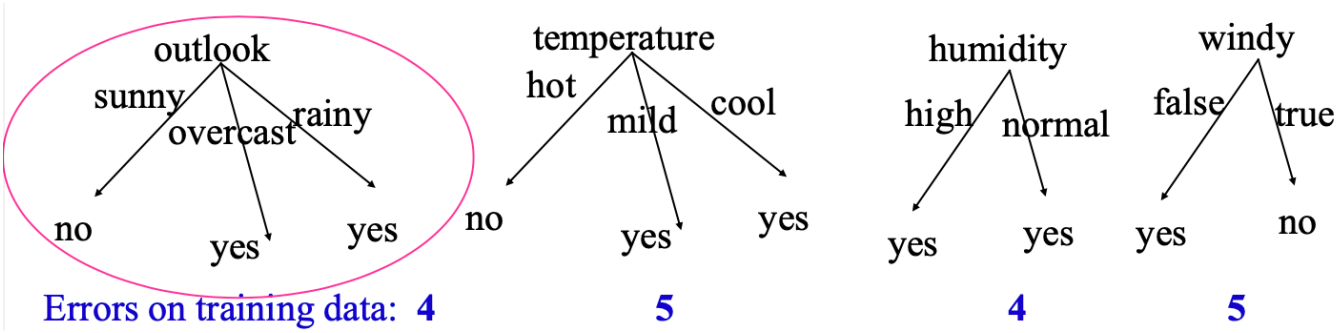
- Take the majority class of the leaf. This means minimising the number of examples from the training data that will be misclassified.



outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

How to Select the Best Rule?

- A rule is generated for each attribute. The best one is the one with the smallest number of misclassifications or errors on the training data (one with the highest accuracy).
- 1R generates a rule (decision stump) for each atttribute, evaluates each rule on the training data, calculates the number of errors and chooses the one with the smallest number of errors.



rule No	attri-bute	attribute values & counts	rules	errors	total errors
1	outlook	sunny: 2 yes, 3 no overcast: 4 yes, 0 no rainy: 3 yes, 2 no	sunny -> no overcast -> yes rainy -> yes	2/5 0/4 2/5	4/14
2	temp.	hot: 2 yes, 2 no* mild: 4 yes, 2 no cool: 2 yes, 1 no	hot -> no mild -> yes cool -> yes	2/4 2/6 1/4	5/14
3	humidity	high: 4 yes, 3 no normal: 6 yes, 1 no	high -> yes normal -> yes	3/7 1/7	4/14
5	windy	true: 3 yes, 3 no* false: 6 yes, 2 no	false -> yes true -> no	2/8 3/6	5/14

- Ties are broken by random choice. In this case, we arbitrarily choose rule 1 instead of rule 3.

Pseudocode and Components

- For each attribute:
    - For each value of that attribute, make a rule as follows:
        - Count how often each class appears.
        - Find the most frequent class.
        - Make the rule assign that class to this attribute value.
    - Choose the rule with the smallest error rate.
- Calculate the error rate of the rules.

Components of 1R

- **Model**: a rule testing the value of one attribute.
- **Preference Function**: number of misclassifications on training data.
- **Search Method**: evaluate all attributes.

Handling Examples with Missing Values

- How do you use 1R or other classification algorithms if there are missing values? (general topic not limited to 1R).

Method 1: Ignore all instances with missing attribute values. However:

- These instances are often useful.
- Sometimes the attributes whose values are missing play no part in the decision, in which case these instances are as good as any other.

Method 2 (Used in 1R): Treat the missing value as another possible value of the attribute; this assumes that the absence of a value is significant e.g. if there are missing `outlook` values, a 1R rule for `outlook` will consider the normal three plus `missing`.

Method 3: Two common ways. Replace the missing values with:

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
?	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	?	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	?	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

a) the most common value for attribute A in all training examples (`mild`, most common for `temp`).

b) most common value for attribute A in all training examples from the same class as the class of the example with the missing value (`mild`, most common from class `yes`).

Dealing With Numeric Attributes

outlook	temp.	humidity	windy	play
sunny	85	85	false	no
sunny	80	90	true	no
overcast	83	86	false	yes
rainy	70	96	false	yes
rainy	68	80	false	yes
rainy	65	70	true	no
overcast	64	65	true	yes
sunny	72	95	false	no
sunny	69	70	false	yes
rainy	75	80	false	yes
sunny	75	70	true	yes
overcast	73	90	true	yes
overcast	81	75	false	yes
rainy	71	91	true	no

- `temperature` and `humidity` are numeric.
- Need to discretize numeric attributes i.e. convert them to nominal.
- The simple procedure is as follows:
  - Sort the training examples in increasing order according to the value of the numeric attribute.

values of temperature:

64	65	68	69	70	71	72	73	75	75	80	81	83	85
yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no

  - Place breakpoints whenver the class changes, halfway between the values.
  - Take the class for each interval and form the rule.

if temperature <64.5 than play=yes
elseif temperature ∈ [64.5, 65.5) than play=no
elseif temperature ∈ [66.5, 70.5) than play=yes
elseif temperature ∈ [70.5, 72.5) than play=no
elseif temperature ∈ [72.5, 77.5) than play=yes
elseif temperature ∈ [77.5, 80.5) than play=no
elseif temperature ∈ [80.5, 84) than play=yes
elseif temperature >=84 than play=no

Problem With Discretization Procedure

- It tends to generate a large number of intervals, i.e. generates nominal attributes with many values (highly branching attributes).
- This is a problem and may lead to overfitting: the error on the training set is very small but when new data is presented to the classifier, the error is high.
  - The classifier has memorised the training examples but has not learned to generalise to new examples.
- Overfitting in 1R due to noise in data: 1 training example with an incorrect class will most likely generate a separate interval (condition in the rule). This new interval is misleading - it was generated because of noise and so the rule may not classify new examples that fall in this interval well.

A Better Discretization Procedure

- Solution is to aggregate. Introduce a threshold - requirement for a minimum number of examples of the majority class in each partition e.g. minimum number = 3.

64	65	68	69	70	71	72	72	75	75	80	81	83	85
yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no
		yes					yes				no		

- When adjacent partitions have the same majority class, merge them.

64	65	68	69	70	71	72	72	75	75	80	81	83	85
yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no
					yes						no		

if `temperature` ≤ `77.5` then `play=yes`

elseif `temperature > 77.5` then `play=no*`

*** Arbitrary choice `no` for the 2d interval; if `yes` has been chosen => no need for any breakpoint at all**

- The final rule set is as below, with Rule 3 i.e. `humidity` chosen.

rule No	attri-bute	Rules	errors	total errors
1	outlook	sunny -> no overcast -> yes rainy -> yes	2/5 0/4 2/5	4/14
2	temp.	≤ 77.5 -> yes > 77.5 -> no*	3/10 2/4	5/14
3	humidity	≤ 82.5 -> yes > 82.5 & ≤ 95.5 -> no >95.5 -> yes	1/7 2/6 0/1	3/14
5	windy	false -> yes true -> no	2/8 3/6	5/14

Decision Boundary

- 1R defines a decision boundary in the feature space. Supposing that the 1R split is binary, the decision boundary is either a vertical or horizontal line.

Why Do Simple Algorithms Work Well?

- 1R was described in a paper by Holte (1993). They were slightly less accurate than the most complex decision trees at the time. It is simple, computational very cheap, and can give a useful first impression of the dataset.
- The structure underlying many real world problems may be quite rudimentary e.g.
  - Just one attribute is sufficient to determine the class of the example.
  - Several attributes contribute independently with equal importance.
  - A linear combination of attributes may be sufficient.

Lecture 6 - Statistical-Based Learning (Naive Bayes), Evaluating and Comparing Classifiers

Bayes Theorem

- Bayesian classifiers are statistical classifiers. They can predict the class membership probability i.e. the probability that a given example belongs to a particular class, based on the Bayes Theorem.
- **Given a hypothesis H and evidence E for this hypothesis, then the probability of H given E is**:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

- For instances of fruit, let E is red and round and H is the hypothesis that E is an apple.
  - $P(H)$ is the probability that any given example is an apple, regardless of how it looks. Called the **prior** probability of H.
  - $P(E)$ is the prior probability of E, that an example from the fruit data set is red and round.
  - $P(H|E)$ is the probability that E is an apple, givne that we have seen E is red and round. Known as the posteriori probability of H conditioned on E. This is based on more information than the prior probability, which is independent of E.
  - $P(E|H)$ is the posteriori probability of E conditioned on H - the probability that E is red and round given that we know that E is an apple.

Naive Bayes Algorithm

- The Bayes Theorem can be applied for classification tasks - Naive Bayes algorithm.
- While 1R makes decisions based on a single attribute, Naive Bayes uses all attributes and allows them to make contributions to the decision that are equally important and independent of one another.
- **Assumptions of Naive Bayes Algorithm**:
  - Independence assumption: the values of the attributes are conditionally independent of each other, given the class (for each class value).
  - Equal importance assumption: all attributes are equally important.
- These are unrealistic assumptions (hence why it is called Naive Bayes), but these assumptions lead to a simple method which works surprisingly well in practice.

Weather Example

- **Given**: weather data.

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

- **Task**: use Naive Bayes to predict the class (yes or no) of the new example.

outlook=sunny, temperature=cool,
humidity=high, windy=true

- H is play = yes, and also play = no i.e. there are two hypotheses.
- E Is the new example.

How Do We Use the Bayes Theorem for Classification?

- Calculate $P(H|E)$ for each H (class) i.e. $P(\text{yes}|E)$ and $P(\text{no}|E)$.
- Compare them and assign E to the class with the highest probability.
- To calculate the three parts of Bayes Theorem, we use the given data (this is the training phase of the classifier).

$$P(yes \mid E) = \frac{P(E \mid yes)P(yes)}{P(E)}$$
$$P(no \mid E) = \frac{P(E \mid no)P(no)}{P(E)}$$

where E
outlook=sunny,temperature=cool,
humidity=high, windy=true

1) How to calculate P(E|yes) and P(E|no) ?

Let's split the evidence E into 4 smaller pieces of evidence:

- **E1 = outlook=sunny, E2 = temperature=cool**
- **E3 = humidity=high, E4 = windy=true**

- Using the Naive Bayes independence assumption, $E1$ to $E4$ are independent given the class. Their combined probability is obtained by the multiplication of per-attribute probabilities. The final equation is thus:

$$P(\text{yes}|E) = \frac{P(E_1|\text{yes})P(E_2|\text{yes})P(E_3|\text{yes})P(E_4|\text{yes})P(\text{yes})}{P(E)}$$
$$P(\text{no}|E) = \frac{P(E_1|\text{no})P(E_2|\text{no})P(E_3|\text{no})P(E_4|\text{no})P(\text{no})}{P(E)}$$

- Numerator: the probabilities will be estimated from the data.
- Denominator: the two denominators are the same, and since we are comparing the two fractions, we **DO NOT** need to calculate $P(E)$.

E1 = outlook=sunny, E2 = temperature=cool
E3 = humidity=high, E4 = windy=true

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

	outlook		temperature			humidity			windy			play	
	yes	no		yes	no		yes	no		yes	no	yes	no
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								
sunny	2/9	3/5	hot	2/9	2/5	high	3/9	4/5	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	mild	4/9	2/5	normal	6/9	1/5	true	3/9	3/5		
rainy	3/9	2/5	cool	3/9	1/5								

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

proportions of days when play is yes

proportions of days when humidity is normal and play is yes i.e. the probability of humidity to be normal given that play=yes

$P(yes \mid E) = ?$ $P(yes \mid E) = \frac{P(E_1 \mid yes)\,P(E_2 \mid yes)\,P(E_3 \mid yes)\,P(E_4 \mid yes)\,P(yes)}{P(E)}$

	outlook		temperature			humidity			windy			play	
	yes	no		yes	no		yes	no		yes	no	yes	no
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								
sunny	2/9	3/5	hot	2/9	2/5	high	3/9	4/5	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	mild	4/9	2/5	normal	6/9	1/5	true	3/9	3/5		
rainy	3/9	2/5	cool	3/9	1/5								

⇒ **$P(E_1|yes)=P(outlook=sunny|yes)=2/9$**
$P(E_2|yes)=P(temperature=cool|yes)=3/9$
$P(E_3|yes)=P(humidity=high|yes)=3/9$
$P(E_4|yes)=P(windy=true|yes)=3/9$

- **P(yes) =? - the probability of a yes without knowing any E, i.e. anything about the particular day; the prior probability of yes; P(yes) = 9/14**

- By substituting the respective evidence probabilities, and then similarly calculating $P(\text{no}|E)$:

$$P(\text{yes}|E) = \frac{\frac{2}{9}\frac{3}{9}\frac{3}{9}\frac{3}{9}\frac{9}{14}}{P(E)} = \frac{0.0053}{P(E)}$$

$$P(\text{no}|E) = \frac{\frac{3}{5}\frac{1}{5}\frac{4}{5}\frac{3}{5}\frac{5}{14}}{P(E)} = \frac{0.0206}{P(E)}$$

$$P(\text{no}|E) > P(\text{yes}|E)$$

$$\text{Therefore, play} = \text{no is more likely than play} = \text{yes.}$$

Another Example

- Consider a volleyball game between team A and team B.
  - Team A has won 65% of the time and team B has won 35%.
  - Among the games won by team A, 30% were when playing on team B's court.
  - Among the games won by team B, 75% were when playing at home.
- If team B is hosting the next match, which team is most likely to win?

Solution

- Host: $\{A, B\}$
- Winner: $\{A, B\}$
- Using NB, the task is to compute and compare the two probabilities:

$$P(winner = A \mid host = B) = \frac{P(host = B \mid winner = A)P(winner = A)}{P(host = B)}$$

$$P(winner = B \mid host = B) = \frac{P(host = B \mid winner = B)P(winner = B)}{P(host = B)}$$

P(winner=A)= ? //probability that A wins =0.65

P(winner=B)=? //probability that B wins =0.35

P(host=B|winner=A)=? //probability that team B hosted the match, given that team A won =0.30

P(host=B|winner=B)=? //probability that team B hosted the match, given that team B won =0.75

$$P(winner = A \mid host = B) = \frac{P(host = B \mid winner = A)P(winner = A)}{P(host = B)} =$$

$$= \frac{0.3 * 0.65}{P(host = B)} = 0.195$$

$$P(winner = B \mid host = B) = \frac{P(host = B \mid winner = B)P(winner = B)}{P(host = B)} =$$

$$= \frac{0.75 * 0.35}{P(host = B)} = 0.2625$$

=>NB predicts team B

Three More Things About Naive Bayes

Problem: Probability Values of 0

- Suppose that the training data was different: `outlook=sunny` had always occured together with `play=no`.
- Then $P(\text{outlook}=\text{sunny}|\text{yes}) = 0$ and so:

$$P(yes \mid E) = \frac{\overbrace{P(E_1 \mid yes)}^{=0} P(E_2 \mid yes) P(E_3 \mid yes) P(E_4 \mid yes) P(yes)}{P(E)}$$

- Regardless of other probabilities, the final probability will always be 0.
- This is not good since the other probabilities are completely ignored due to the multiplication by 0. The prediction for `outlook=sunny` will always be no, regardless of the other attributes.

Solution

- Assume that our training data is so large that adding 1 to each count would not make a difference in calculating the probabilities, but will avoid the case of 0 probability.
- This is called the Laplace Correction/Estimator.

Example

- Add 1 to the numerator and k to the denominator, where k is the number of attribute values for the given attribute.
- Example:
  - A dataset with 2000 examples, 2 classes: `buy_Mercedes=yes` and `buy_Mercedes=no`; 1000 examples in each class.
  - One of the attributes is `income` with 3 values: `low`, `medium` and `high`.
  - For class `buy_Mercedes=yes`, there are 0 examples with `income=low`, 10 with `income=medium` and 990 with `income=high`.
- Probabilities without the Laplace correction for class `yes`: 0/1000=0, 10/1000=0.01, 990/1000=0.99.
- Probabilities with the Laplace correction: 1/1003=0.001, 11/1003=0.011, 991/1003=0.988.
- The correct probabilities are close to the adjusted probabilities, yet the 0 probability value is avoided!

Another Example

	yes	...
sunny	0	...
overcast	4	...
rainy	3	...
		...
sunny	0/9	...
overcast	4/9	...
rainy	3/9	...

P(sunny|yes)=0/9 →problem
P(overcast|yes)=4/9
P(rainy|yes)=3/9

Laplace correction

- **Assumes that there are 3 more examples from class *yes*, 1 for each value of *outlook***
- **This results in adding 1 to the numerator and 3 to the denominator of all probabilities**
- **Ensures that an attribute value which occurs 0 times will receive a nonzero (although small) probability**

$$P(sunny \mid yes) = \frac{0+1}{9+3} = \frac{1}{12}$$

$$P(overcast \mid yes) = \frac{4+1}{9+3} = \frac{5}{12}$$

$$P(rainy \mid yes) = \frac{3+1}{9+3} = \frac{4}{12}$$

Generalisation of the Laplace Correction: M-Estimate

- Add a small constant m to each denominator and mp_i to each numerator, where p_i is the prior probability of the i values of the attribute.

$$P(sunny \mid yes) = \frac{2 + mp_1}{9 + m}$$ $$P(overcast \mid yes) = \frac{4 + mp_2}{9 + m}$$ $$P(rainy \mid yes) = \frac{3 + mp_3}{9 + m}$$

- Note that $p_1 + p_2 + \ldots + p_m = 1$ where m is the number of attribute values.
- Advantage of using prior probabilities: it is rigorous.
- Disadvantage: computationally expensive to estimate prior probabilities.
- Large m: the prior probabilities are very important compared with the new evidence coming in from the training data; small m is less important.
- Typically, we assume that each attribute value is equally probable i.e. $p_1 = p_2 = \ldots = p_n = 1/n$. The Laplace correction is a special case of the m-estimate, where $p_1 = p_2 = \ldots = p_n = 1/n$ and $m = n$. Thus, 1 is added to the numerator and m to the denominator.

Handling Missing Values

- If there are missing attribute values in the new example, do not include this attribute.
  - **e.g. outlook=?, temperature=cool, humidity=high, windy=true**
  - **Then:** $$P(yes \mid E) = \frac{\frac{3}{9}\frac{3}{9}\frac{3}{9}\frac{9}{14}}{P(E)} = \frac{0.0238}{P(E)}$$ $$P(no \mid E) = \frac{\frac{1}{5}\frac{4}{5}\frac{3}{5}\frac{5}{14}}{P(E)} = \frac{0.0343}{P(E)}$$

  - **outlook is not included. Compare these results with the previous results!**
  - **As one of the fractions is missing, the probabilities are higher but the comparison is fair - there is a missing fraction in both cases**
- Missing attribute value in a training example - do not include this value in the counts. Calculate the probabilities based on the number of values that actually occur and not on the total number of training examples.

Handling Numeric Attributes

outlook			temperature		humidity		windy			play	
	yes	*no*	*yes*	*no*	*yes*	*no*		*yes*	*no*	*yes*	*no*
sunny	2	3	83	85	86	85	false	6	2	9	5
overcast	4	0	70	80	96	90	true	3	3		
rainy	3	2	68	65	80	70					
			64	72	65	95					
			69	71	70	91					
			75		80						
			75		70						
			72		90						
			81		75						
sunny	2/9	3/5	*mean*	73	74.6	*mean*	79.1	86.2	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	*std dev*	6.2	7.9	*std dev*	10.2	9.7	true	3/9	3/5		
rainy	3/9	2/5											

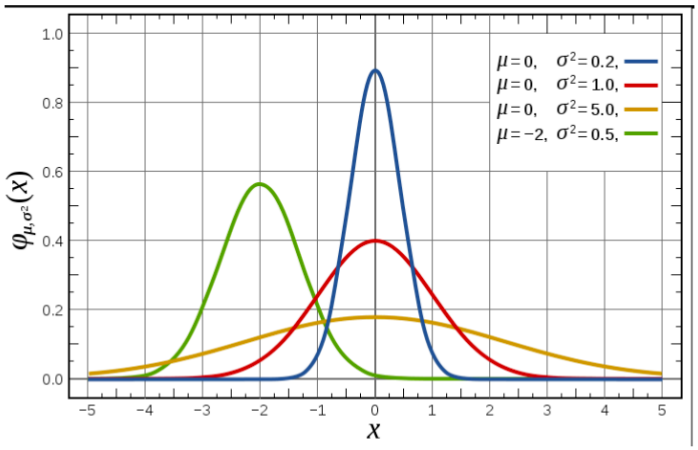
numeric

• We would like to classify the following new example:
outlook=sunny, temperature=66, humidity=90, windy=true

• Question: How to calculate
P(temperature=66|yes)=?, P(humidity=90|yes)=?
P(temperature=66|no)=?, P(humidity=90|no) ?

- Answer: by assuming the numerical values have a normal (Gaussian, bell curve) probability distribution and using the probability density function.
- For a normal distribution with mean μ and standard deviation σ, the probability function is:

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



- A probability density function of a continuous random variable is closely related to probability but not exactly the probability (e.g. the probability that x is exactly 66 is 0).
- The probability that a given value $x \in (x - \epsilon/2, x + \epsilon/2)$ is $\epsilon \times f(x)$.
- Given that 73 represents the mean for temperature for `class=yes`:

$$f(temperature = 66 \mid yes) = \frac{1}{6.2\sqrt{2\pi}} e^{-\frac{(66-73)^2}{2*6.2^2}} = 0.034$$

$$f(humidity = 90 \mid yes) = 0.0221$$

std.dev. for temp. for class=yes

$$P(yes \mid E) = \frac{\frac{2}{9}\ 0.034\ 0.0221\ \frac{3}{9}\ \frac{9}{14}}{P(E)} = \frac{0.000036}{P(E)}$$

=>P(no|E) > P(yes|E)

=> no play

$$P(no \mid E) = \frac{\frac{3}{5}\ 0.0291\ 0.038\ \frac{3}{5}\ \frac{5}{14}}{P(E)} = \frac{0.000136}{P(E)}$$

Advantages of NB

- Simple approach – the probabilities are easily computed due to the independence assumption.
- Clear semantics for representing, using and learning probabilistic knowledge.
- Excellent computational complexity:
  - Requires 1 scan of the training data to calculate all statistics (for both nominal and continuous attributes assuming normal distribution).
  - $O(pk)$, p - # training examples, k-valued attributes.
- In many cases outperforms more sophisticated learning methods - always try the simple method first!
- Robust to isolated noise points as such points are averaged when estimating the conditional probabilities from data.

Disadvantages of NB

- Correlated attributes reduce the power of Naive Bayes.
  - Violation of the independence assumption.
  - Solution: apply feature selection beforehand to identify and discard correlated (redundant) attributes.
- Normal distribution assumption for numeric attributes - many features are not normally distributed. Solutions:
  - Discretize the data first, i.e. numerical $\rightarrow$ nominal attributes.
  - Use other probability density functions, e.g. Poisson, binomial, gamma, etc.
  - Transform the attribute using a suitable transformation into a normally distributed one (sometimes possible).
  - Use kernel density estimation – doesn't assume any particular distribution.

Evaluating and Comparing Classifiers

- Our goal with classifiers are that they generalise well on new data i.e. they correctly classify new data, unseen during traing.

How to Evaluate the Performance of Classifiers

Accuracy and Error Rate

- **Accuracy**: proportion of correctly classified examples.
- **Error Rate**: complimentary to accuracy, the proportion of incorrectly classified examples.
- These two sum to 1, and since it is typically in percentage, thus sum to 100%.
- They are evaluated on the training and test set.
    - Accuracy on training data is overly optimistic, not a good indicator of performance on future data.
    - Accuracy on test data is the performance measure used.

Making the Most Out of Data

- Generally:
    - The larger the training data, the better the classifier.
    - The larger the test data, the better the accuracy estimate.
- Dilemma: ideally, we want to use as much data as possible for
    - Training to get a good classifier
    - Testing to get a good accuracy estimate
- Once the evaluation is completed, all the data can be used to build the final classifier.
    - Training, validation and test sets are joined together, a classifier is built using all of them for actual use by a customer.
    - The accuracy of the classifier must be quoted to the customer based on the test data.

Holdout Procedure

- Split data into two independent (non-overlapping) sets: training and test (usually 2/3 and 1/3).
- Use the training data to build the classifier.
- Use the test data is evaluate how good the classifier is by calculating performance measures such as accuracy.

outlook	temp.	humidity	windy	play
sunny	85	85	false	no
sunny	80	90	true	no
overcast	83	86	false	yes
rainy	70	96	false	yes
rainy	68	80	false	yes
rainy	65	70	true	no
overcast	64	65	true	yes
sunny	72	95	false	no
sunny	69	70	false	yes
rainy	75	80	false	yes
sunny	75	70	true	yes
overcast	73	90	true	yes
overcast	81	75	false	yes
rainy	71	91	true	no

training data: 9 examples (2/3)

test data: 5 examples (1/3)

Validation Set

- Sometimes we need a 3rd set: the validation set. Some classification methods including decisioin trees and neural networks operate in two stages:
  - Stage 1: build the classifier.
  - Stage 2: tune its parameters.
- The test data should not be used in any way to create the classifier, including parameter tuning.
- Proper procedure uses three non-overlapping data sets.
  - 1) Training set - to build the classifier
  - 2) Validation set - to tune the parameters
  - 3) Test set - to evaluate accuracy
- Examples:
  - DTs: training set used to build the tree, validation set for pruning, test set for performance.
  - NNs: validation set to stop the training and prevent overtraining.

Stratification

- An improvement of the holdout method.
- **Problem**: the hldout method reserves a certain amount for testing and uses the remainder for testing, however, the training set may not be representative of all classes (if all examples of a certain class is missing in the training set, the classifier cannot learn to predict this class).
- **Solution**: stratification ensures that each class is represented with approximately equal proportions in both data sets (training and test).
- Stratification is used together with the evaluation method e.g. holdout or cross validation.

Repeated Holdout Method

- Holdout can be made more reliable by repeating the process:
  - E.g. ten iterations, in each, a certain proportion (e.g. 2/3) is randomly selected for training (possibly with stratification) and the remainder is used for testing.
  - The accuracy on the different iterations are average to yield an overall accuracy.
- Can be improved by ensuring that the test sets do not overlap using cross validation.

Cross Validation

- S-fold cross validation:
  - Data is split into S subsets of equal size.
  - A classifier is built S times. Each time, the testing is on one segment and the training is on the remaining $S-1$ segments.
  - Average accuracies for each run to calculate the overall accuracy.
- The standard is 10-fold cross-validation.

Leave-One-Out Cross-Validation

- n-fold cross-validation, where n is the number of examples in the data set. We thus need to build the classifier ten times.
- **Advantages**:
  - The greatest possible amount of data is used for training, increasing the chance of building an accurate classifier.
  - Deterministic procedure - no random sampling is involved (no point in repeating the procedure since the same results are obtained).
- **Disadvantage**: High computational cost, although it is useful for small data sets.

How to Compare Classifiers

- Given two classifiers $C1$ and $C2$, we want to find out which one is better on a given task. Comparing the 10-fold CV estimates might give a general picture, but we aren't able to definitely say whether the difference is significant.
- A paired t-test can be used.

	C1	C2	
Fold 1	**95%**	**91%**	d_1=|95-91|=4
…			
Fold 2	**82%**	**85%**	d_2=|82-85|=3
==========			
mean	**91.3%**	**89.4%**	d_{mean}=3.5

$$\sigma = \sqrt{\frac{\sum_i^k (d_i - d_{mean})^2}{k-1}}$$

1. **Calculate the differences d_i**
2. **Calculate the standard deviation of the differences (an estimate of the true standard deviation)**
   If k is sufficiently large, d_i is normally distributed
3. **Calculate the confidence interval Z:**

$$Z = d_{mean} \pm t_{(1-\alpha)(k-1)} \frac{\sigma}{\sqrt{k}}$$

t is obtained from a probability table
1-α – confidence level
k-1 – degree of freedom

4. **Interval contains 0 – difference not significant, else significant**

Suppose that:

- **We use 10 fold CV => k=10**
- **d_{mean}=3.5; $\frac{\sigma}{\sqrt{k}} = 0.5$**
- **We are interested in significance at 95% confidence level**
- **Then:** $Z = 3.5 \pm 2.26 \times 0.5 = 3.5 \pm 1.13$

=> The interval does not contain 0 => the difference is statistically significant

	$(1-\alpha)$				
$k-1$	0.8	0.9	0.95	0.98	0.99
1	3.08	6.31	12.7	31.8	63.7
2	1.89	2.92	4.30	6.96	9.92
4	1.53	2.13	2.78	3.75	4.60
9	1.38	1.83	2.26	2.82	3.25
14	1.34	1.76	2.14	2.62	2.98
19	1.33	1.73	2.09	2.54	2.86
24	1.32	1.71	2.06	2.49	2.80
29	1.31	1.70	2.04	2.46	2.76

Confusion Matrix

2 class prediction (classes *yes* and *no*) – 4 different outcomes

Confusion matrix:

examples	# assigned to class yes	# assigned to class no
# from class yes	true positives (tp)	false negatives (fn)
# from class no	false positives (fp)	true negatives (tn)

- The accuracy is thus:

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

- Information retrieval (IR) also uses recall (R), precision (P), and their combination, F1 measure (F1) as performance measures.

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

$$\text{F1} = \frac{2 \times P \times R}{P + R}$$

- Typically we can maximise one of recall and precision but not both.

Extreme example 1: all e-mails are classified as spam and blocked

email	# classified as spam	# classified as not spam
# spam	25 (tp)	0 (fn)
# not spam	75 (fp)	0 (tn)

Spam precision = 25%, Spam recall =100%, Accuracy = 25%

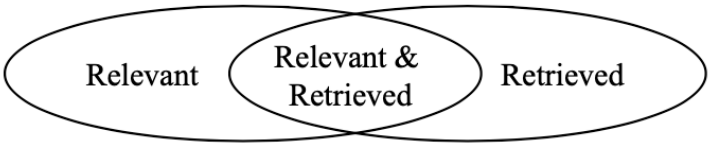
Extreme example 2: all but one e-mail are classified as not-spam

email	# classified as spam	# classified as not spam
# spam	1 (tp)	79 (fn)
# not spam	0 (fp)	20 (tn)

Spam precision = 100%, Spam recall =12.5%, Accuracy = 21%

IR Example 2: Text Retrieval

- Given a query, a text retrieval system retrieves a number of documents.
  - Retrieved - the number of all retrieved documents.
  - Relevant - the number of all documents that are relevant.



- Recall, Precision and F1 are used to address the accuracy of the retrieval.

$$\text{Precision} = \frac{\text{Relevant and Retrieved}}{\text{Retrieved}}$$
$$\text{Recall} = \frac{\text{Relevant and Retrieved}}{\text{Relevant}}$$

Cost-Sensitive Evaluation

- Misclassification may have a different cost. The cost of misclassifying a legitimate email as spam is greater than letting a spam email slip through.
- To reflect the different costs, we can calculate weighed (adjusted) accuracy, precision and F1: blocking a legitimate email counts as X errors (e.g. 10, 100 etc. depending on the application).
- Other examples where the cost of different errors is different:
  - Loan approval: lending to a defaulter > refusing a loan to a non-defaulter.
  - Oil spill detection: failing to detect an oil spill > false alarms.

Inductive Learning (Why We Need Empirical Evaluation)

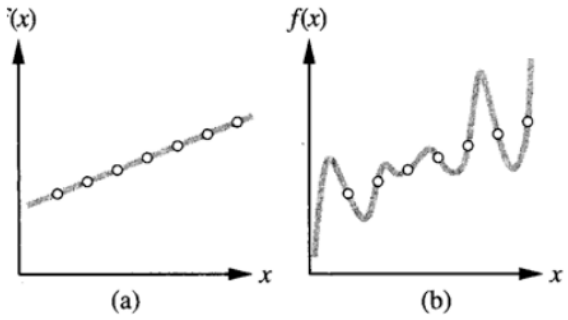
- We need to do empirical evaluation since supervised learning is inductive learning i.e. inducing the universal from the particular e.g. All apples I have seen are red $\rightarrow$ All apples are red.
- **Given**: a set of examples $(x, f(x))$.
  - x is the input vector.
  - $f(x)$ is the output of a function f applied to x.
  - We don't know what $f(x)$ is.
- **Find**: A function h (hypothesis) that is a good approximation of f.

Difficulty with Finding Hypotheses

- We may generate many hypotheses h. The set of all possible hypotheses h form the hypothesis space H.
- We can tell if a particular h is a good approximation of f if it generalises well (correctly predicts new examples). This is why we use a test set to measure performance.
- A good h does not necessarily imply fitting the given samples x perfectly - we want to extract patterns from the data, not memorise it.

Which Consistent Hypothesis is the Best?

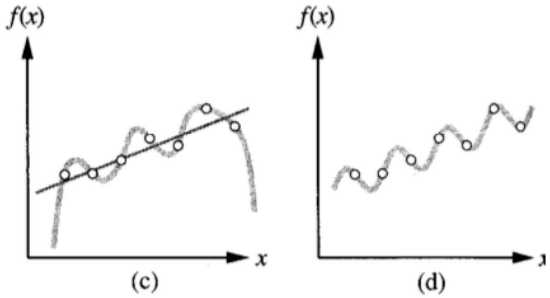
- **Given:** a set of 7 examples $(x, f(x))$, where x and $f(x)$ are real numbers.
- **Task**: find $h \approx f$ such that:
  - h is a function a single variable x.
  - the hypothesis space is the set of polynomials of at most k e.g. $6x^2 + 3x + 1$ is of degree 2.
- Consider these two solutions:
  - Both hypotheses are consistent with the training data (agree with it, and fit the examples perfectly).
  - We'd say that the fit with a line (degree 1) is better than a fit with a degree 7 polynomial.



Multiple Consistent Hypotheses

- Recall that we'd like to **extract pattern** from the data.
- Ockham's Razor: prefer the simplest hypothesis consistent with the training data. It is also simpler to compute.
- Defining simplicity is not always easy in general (it is in the above case).

Importance of the Hypothesis Space



- d shows the true function: $sin(x)$.
- In c, we are searching the hypothesis space h consisting of polynomial functions, which $sin(x)$ is not

part of.
- Only a polynomial with degree 6 perfectly fits the data (the straight line doesn't), but this is not a good choice.
- **Thus**: the choice of hypothesis space is important. The sin function cannot be learnt accurately using polynomials.
- A learning problem is realisable if H contains the true function. In practice, we can't tell if the problem is realisable because we don't know the true hypothesis (we are trying to learn it from examples).

Lecture 7 - Decision Trees

- DT's are supervised classifiers. They were developed in parallel in ML by Ross Quinlan (USYD) and in statistics by Breiman, Friendman, Olshen and Stone.
    - Quinlan has refined the DT algorithm over the years - ID3 in 1986, C4.5 in 1993 (commercial version is used in many DM packages).
- There are 2 DT versions in WEKA.
    - id3 - nominal attributes only
    - J48 - both nominal and numeric

Decision Tree Summary

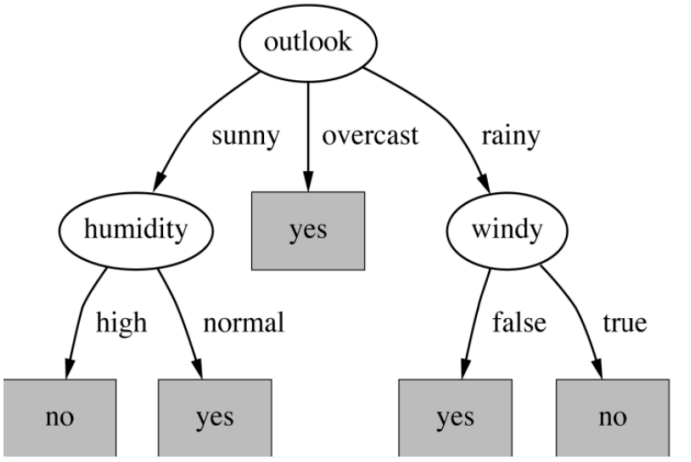
Components of DT

- Model (structure): a non pre-specified tree derived from data.
- Search method (how the data is searched by the algorithm to build the structure):
    - Two phases: grow and prune.
    - Growing the tree: hill climbing search guided by information gain, uses the training data set.
    - Pruning the tree: various approaches such as sub-tree replacement with validation set to determine how much to prune (performs a greedy-esque search to choose the best sub-tree to be replaced with a leaf node).

Properties of DTs

- Easy to implement. Efficient:
    - Cost of building the tree $O(mn \log n)$, n instances and m attributes.
    - Cost of pruning the the tree with sub-tree replacement: $O(n)$.
    - Cost of pruning by sub-tree lifting: $O(n\ (\log n)^2)$.
    - Total cost: $O(mn\ \log n) + O(n\ (\log n)^2)$.
- Interpretable:
    - The output of the DT (tree = a set of rules) is considered easier to understand by humans and use for decision making than the output of other algorithms e.g. neural network, support vector machines.

Decision Tree Specifics



- **DT representation**:
  - each non-leaf node represents an attribute.
  - each branch corresponds to an attribute value.
  - each leaf node assigns a class.
- To predict the class for a new example: start from the root and test the values of the attributes until you reach a leaf node, return the class of the leaf node.
- DTs thus can express any function of the input attributes - there is a consistent DT for any training set with 1 path to a leaf i.e. lookup table, explicitly encoding the training data.
  - This DT will no generalise well to new examples. We prefer to find more compact decision trees.
- DTs can be expressed as a set of mutually exclusive rules (each rule is a conjunction of tests with 1 rule = 1 path in the tree). DTs are thus a disjunction of conjunctions.

Constructing Decision Trees

- Top down in a recursive divide and conquer fashion:
  - The best attribute is selected for a root node and a branch is created for each possible attribute value.
  - The examples are split into subsets, one for each branch extending from the node.
  - The procedure is repeated recursively for each branch, using only the examples that reach the branch.
  - Stop if all examples have the same class; make a leaf node corresponding to this class.

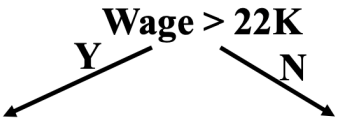
Stopping Criteria

- Stop if:
  - The typical case: **all examples in the subset following the branch have the same class** $\rightarrow$ make a leaf node corresponding to this class.
  - Additional condition: **cannot split any further** - all examples in the subset have the same attribute values but different classes (there's noise in the data) $\rightarrow$ make a leaf node and label it with the majority class of the subset.
  - Additional condition: **subset is empty** (e.g. there are no examples with the attribute value for the branch) $\rightarrow$ create a leaf node and label it with the majority class of the subset of the parent.

Example

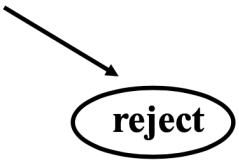
- Assume that we know how to select the best attribute at each step.

name	wage	dependents	sex	loan
Jim	40K	7	M	reject
Jill	20K	1	F	reject
Jack	58K	5	M	accept
Jane	23K	3	F	accept



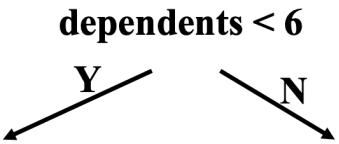
name	wage	dependents	sex	loan
Jim	40K	7	M	reject
Jack	58K	5	M	accept
Jane	23K	3	F	accept

name	wage	dependents	sex	loan
Jill	20K	1	F	reject



- From here, we look at the leftward node.

name	wage	dependents	sex	loan
Jim	40K	7	M	reject
Jack	58K	5	M	accept
Jane	23K	3	F	accept

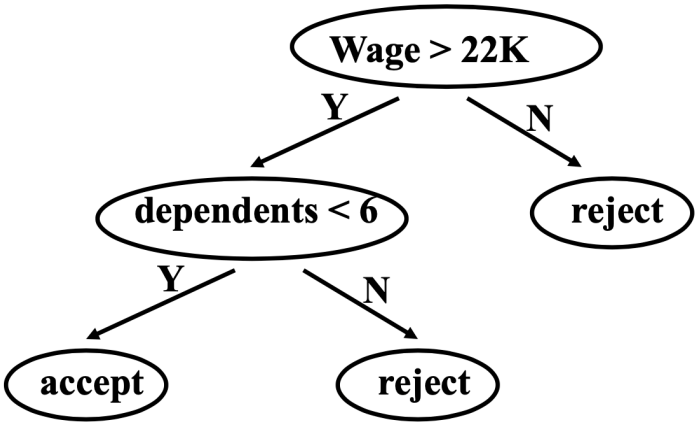


name	wage	dependents	sex	loan
Jack	58K	5	M	accept
Jane	23K	3	F	accept

name	wage	dependents	sex	loan
Jim	40K	7	M	reject

- Finally:

name	wage	dependents	sex	loan
Jim	40K	7	M	reject
Jill	20K	1	F	reject
Jack	58K	5	M	accept
Jane	23K	3	F	accept



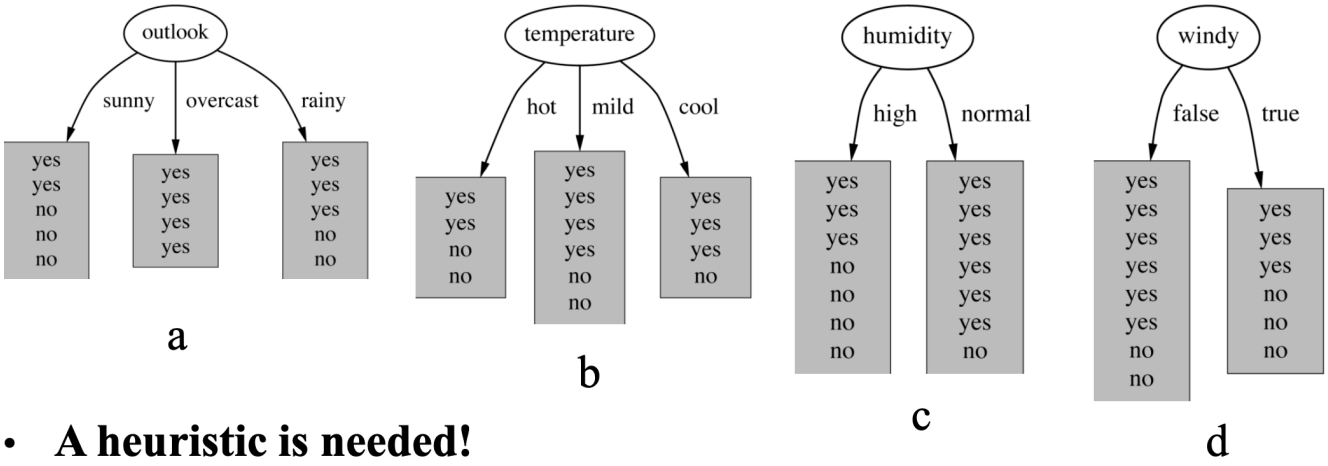
Pseudo-Code

- Aim: find a tree consistent with the training examples.
- Idea: recursively choose the best attribute as root of sub-tree.

Majority class of the sub-set of the parent node

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default  //case c)
    else if all examples have the same classification then return the classification  //case a)
    else if attributes is empty then return MODE(examples)  //case b)
    else                                  Majority class of examples
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value v_i of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← DTL(examples_i, attributes − best, MODE(examples))
            add a branch to tree with label v_i and subtree subtree
        return tree
```

How to Find the Best Attribute?



- **A heuristic is needed!**

- A measure of 'purity' of each node would be a good choice, as the 'pure' subsets (containing only yes or no) will not have to be split further and the recursive process will terminate.
    - At each step, we can choose the attribute which produces the purest children nodes.

Entropy

- Given a set of examples with their class e.g. the weather data with 9 examples from class yes and 5 examples from class no.

Information Content - Interpretation 1

- Entropy measures the homogeneity (purity) of a set of examples with respect to their class.
- The smaller the entropy, the greater the purity of the set. It is the standard measure in signal compression, information theory, physics.

Formal Definition

- Entropy $H(S)$ of data set S, where P_i is the proportion of examples that belong to class i.

$$H(S) = I(S) = -\sum_i P_i \times log_2 P_i$$

- For our example:

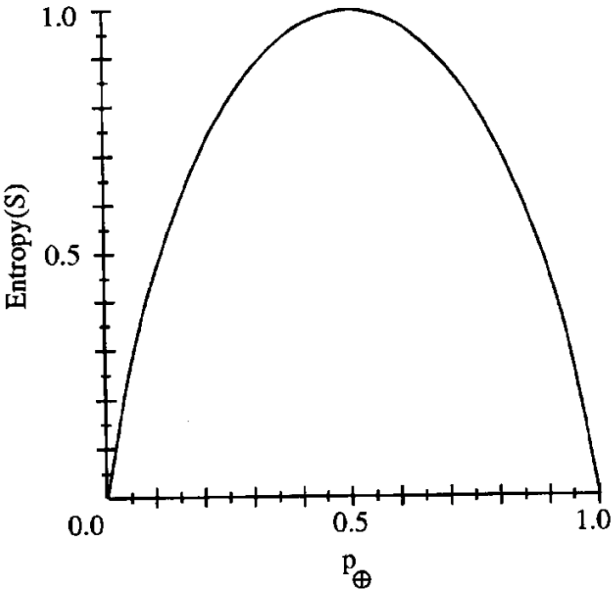
$$H(S) = -P_{yes} \log_2 P_{yes} - P_{no} \log_2 P_{no} = I(P_{yes}, P_{no}) = I(\frac{9}{14}, \frac{5}{14}) = -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} = 0.940\ bits$$

- Entropy is measured in bits so we used log to the base 2. Note: $log(0)$ is undefined, so we just assume that it is 0 when we calculate entropy.

Range of Entropy for Binary Classification

- 2 classes: yes and no. On the x-axis is the proportion of positive examples (with the proportion of negative examples being 1-p). On the y-axis is the entropy $H(S)$.

$$H(S) = I(p, 1-p) = -p \log_2 p - (1-p) \log_2 (1-p)$$



- $H(S) \in [0, 1]$.
- $H(S) = 0$ means that all examples in S belong to the same class (no disorder, min entropy).
- $H(S) = 1$ means that there are an equal number of yes and no (S is as disordered as it can be, max entropy).

Information Theory - Interpretation 2

- Based on a Sender and a Receiver.
- Sender sends information to Receiver about the outcome of an event (e.g. flipping a coin, 2 possibilities: heads or tails).
- Receiver has some prior knowledge about the outcome, e.g.
  - Case 1: knows that the coin is honest
  - Case 2: knows that the coin is rigged so that it comes heads 75% of the time
- An answer telling the outcome of a toss will be less predictable for the Receiver with Case 1 - it is thus more informative to them about the nature of the dice.
  - Entropy represents the amount of surprise of the receiver by the answer based on the probability of the answers.

Formal Definition

- Given a set of possible answers (messages) $M = \{m_1, m_2, \ldots, m_n\}$ and a probability $P(m_i)$ for the occurrence of each answer, the expected information content (entropy) of the actual answer is:

$$H(M) = -\sum_{i}^{n} P(m_i).\log_2 P(m_i) = entropy(M)$$

Example 1: Flipping coin

- **Case 1 (honest coin):**

H(coin_toss)= I(1/2, 1/2)= -(1/2)log(1/2) -(1/2)log(1/2) = 1 bit

- **Case 2 (rigged coin):**

H(coin_toss) = I(1/4, 3/4)= -(1/4)log(1/4) -(3/4)log(3/4) = 0.811 bits

Min Number of Bits - Interpretation 3

Example

- Suppose that X is a random variable with 4 possible values A, B, C and D;
  $P(X = A) = P(X = B) = P(X = C) = P(X = D) = 1/4$.
- You must transmit a sequence of values of X over a serial binary channel. How?
  - Solution: encode each symbol with 2 bits, e.g. A=00, B=01, C=10, D=11.
- Now you are told that the probabilities are not equal, e.g.
  $P(X = A) = 1/2, P(X = B) = 1/4, P(X = C) = P(X = D) = 1/8$.
  - A better encoding can now be made with average 1.75 bits i.e. A=0, B=10, C=110, D=111
  - Average number of bits per symbol = $1/2(1) + 1/4(2) + 1/8(3) + 1/8(3) = 1 + 3/4 = 1.75$ bits.

General Version

- Suppose that X is a random variable with m possible values $V_1 \dots V_m$;
  $P(X = V_1) = P_1, P(X = V_2) = P_2, \dots, P(X = V_m) = P_m$.
- The smallest possible number of bits per symbol on average needed to transmit a stream of symbols drawn from X's distribution is:

$$H(X) = -\sum_{i=1}^{m} P_i \times \log_2 P_i$$

- **High entropy** – the values of X are all over place.
  - The histogram of the frequency distribution of values of X will be flat.
- **Low entropy** – the values of X are more predictable.
  - The histogram of the frequency distribution of values of X have many lows and one or two highs.

Information Gain

- On the topic of DTs, we have three definitions of entropy measures.
  - The disorder of a set of training examples with respect to their class Y.
  - Shows the amount of surprise of the receiver by the answer Y based on the probability of the answers.
  - The smallest possible number of bits per symbol (on average) needed to transmit a stream of symbols drawn from Y's distribution.
- We use the first to define the information gain: measure of the effectiveness of an attribute to classify the training data.
  - It measures the reduction in entropy caused by using this attribute to partition the set of training

examples.

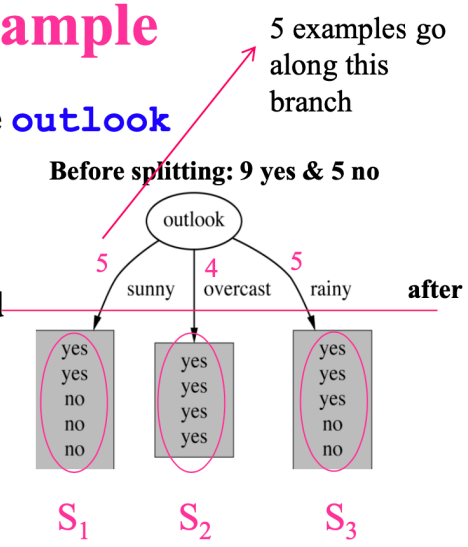
- The best attribute is the one with the highest information gain i.e. largest reduction in the entropy of the parent node, and the one that is expected to lead to pure partitions fastest.

Example

Information Gain - Example

- **Let's calculate the information gain of the attribute `outlook`**
- **Information gain measures reduction in entropy**
- **It is a difference of 2 terms: *T1 –T2***
- ***T1* is the entropy of the set of examples S associated with the parent node before the split**

$$T1 = H(S) = I(\frac{9}{14}, \frac{5}{14}) = 0.940\ bits$$



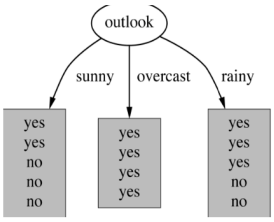
- ***T2* is the remaining entropy in S, after S is split by the attribute (e.g. `outlook`)**
  - **It takes into consideration the entropies of the child nodes and the distribution of the examples along each child node**
  - **E.g. for a split on `outlook`, it will consider the entropies of S1, S2 and S3 and the proportion of examples following each branch (5/14, 4/14, 5/15):**

$$T2 = H(S \mid outlook) = \frac{5}{14} \cdot H(S_1) + \frac{4}{14} \cdot H(S_2) + \frac{5}{14} \cdot H(S_3)$$

- Therefore:

$$Gain(S \mid A) = H(S) - \sum_{j \in values(A)} P(A = v_j) H(S \mid A = v_j) =$$

$$= H(S) - \sum_{j \in values(A)} \frac{|S_v|}{|S|} H(S \mid A = v_j)$$



Gain(S|A) is the information gain of an attribute A relative to S

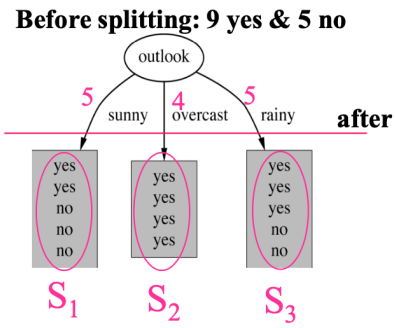
Values(A) is the set of all possible values for A

S_v is the subset of S for which A has value v

=entropy of the original data set S

=conditional entropy H(Y|A);
=expected value of the entropy after S is partitioned by A
=called *Reminder* in Russell and Norvig

- Calculating the entropy of the child:



$$H(S \mid outlook = sunny) = I(\frac{2}{5}, \frac{3}{5}) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.971 bits$$

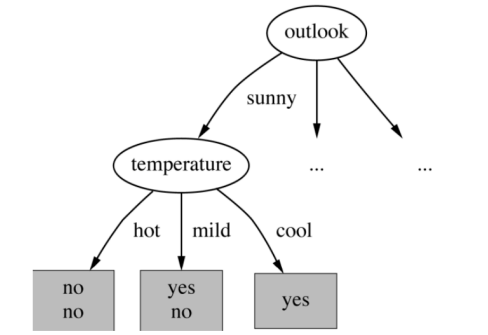
$$H(S \mid outlook = overcast) = I(\frac{4}{4}, \frac{0}{4}) = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4} = 0 bits$$

$$H(S \mid outlook = rainy) = I(\frac{3}{5}, \frac{2}{5}) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.971 bits$$

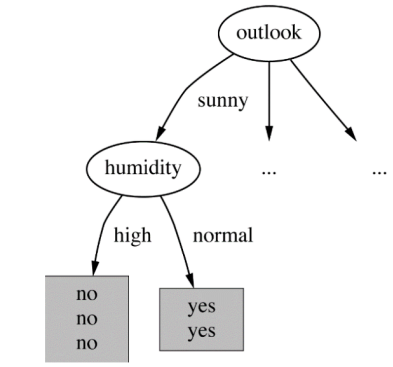
$$H(S \mid outlook) = \frac{5}{14} \cdot 0.971 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.971 = 0.693 bits$$

- The $Gain(S|outlook) = 0.247$ bits. The other three attributes are temperature $= 0.029$, humidity $= 0.152$, windy $= 0.048$.

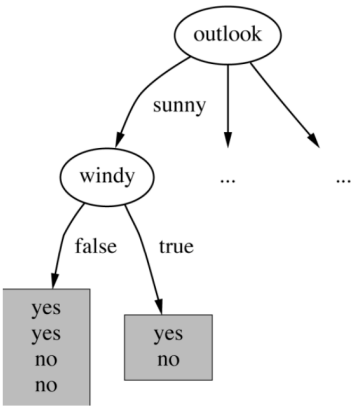
Continuing the Split From Here On



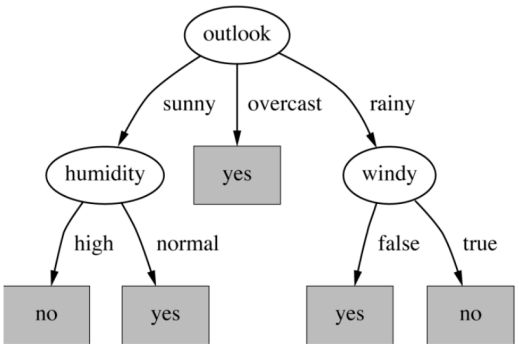
Gain(S, temperature)=0.571 bits



Gain(S, humidity)=0.971 bits

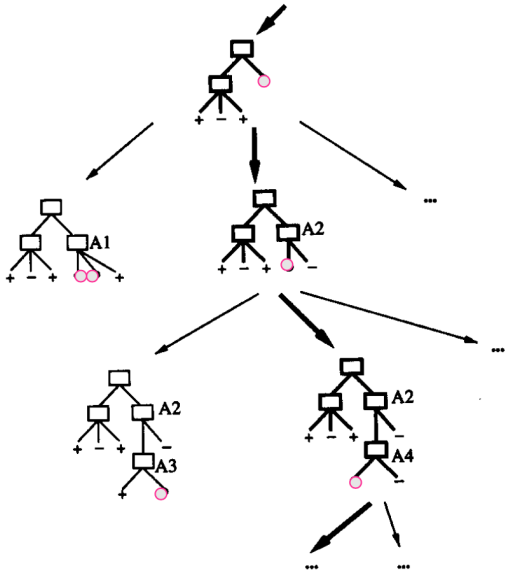


Gain(S, windy)=0.020 bits



Final DT

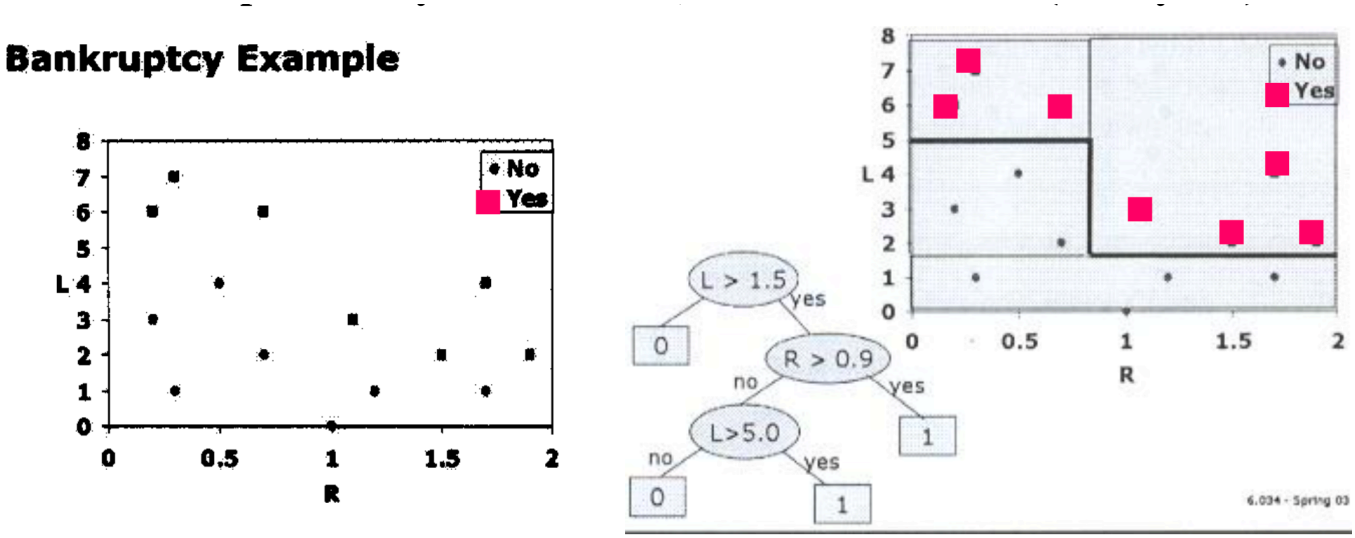
Building DT as a Search Problem



- DT algorithm searches the hypothesis space for a hypothesis that fits (correctly classifies) the training data. It uses the following search strategy:
  - Simple to complex search (starting with an empty tree and progressively considering more elaborate hypotheses).
  - Hill climbing with information gain as an evaluation function.
- Information gain is an evaluation function of how good the current state is (how close to the goal state, a tree that classifies all training examples correctly).

DT Decision Boundary

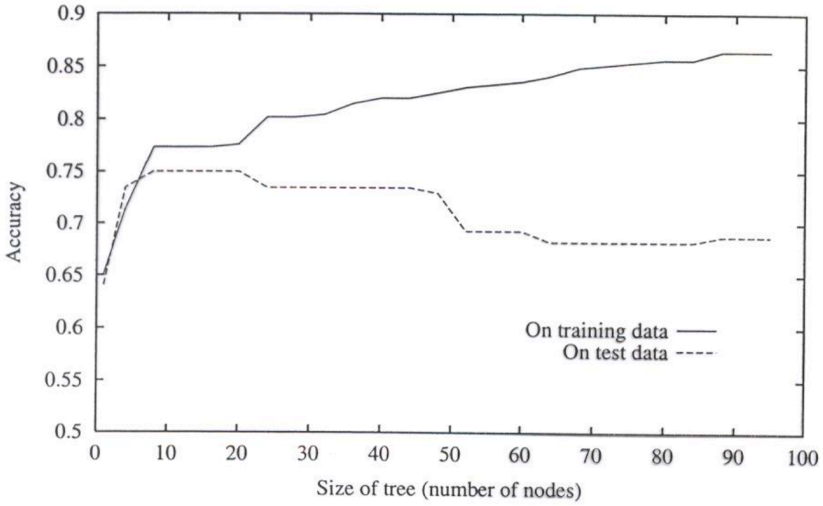
- DTs define a decision boundary in the feature space. Example: binary classification, numeric attributes (binary DT).



Overfitting

- The error on the training data is very small but the error on new data (test data) is high.
  - The classifier has memorized the training examples but has not extracted pattern from them and is not classifying well the new examples!
- The more formal defintion of overfitting at a generic level and not only for DTs is:

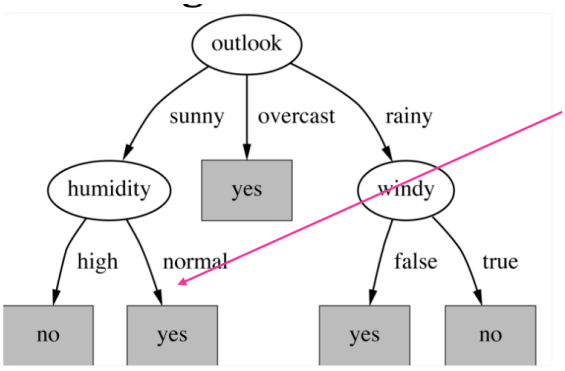
- Given H - a hypothesis space, $h \in H$ - a hypothesis, D - entire distribution of instances, $train$ - training instances.
- h is said to overfit the training data if there exists some alternative hypothesis $h' \in H$ such that $\text{error}_{\text{train}} h < \text{error}_{\text{train}} h'$ and $\text{error}_D h > \text{error}_D h'$.



Overfitting in DTs - Reasons

- A DT grows each branch of the tree deeply enough to perfectly classify the training examples.
- In doing this (and depending on the data), the tree may become very specific (like a look-up table) and not represent patterns, but only memorize the data.
- Problems with the training data also leads to problems with the induced DT
  - Training data is too small $\rightarrow$ not enough representative examples to build a model that can generalize well on new data
  - Noise in the training data, e.g. incorrectly classified examples $\rightarrow$ DT learns them but they are incorrect and will incorrectly classify new examples.

Overfitting Due to Noise



- Let's say that the following positive example is incorrectly labelled as negative: `outlook=sunny`, `temperature=hot`, `humidity=normal`, `wind=yes`, `play=no`.
- The new tree will test below the highlighted node, becoming more complex. We still expect the original tree to outperform the new tree on the subsequent data.

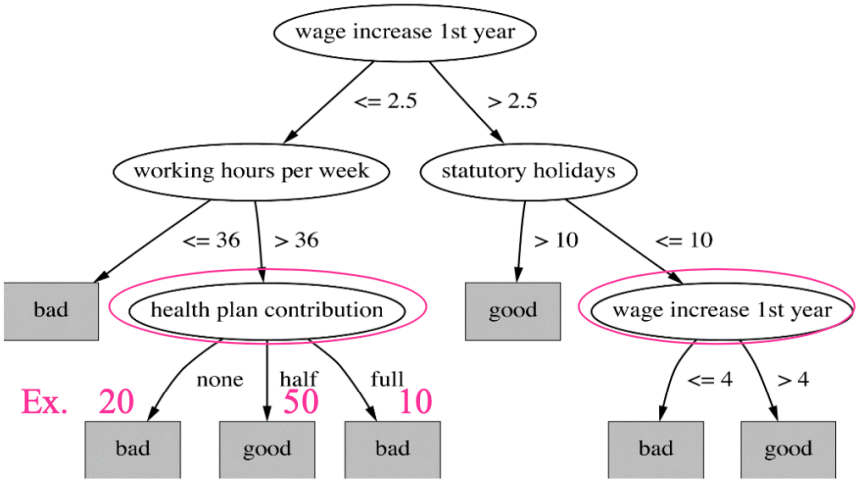
Tree Pruning

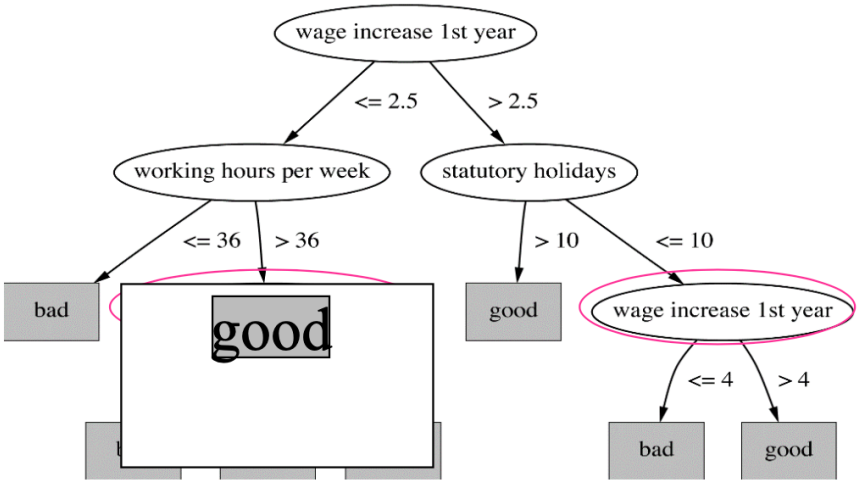
- Used to avoid overfitting in DTs.
  - Pre-pruning: stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data.
  - Post-pruning: fully grow the tree (allowing it to perfectly over the training data) and then prune it. Since this is the more successful and widely used method, there are two main approaches.
    - Tree-pruning - directly prune the tree.
      - Sub-tree replacement
      - Sub-tree raising
    - Rule pruning - convert the three into a set of rules and then prune them.

Using Validation Sets

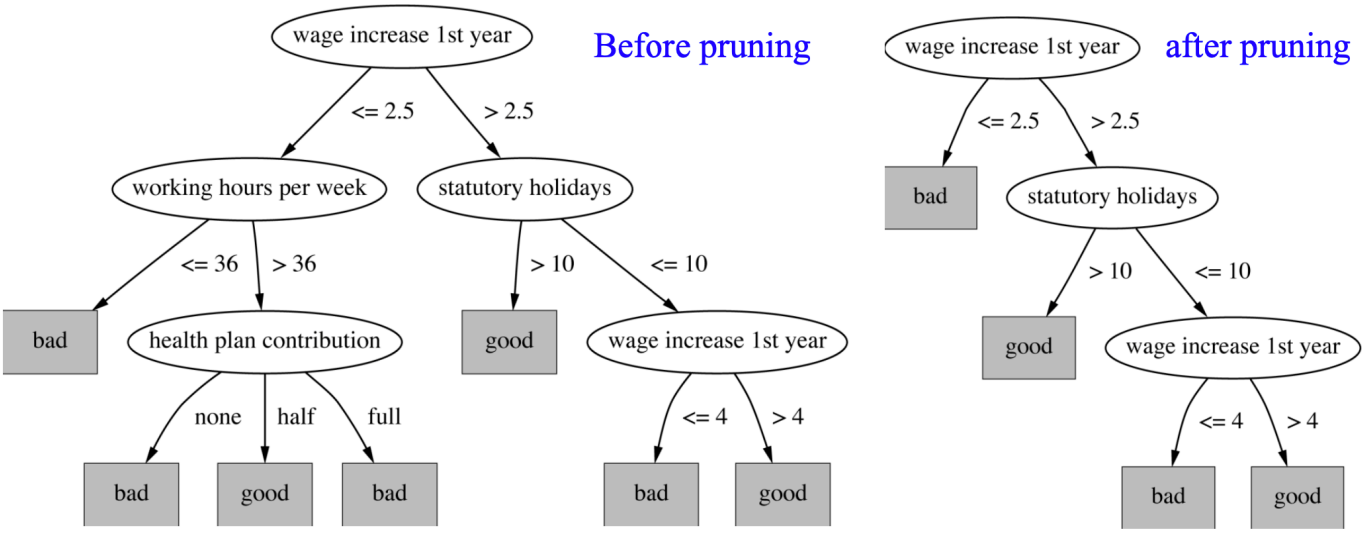
- We determine when to stop pruning by estimating the accuracy using validation sets. We could use the training data, but this is too optimistic (statistical underpinning of heuristic is rather weak).
- Available data is separated into three sets:
  - Training set - used to build the DT.
  - Validation set - to evaluate the impact of pruning and decide when to stop.
  - Test data - to evaluate how good the final DT is.
- **Motivation**:
  - Even though the DT may be misled by random errors and coincidental regularities within the training set, the validation set is unlikely to exhibit the same random fluctuations $\rightarrow$ the validation set can provide a safety check against overfitting of the training set.
  - The validation set should be large enough; typically 1/2 of the available examples are used as training set, 1/4 as validation set, and 1/4 as test set.
- **Disadvantage:** The tree is built on less data.
  - When the data is limited, withholding part of it for validation reduces even further the examples available for training.

Tree Post-Pruning by Sub-Tree Replacement

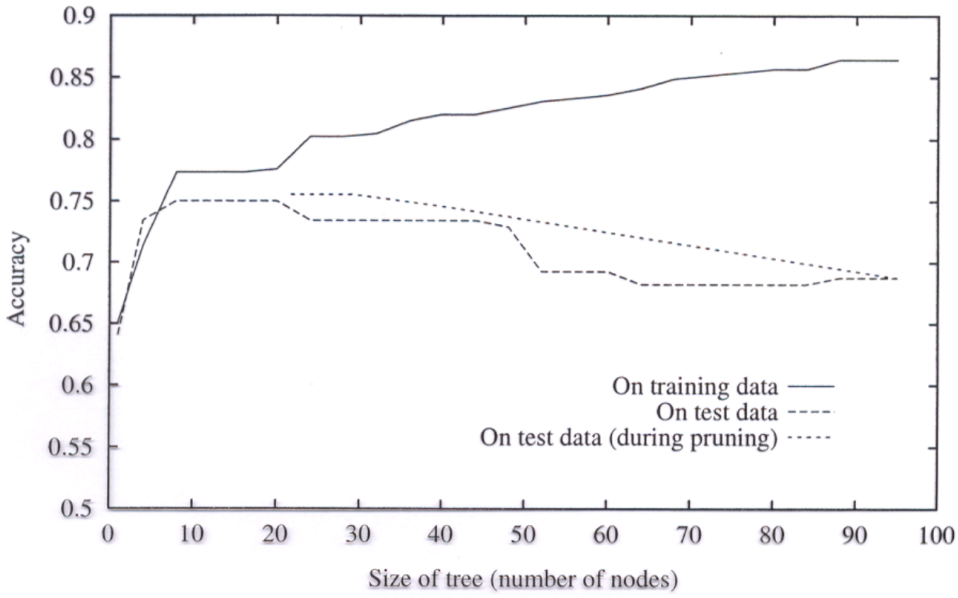




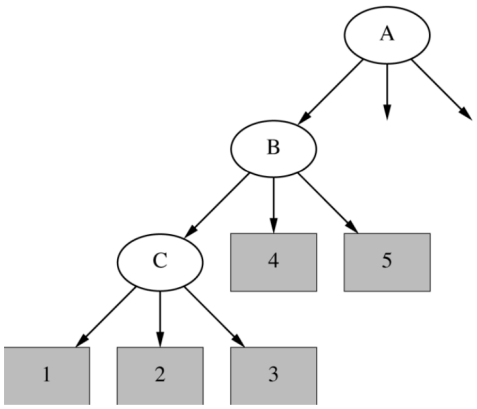
- Each non-leaf node (i.e. each test node) is a candidate for pruning.
- Start from the leaves and work towards the root.
- For each candidate node:
    - Remove the sub-tree rooted at it.
    - Replace it with a leaf with the class being the majority class of examples that go along the candidate node.
    - Compare the new tree with the old tree by calculating the accuracy on the validation set for both.
    - If the accuracy of the new tree is better or the same as the accuracy of the old tree, keep the new tree (we say that the candidate node is pruned).
- At each step, we will evaluate all candidate nodes and accept the best pruning – i.e. the one that will improve the accuracy most. No pruning if the new tree is worse than the old tree.

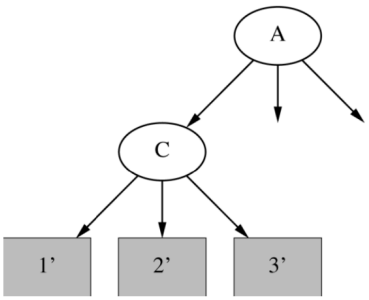


- The accuracy of test data increases as nodes are pruned.



Post-Pruning: Sub-Tree Raising



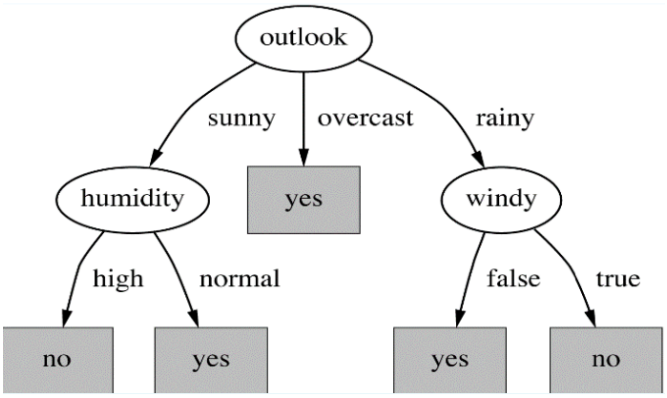


Node C is raised to subsume node B

examples at nodes 4 , 5 are re-classified into the new sub-tree headed by C => 1', 2', 3'

- Sub-tree raising is potentially time consuming. It is restricted to raising the sub-tree of the most popular branch i.e. raise C only if the branch from B to C has more examples than the branches from B to 4 or B to 5.
    - If another branch is more popular, raise that one instead.

Rule Post-Pruning



- Grow the tree until it fits the training data.
- Convert into an equivalent set of rules by creating 1 rule for each path from the root to a leaf e.g. `outlook=sunny`, and `humidity=high` then `play=no` is rule 1 of 5 for the above tree
- Prune each rule by removing the pre-conditions that are not harmful i.e. the estimate rule accuracy is the same or higher (accuracy on validation set).
- Sort the pruned rules by their estimated accuracy and consider them in this sequence when classifying subsequent instances.

Why Convert DT to Rules Before Pruning?

- Bigger flexibility
  - When trees are pruned, there are only 2 choices - to remove the node completely or retain it.
  - When rules are pruned, there are less restrictions:
    - Pre-conditions are removed rather than nodes.
    - Each branch in the tree is treated separately.
    - Removes the distinction between attribute tests that occur near the root of the tree and those near the leaves.
- It is also easier to read a set of rules than a tree.

Numeric Attributes

- Numerical attributes need to be converted into nominal - this is called discretisation. In DTs, we restrict the possibilities for a numerical attribute to a binary split.
- Procedure:
  - Sort the examples according the values of the numerical attribute.
  - Identify adjacent examples that differ in their class and generate a set of candidate splits (split points are placed halfway).
  - Evaluate Gain (or other measure) for every possible split point and choose the best split point.
  - Gain for best split point is Gain for the attribute.

- **values of `temperature`:**

64	65	68	69	70	71	72	73	74	75	80	81	83	85
yes	no	yes	yes	yes	no	no	no	yes	yes	no	yes	yes	no

- **7 possible splits; consider split between 70 and 71**
- **Information gain for** **- 1) temperature < 70.5 : 4 yes & 1 no**
  - 2) temperature =>70.5 : 4 yes & 5 no

$$H(S) = -\frac{8}{14}\log_2\frac{8}{14} - \frac{6}{14}\log_2\frac{6}{14} = 0.985\,bits$$

$$H(S_{temp<70.5}) = -\frac{4}{5}\log_2\frac{4}{5} - \frac{1}{5}\log_2\frac{1}{5} = 0.722\,bits$$

$$H(S_{temp=>70.5}) = -\frac{4}{9}\log_2\frac{4}{9} - \frac{5}{9}\log_2\frac{5}{9} = 0.991\,bits$$

$$H(S\,|\,temp70.5) = \frac{5}{14}0.722 + \frac{9}{14}0.991 = 0.895bits$$

$$Gain(S\,|\,temp70.5) = 0.985 - 0.895 = 0.09bits$$

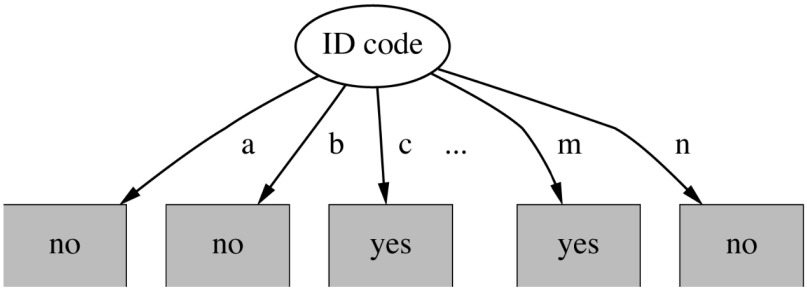
$$Gain(S\,|\,A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v)$$

Alternative Measures for Selecting Attributes

- **Problem**: if an attribute is highly branching (with a large number of values), information gain will select it.
- **Reason**: highly branching attributes are more likely to create pure subsets with low entropy and high information gain (imagine the information gain of a lookup list).
- This leads to overfitting.

Weather data with ID code

ID code	outlook	temp.	humidity	windy	play
a	sunny	hot	high	false	no
b	sunny	hot	high	true	no
c	overcast	hot	high	false	yes
d	rainy	mild	high	false	yes
e	rainy	cool	normal	false	yes
f	rainy	cool	normal	true	no
g	overcast	cool	normal	true	yes
h	sunny	mild	high	false	no
i	sunny	cool	normal	false	yes
j	rainy	mild	normal	false	yes
k	sunny	mild	normal	true	yes
l	overcast	mild	high	true	yes
m	overcast	hot	normal	false	yes
n	rainy	mild	high	true	no



Tree stump for the ID code attribute

- **split based on IDcode:**

$$H(S_a) = -\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1} = 0\,bits$$

...

$$H(S_m) = -\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1} = 0\,bits$$

- **the weighted sum of entropies:**

$$H(S \mid IDcode) = \frac{1}{14}\cdot 0 + \ldots + \frac{1}{14}\cdot 0 = 0 bits$$

- **entropy at the root**

$$H(S) = -P_{yes}\log_2 P_{yes} - P_{no}\log_2 P_{no} = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940\ bits$$

- **Gain**

$$Gain(S, Idcode) = H(S) - \sum_{v \in Values(Idcode)} \frac{|S_v|}{|S|} H(S_v) = 0.940 bits$$

Gain Ratio

- Gain Ratio: a modification of Gain that reduces its bias towards highly branching attributes.

$$GainRatio(S \mid A) = \frac{Gain(S \mid A)}{SplitInformation(S \mid A)}$$

- It takes the number and size of branches into account when choosing an attribute and penalises highly-branching attributes by incorporation SplitInformation, which is the entropy of S with respect to A.

$$splitInformation(S \mid A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

For the Above Example

Computing Gain ratio for IDcode:

$$SplitInformation(S \mid Idcode) = -\frac{1}{14} \log_2 \frac{1}{14} * 14 = 3.807 bits$$

$$GainRatio(S \mid IdCode) = \frac{0.940}{3.807} = 0.246 bits$$

Gain was 0.94 => GainRatio is significantly lower

`outlook` - Gain=0.247, GainRatio=0.156
`temperature` - Gain=0.029, GainRatio=0.021
`humidity` - Gain=0.152, GainRatio=0.152
`windy` - Gain=0.048, GainRatio=0.049
`IdCode` – Gain=0.94, GainRatio=0.246

- In this case, `outlook` is penalised for splitting into 3 subsets, only marginally beating out the now improved `humidity`, which only splits into 2 subsets.
- `IDcode` is still the selected attribute, although its advantage is greatly reduced.

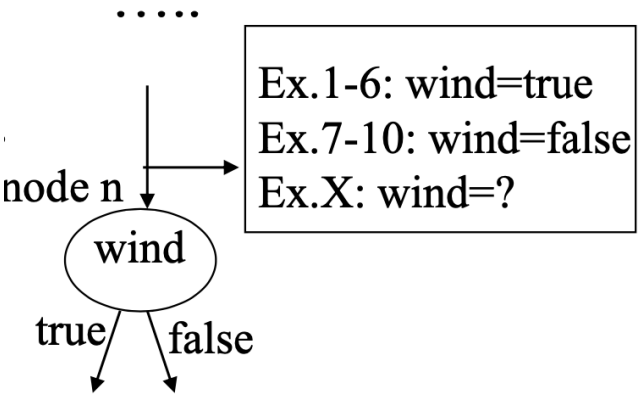
Handling Examples with Missing Values

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
?	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	?	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	?	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

- **Task**: At node n, we need to compute Gain for attribute A, but some of the examples that have reached n have missing values for A.

Solution with Another Example

- As we build the DT, we are at node n and need to calculate $Gain(wind)$. 11 examples have reached node n: ex.X with a missing value for wind, and 10 other examples without missing values.
- How do we calculate $Gain(wind)$?



- Solution:
  - Assign probability to each branch of wind using the examples without missing values that have reached n.
  - Use these probabilities in $Gain(wind)$ for the examples with missing values i.e. $P(wind = true) = 0.6, P(wind = false) = 0.4$.
  - For ex.X: 0.6 of ex.X is distributed down the branch for wind=true and 0.4 of ex.X down the branch for wind=false.
- These fractional examples are used to compute $Gain(wind)$ and can be further subdivided at subsequent branches of the tree if another missing attribute value must be tested.
- The same fractioning strategy can be used for classification of new examples with missing values.

Handling Attributes with Different Costs

- Consider a medical diagnosis with the following attributes: `temperature`, `biopsyResult`, `pulse`, `bloodTestResult`.
- These attributes have different costs (monetary, patient comfort). We prefer DTs that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classification.
- We learn a DT that contains low cost attributes by incorporating the cost in Gain by penalising high cost attributes.

Tan & Schlimmer

$$\frac{Gain^2(S \mid A)}{Cost(A)}$$

Nunez , where w ∈ [0,1] determines cost importance

$$\frac{2^{Gain(S|A)} - 1}{(Cost(A)+1)^w}$$

Lecture 8 - Introduction to NN, Perceptrons

Biological Neural Networks

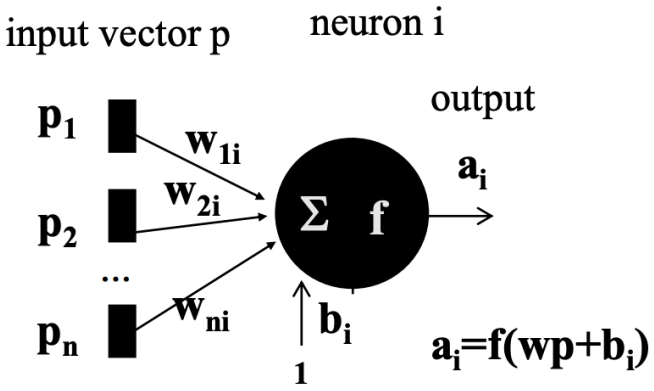
- Field of AI that studies networks of artificial neurons. Inspired by the desire to produce artifical systems capable of "intelligent" computations, and enhance our understanding of the human brain,
- Artificial neurons are simple abstractions of biological neurons connected to form networks that are computer implemented, and do not have a fraction of the power of the human brain.
- The brain performs tasks such as pattern recognition, perception, motor control many times faster than the fastest computers.
    - Perceptual recognition in 100-200ms e.g. familiar face in unfamiliar scene. Computers are still unable.
    - Sonar system of a bat has the precision of target location still impossible to match by current radars, with a brain the size of a plum.

Human Brain and Biological neuron

- Humans have 100 billion neurons in the brain, and 10 000 connections per neuron.
    - Neurons operate in miliseconds, computer in nanoseconds.
- Structure:
    - Body - contains the chromosomes
    - Dendrites - input
    - Axon - output. If the input signal is strong enough, an electrical signal is generated that travels along the axon.
    - Synapse - the narrow gap between the axon of one neuron and the dendrite of another. It is chemically activated, releasing neurotransmitters.
- Some of our neural structures are there at birth, which others e.g. synapses are formed and modified via learning and experience. Learning achieved by:
    - Creation of new synaptic connections between neurons.
    - Changing the strength of existing synaptic connections.
- The synapses are thought to be mainly responsible for learnining: "The strength of a synapse between 2 neurons is increased by the repeated activation of one neuron by the other across the synapse."

Artificial Neural Networks

- A network of many simple neurons (units, nodes), linked by connections that have a numeric weight (knowing from training examples). They are represented as a directed graph.
- Neurons:
    - receive numeric inputs (from the environment or other neurons) via the connections.
    - produce numeric output using their weights and inputs.
    - are organised into layers: input, hidden and output neurons.
- NN learn from examples, can be used for supervised or unsupervised learning, They are guided by a Learning Rule - a procedure for modifying the weights in order to perform a certain task.



Formally

- A connection from neuron i to j has a numeric weight w_{ij} which determines its strength.
- Given an input vector p, the neuron first computes the weighted sum wp, and then applies a transfer function f to determine the output a.
- The transfer function f has different forms depending on the type of NN.
- A neuron typically has a special weight called bias weight b. It is connected to a fixed input of 1.
- A NN represents a function of the inputs p and the weights w and b. By adjusting the weights, we change this function. This is done by using a learning rule.

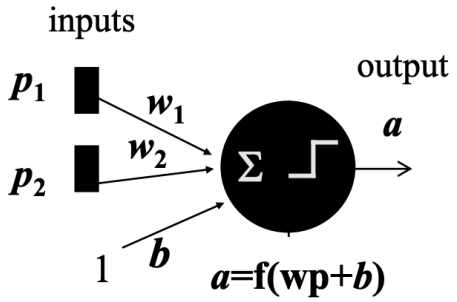
Taxonomy of NNs

- Feedforward (acyclic) and recurrent (cyclic, feedback). Supervised and unsupervised.
- **Feedforward supervised** networks are our main focus. Typically used for classification and regression, they include perceptrons, multilayer backpropagation networks, deep neural networks etc.
- Feedforward unsupervised networks include Hebbian networks for associative learning and competitive networks that perform clustering and visualisation.
- Recurrent networks are used for temporal data processing.

Perceptron

- A supervised NN that uses a step transfer function i.e. its input is binary e.g. 0/1 or -1/1.
- Its output is a weighted sum of its inputs, subject to a step transfer function.
- It forms a linear decision boundary. The step transfer function can for example be:

$$a = f(n) = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{if n} < 0 \end{cases}$$



- **1. Output:**

$a = step(\mathbf{wp}+b) =$
$= step(w_1 p_1 + w_2 p_2 + b)$

- **2. Decision boundary:**

$n = \mathbf{wp}+b = w_1 p_1 + w_1 p_2 + b = 0$

- **3. Let's assign values to the parameters of the perceptron (w and b):**

$w_1 = 1;\ w_2 = 1;\ b = -1;$

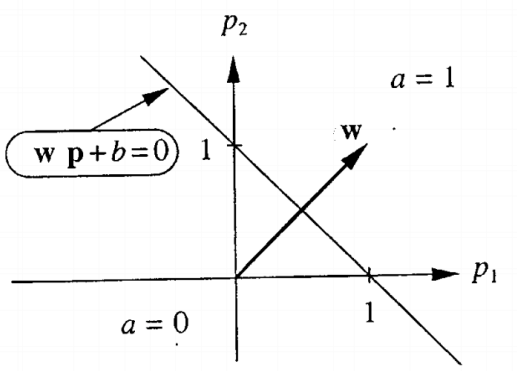
- **4. The decision boundary is:**

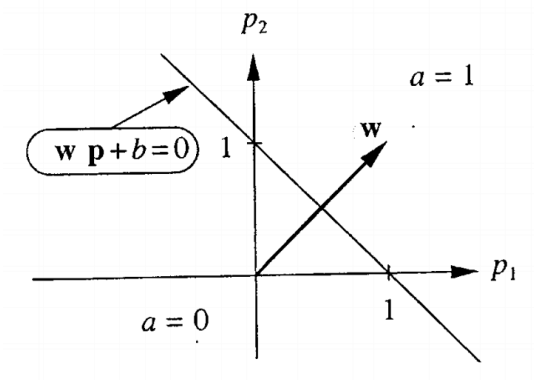
$n = \mathbf{wp}+b = w_1 p_1 + w_2 p_2 + b =$
$= p_1 + p_2 - 1 = 0$

- **It is a line in the input space**
- **It separates the input space into 2 subspaces: output =1 and 0**

- **5. Draw the decision boundary:**

$p_1 = 0 \Rightarrow p_2 = 1;$ (p_2 intersect)
$p_2 = 0 \Rightarrow p_1 = 1;$ (p_1 intersect)





- **Properties of the decision boundary – it is orthogonal to w**

- **=>The decision boundary is defined by the weight vector (if we know the weight vector, we know the decision boundary)**

- **Most important conclusion: the decision boundary of a perceptron is a line**

- **6. Find the side corresponding to an output of 1:**

$$\mathbf{p} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

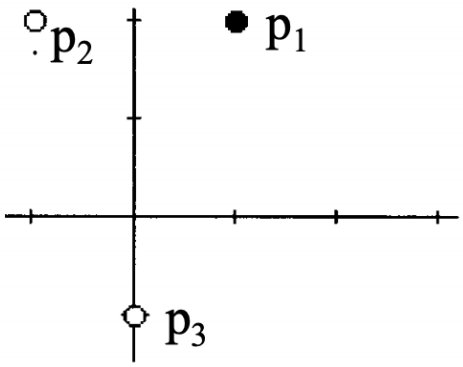
$$a = step(\mathbf{wp} + b) =$$
$$= step([1\,1]\begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1) = 1$$

Perceptron Learning Rule - Derivation

- Supervised Learning:
  - Given: a set of 3 training examples from 2 classes: 1 and 0. Supposing no bias weight.
  - Goal: learn to classify these examples correctly with a perceptron i.e. learn to associate input p_i with output t_i.

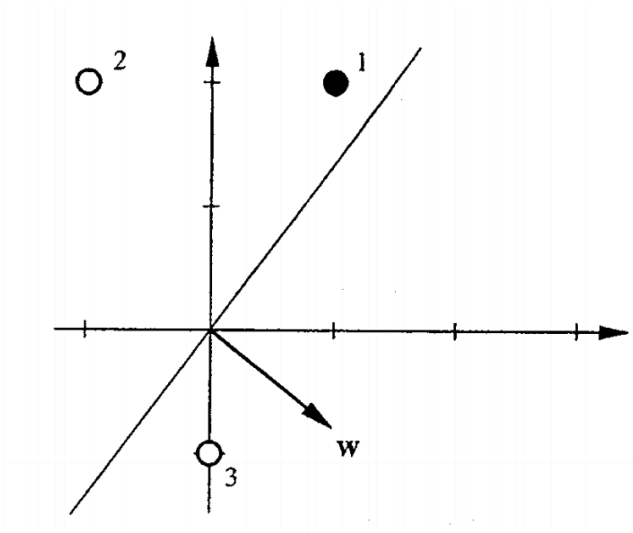
$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\} \quad \left\{\mathbf{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\} \quad \left\{\mathbf{p}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}$$

- Idea: Start with a random set of weights (i.e. random initial decision boundary), feed each examples, iteratively adjust the weights until the examples are correctly classified.



1) Initialisation of Weights

- Let out random initial weight vector be $w = [1 \ -0.8]$. This defines our initial classification boundary.



2) Start Training

- Feed the input examples to the perceptron iteratively, calculate the output and adjust w until all three examples are correctly classified.

First Input Vector

$$\textbf{First input example:} \quad \left\{\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\}$$

$$a = step(\mathbf{w}\mathbf{p}_1) = step([1 \quad -0.8]\begin{bmatrix}1\\2\end{bmatrix}) = step(-0.6) = 0$$

- Incorrect classification (Should be 1 but was 0). We need to alter the wight vector so that it points more toward p_1, so that in the future it has a better chance of classifying it correctly.
    - Let's add p_1 to w. Repeated presentations of p_1 would cause the direction of w to approach the

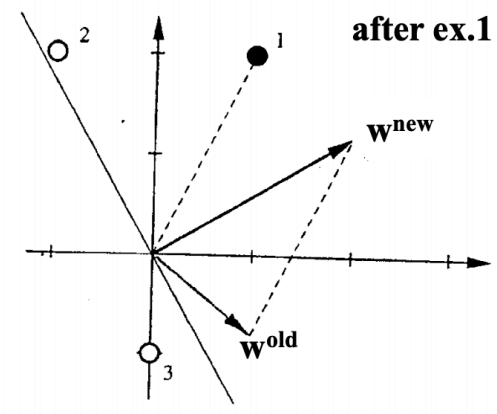
direction of p_1.

- **=> Tentative learning rule (rule 1):**

If $t = 1$ and $a = 0$, then $\mathbf{w}^{new} = \mathbf{w}^{old} + \mathbf{p}^T$

- **Applying the rule:**

$$\mathbf{w}^{new} = \mathbf{w}^{old} + p_1^T = [1 \quad -0.8] + [1 \quad 2] = [2 \quad 1.2]$$

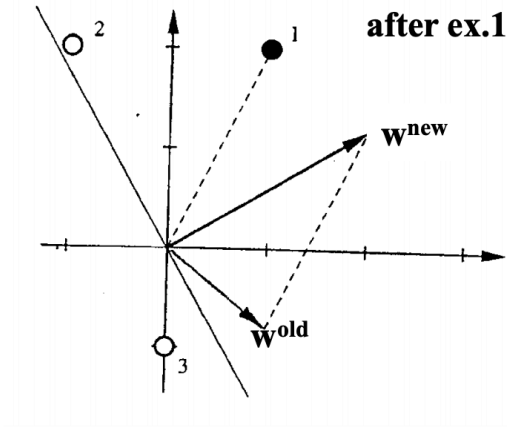


Second Input Vector

$$\left\{ \mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_2 = 0 \right\}$$

$$a = step(\mathbf{w}\mathbf{p}_2) = step([2 \quad 1.2]\begin{bmatrix} -1 \\ 2 \end{bmatrix}) = step(0.4) = 1$$

Incorrect classification!
Output should be 0 but it is 1!

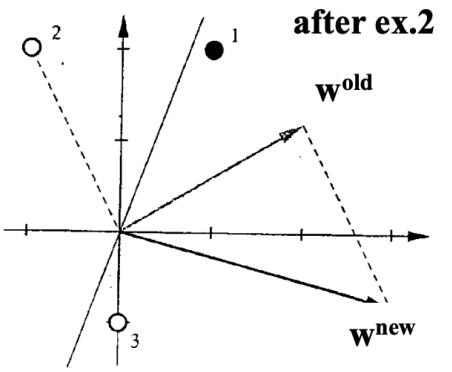


- **We can move the weight vector w away from the input (i.e. subtract it) => rule 2:**

If $t = 0$ and $a = 1$, then $\mathbf{w}^{new} = \mathbf{w}^{old} - \mathbf{p}^T$

- **Applying the rule:**

$$\mathbf{w}^{new} = \mathbf{w}^{old} - p_2^T = [2 \quad 1.2] - [-1 \quad 2] = [3 \quad -0.8]$$



Third Input Vector

- Output should be 0 but is 1. We subtract the third input vector from the weight to get a new weight vector.
- All examples have been fed once and we say that epoch 1 has been completed. We can check how each example is classified by the current classifier. If all are correctly classified, stop. Otherwise, repeat.

Perceptron Learning Law - Summary

1. Initialise weights including biases to small random values and set `epoch=1`.
2. Choose an example from the training set randomly.
3. Calculate the actual output of the network for this example a, also called network activation.
4. Compute the output error $e = t - a$.
5. Update the weights using the unified rule:

$$w^{\text{new}} = w^{\text{old}} + ep^T$$
$$b^{\text{new}} = b^{\text{old}} + e$$

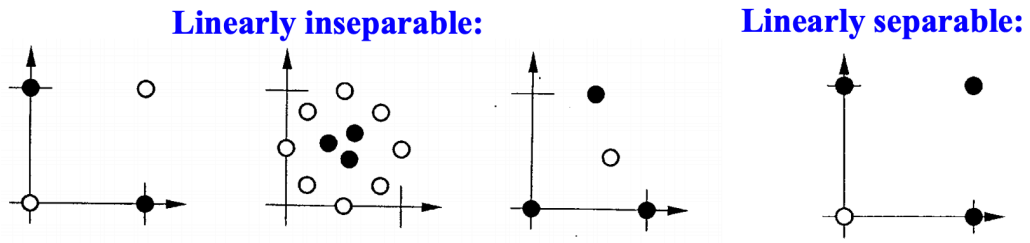
6. Repeat steps 2-5 by choosing another example from the training data.
7. At the end of each epoch, check if the stopping criteria is satisfied i.e. all examples are correctly classified or maximum epoch is reached. Otherwise, increment epoch and repeat from 2.

Stopping Criterion

- Checked at the end of each epoch/training epoch.
- All training examples are passed again, the output calculated and compared with NO weight change.
- Maximum epoch number is reached (starts from 1).

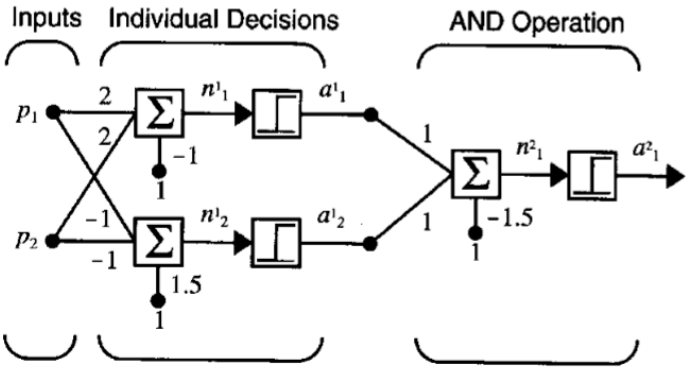
Capability and Limitations

- The output values of a perceptron can take only 2 values: 0 or 1 (or -1 and 1).
- Theorem: If the training examples are linearly separable, the perceptron learning rule is guaranteed to converge to a solution in a finite number of steps i.e. a set of weights that correctly classify the training examples.
- The perceptron will find a line (hyperplane) that separates the two classes. It doesn't try to find an optimal line (middle of positive and negative), but simply stops when a solution is found.

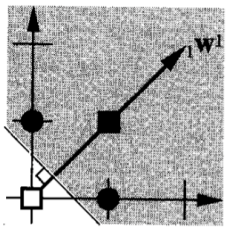


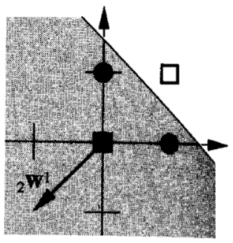
Multilayer Neural Networks

- The XOR problem is not linearly separable, and cannot be solved by a single layer perceptron network.
- It can be solved by a multilayer perceptron network however. The 2 perceptrons in the input layer identify linearly separable parts, and their outputs are combined by another perceptron to form the final solution.



•1st layer boundaries:

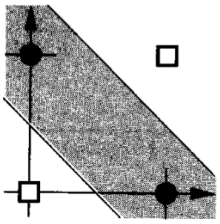




1st layer, 1st neuron; 1st decision boundary

1st layer, 2d neuron: 2nd decision boundary

• 2nd layer combines the two boundaries together:



2nd layer, 1st neuron: combined boundary

- Rosenblatt and Widrow (who proposed the perceptron in 1969) could not successfully modify the perceptron's rule to train more complex networks, and thus most of the scientific community walked away from NNs.
- An algorithm for training multi-layer perceptron networks were proposed in 1974, and discussed the backpropagation algorithm, which is also used to train deep NNs.

Neural Network Model

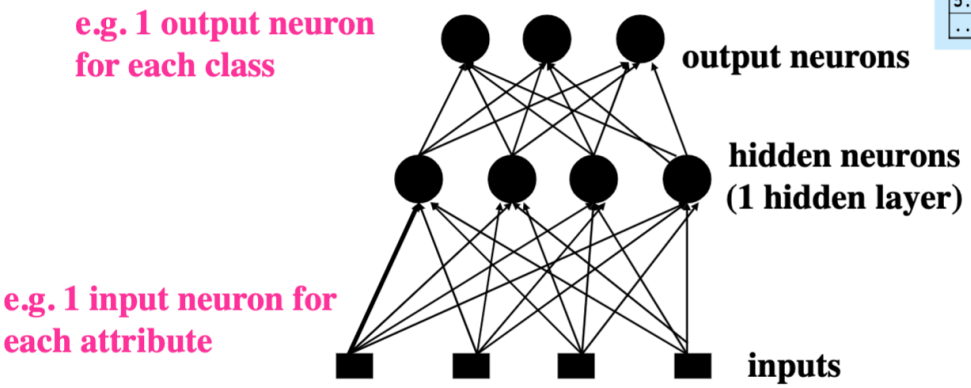
Model	
Architecture	Neurons and Connections. Input, Hidden, Output neurons. Fully or partially connected. Nuron model where computation performed by each neuron, type of transfer function. Initialisation of weights.
Learning Algorithm	How are weights changed in order to facilitate learning? Goal for Classification: mapping between input vectors and their classes.
Recall Technique	Once the NN training is completed, how is the information obtained from the trained NN?

Multi-Layer Perceptron Network Architecture

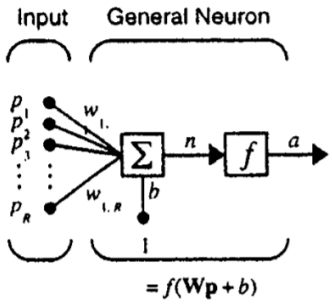
1) A network with 1 or more hidden layers

- **e.g. a NN for the iris data:**

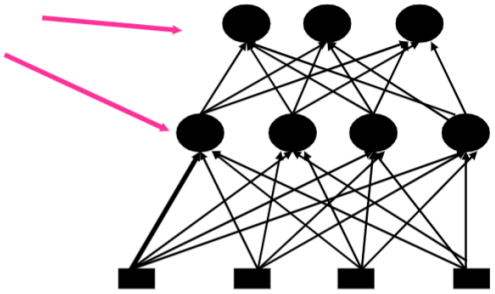
sepal length	sepal width	petal length	petal width	iris type
5.1	3.5	1.4	0.2	Iris setosa
4.9	3.0	1.4	0.2	Iris setosa
6.4	3.2	4.5	1.5	iris versicolor
6.9	3.1	4.9	1.5	iris versicolor
6.3	3.3	6.0	2.5	iris virginica
5.8	2.7	5.1	1.9	iris virginica
...				

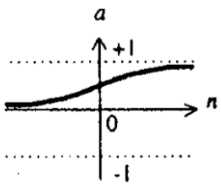


- A feedforward network - each neuron receives input only from the neurons in the previous layer.
- A fully connected network (typically) - all neurons in a layer are connected with all neurons in the next layer.
- Its weights are initialised to small random values e.g. $[-1, 1]$.
- Each neuron except the input neurons compute the weighted sum of its inputs, and applies a differentiable transfer function.
    - Any differentiable transfer function f can be used i.e. the derivative should exist for every point. Most commonly used are sigmoid and tan-sigmoid (hyperbolic tangent sigmoid).



$$a = f(\mathbf{w}\mathbf{p}+b)$$

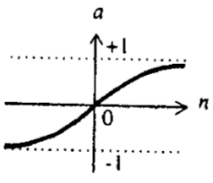




$a = logsig(n)$

Log-Sigmoid Transfer Function

$$a = \frac{1}{1+e^{-n}}$$



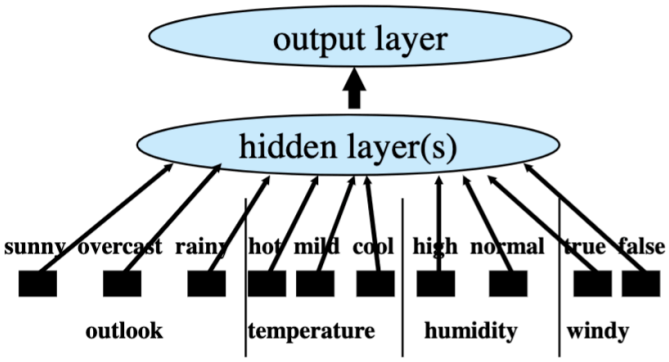
$a = tansig(n)$

Tan-Sigmoid Transfer Function

$$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$$

Number of Input Neurons

- Input neurons do not perform any computation - they just transmit data.
- For numerical data, 1 input neuron for each attribute.
- For categorical data, 1 input neuron for each attribute value. These input examples must be encoded - typically binary depending on the value of the attribute.



outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
...				

Number of Output Neurons

- Typically 1 neuron for each class. The targets typically use binary encoding.
- How to encode the output and represent targets?
  - Local Encoding: 1 output neuron, with different values representing different classes e.g. < 0.2 = class 1, > 0.8 = class 2, anything else is ambiguous (class 3).
  - Distributed (binary, 1-of-n) Encoding - typically used in multi class problems where number of outputs = number of classes.
- **Distributed is preferred over local encoding** since it provides more degree of freedom to represent the target function (n times as many weights available). Differences between outputs with the highest value and second highest can be used as a measure of confidence in the prediction (close values = ambiguous classification).

Number of Hidden Layers and Neurons in Them

- Typically, by trial and error. The given task (data) constrains the number of input and output but not the number of hidden layers.
  - Too many hidden layers and neurons (too many weights) leads to overfitting.
  - Too few leads to underfitting as the NN is not able to learn the input-output mapping.
  - A heuristic to start with is: 1 hidden layer with (inputs + outputs)/2) hidden neurons.

Lecture 9 - Backpropagation Algorithm, Deep Learning

Learning in Multi-Layer Perceptron NNs

- Supervised Learning on the training data:
  - Consists of labelled examples (p, d) i.e. the desired output d for them is given (p is the input vector).
  - Can be viewed as a teacher during the training process.
  - Error is the difference between the desired d and the actual a network output.
- Idea of the backpropagation learning. For each training example p:

    - Propagate p through the network and calculate the output a. Compare the desired d with the actual output a and calculate the error.
    - Update weights of the network to reduce the error.This is done until error over all examples is less than a threshold.
- It is called backpropagation since it adjusts the weights backwards (from the output to the input neurons) by propagating the weight chance Δw.

$$w_{pq}^{\text{new}} = w_{pq}^{\text{old}} + \Delta w_{pq}$$

Error Function

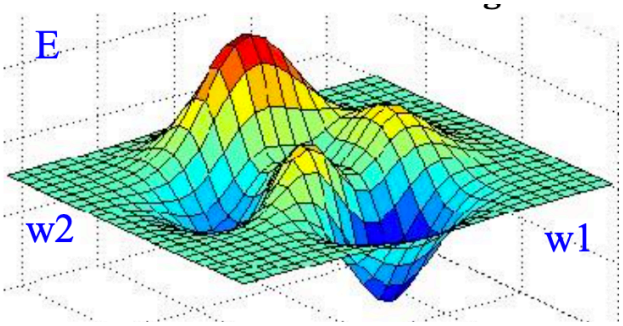
- Sum of Squared Errors (E) is a classical measure of error e.g. E for a single training example over all output neurons i:
    - where d_i is the desired value for the output neuron i, and a_i is the actual NN value for that neuron.

$$E = \frac{1}{2}\sum_i e_i^2 = \frac{1}{2}\sum_i (d_i - a_i)^2$$

- Backpropagation learning can be viewed as an optimisation search in the weight space.
    - Goal state: the set of weights for which the performance index (error E) is minimum.
    - Search method: hill climbing.

Error Landscape in Weight Space

- E is a function of the weights. 1 state = 1 set of weights. Several local minima and one global minimum.



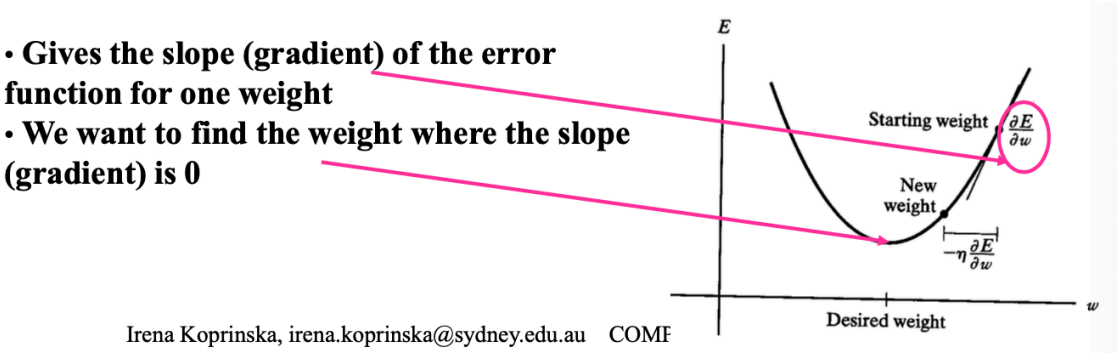
E as a function of w1 and w2

- Take steps downhill to minimise the error - this will find a local minimum (closest to the starting position) although it is not guaranteed to find the global minimum (unless there's only one minimum). The local minimum may be a good enough solution.
- Gradient descent doesn't guarantee that the error is reduced after each adjustment if a local minimum is reached.
- We get to the bottom as fast as possible (make the largest reduction in error) by using steepest gradient descent.

Steepest Gradient Descent

- The direction of the steepest descent is called gradient and can be computed (dE/dw). A function decreases most rapidly when the direction of movement is in the *direction of the negative of the gradient*.
- We want to adjust the weights so that the change moves the NN down the error surface in the direction of the locally steepest descent, given by the negative gradient.

- η - learning rate that defines the step, typically in the range $(0, 1)$.

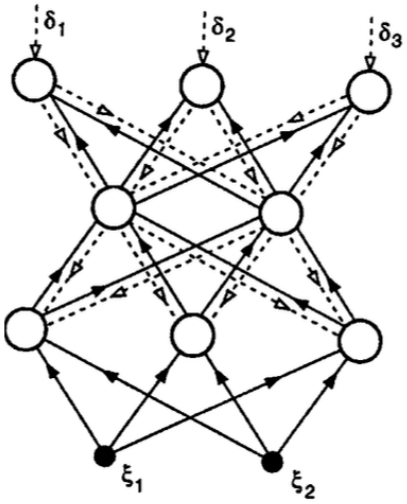


Universality of Backpropagation

- Every BOOLEAN FUNCTION of the inputs can be represented by a network with a single hidden layer.
- Continuous Functions: Universal Approximation Theorems - any continuous function can be approximated with arbitrarily small error by one hidden layer (Cybenko, Hornik et al).
- Any function (including discontinuous ones) can be approximated to arbitrary small error by a network with two hidden layers (Cybenko).
- NOTE: these are existence theorems - they say that a solution (NN) exists but don't say how to choose the number of hidden neurons. For a given NN, it is hard to say exactly which functions can be represented and which can't.
    - This also doesn't mean that 1 hidden layer is the most effective representation that will result in the fastest learning, easiest implementation, or best solution.

Backpropagation Algorithm Specifics

- The backpropagation algorithm adjusts the weights by working backwards from the output layer to the input layer $\rightarrow$ calculates the error and propagates this error from layer to layer.
- **Two approaches:**
    - Batch: weights are adjusted once, after all training examples are applied i.e. total error is calculated.
    - Incremental: weights are adjusted after each training example is applied. Also called stochastic or approximate gradient descent, it is the preferred method since it is faster and finds better solutions by escaping local optima.
- Solid lines are forward propagation of signals, while dashed lines are backward propagation of error.



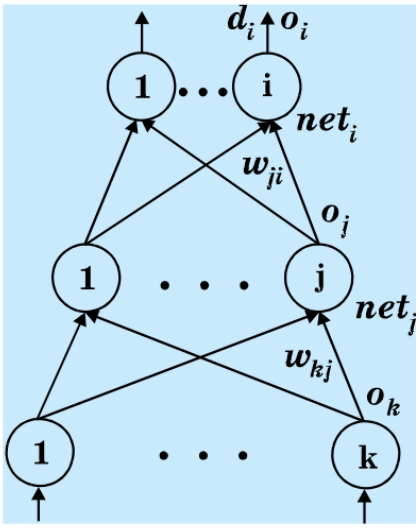
Notation

- A neural network with one hidden layer.
- Indexes for the neurons: k over inputs, j over hidden, i over output neurons.
- E – error function that we would like to minimize after each training example (i.e. incremental mode):

$$E = \frac{1}{2}\sum_i (d_i - o_i)^2$$

- Furthermore, d_i is the target value for neuron i for the example below, and o_i is the actual value for neuron i.
- NOTE: the weights are the parameters of the NN.

Expressing E in Terms of NN Weights and Input Example



1. Input for the hidden neuron j

$$net_j = \sum_k w_{kj} \cdot o_k + b_j$$

2. Activation of neuron j as function of its input

$$o_j = f(net_j) = f(\sum_k wkj \cdot o_k + b_j)$$

3. Input for the output neuron i

$$net_i = \sum_j w_{ji} \cdot o_j + b_i = \sum_j w_{ji} \cdot f(\sum_k w_{kj} \cdot o_k + t_j) + b_i$$

4. Output for output neuron i

$$o_i = f(net_i) = f(\sum_j w_{ji} \cdot o_j + b_j) = f(\sum_j w_{ji} \cdot f(\sum_k w_{kj} \cdot o_k + b_j) + b_i)$$

5. E in terms of NN weights

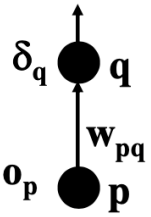
$$E = \frac{1}{2} \sum_i (d_i - o_i)^2 \text{where } o_i \text{ is as above}$$

Backpropagation Rule Summary

- In order to train our neural network, we wish to minimise the sum of the squared errors of the output values over all training examples, by backpropagating the error and adjusting weights across input and hidden layers. This is possible since the error can be expressed in terms of the weights, inputs and outputs. In doing so, we aim to use steepest gradient descent in order to find a local minima within the weight space. Standard GF is usually slow as it requires a small learning rate for stable learning.
- Given a current weight at time t, $w_{pq}(t)$, we want to add the weight change to find the weight at time $t+1$, i.e. $w_{pq}(t) + \Delta w_{pq}$.

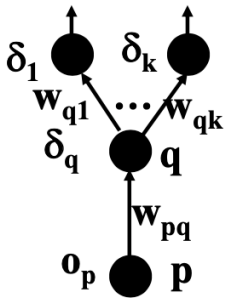
$$\Delta w_{pq} = \eta \cdot \delta_q \cdot o_p$$

- The weight change is proportional to the output activity of neuron p and the error δ of neuron q. The error is calculated in two different ways.
- q **is an output neuron**:



$$\delta_q = (d_q - o_q) \cdot f'(net_q)$$

- q **is a hidden neuron**:



$$\delta_q = f'(net_q) \sum_i w_{qi} \cdot \delta_i$$

- In both cases, $f'(net_q)$ is the derivative of the activation function used in neuron q with respect to the input of q, (net_q).
    - By using derivative tricks, we can represent $f'()$ in terms of $f()$. **NOTE:** $f(net_q) = o_q$.

Step by Step Summary

1. Determine the architecture of the network i.e. how many input, output, hidden neurons/layers, what output encoding.
2. Initialise all weights including biases to small random values, typically $\in [-1, 1]$.
3. Repeat until termination criterion satisfied (i.e. 1 **epoch** - 1 pass through training set):
    - (forward pass) Present a training example and propagate it through the network to calculate actual output of the network.
    - (backward pass) Compute the error (the δ values for the output neurons). Starting with the output layer, and repeating for each layer in the network, propagate the δ values back to the previous layer and update the weights between layers. Remember, biases are weights with input 1.
4. The stopping criterion is checked at the end of each epoch. Training stops when 1 of 2 conditions satisfied.
    - Error on training data is below a threshold set heuristically. This requires all training examples to be propagated again and the total error to be calculated (like a perceptron). Note: different types of errors may be used e.g. mean square, mean absolute, accuracy.
    - Maximum number of epochs is reached.
    - Instead of 1) or 1)+2), we can use **early stopping using a validation set** to prevent overfitting. Also, it typically takes hundreds or thousands of epochs for a NN to converge.

EXAMPLE 8b,9a 34.

How to Determine if an Example is Correctly Classified?

- Binary encoding of the target outputs.
    - Apply each example and get the resulting output activations of the output neurons; the example will belong to the class corresponding to the output neuron with the highest activation.
    - This is to say that the output value is regarded as the probability of the example to belong to the class corresponding to this output.

Overfitting

- Occurs when:
    - Training examples are noisy.
    - Small datasets are not representative of the whole data.
    - The number of free (trainable) parameters is bigger than the number of training examples i.e. more weights than training examples.
- Preventing overtraining can be done by using a network that is just large enough to provide an adequate fit.
    - Use Ockham's Razor - don't use a biggernet when a smaller one works!

    - Dont use a network with more free parameters than training examples.
- How large a network should be for a specific application is difficult to know beforehands.

Validation Set Approach

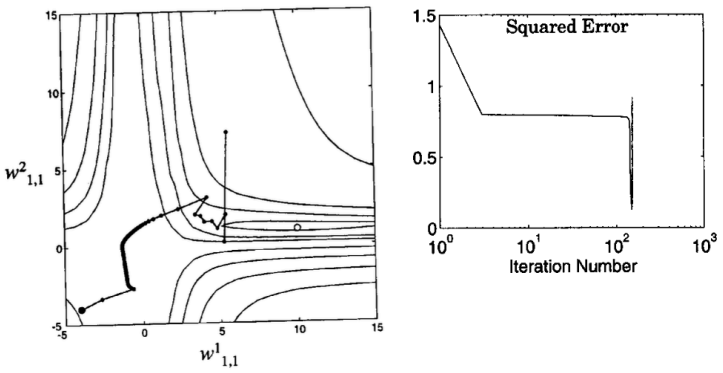
- Available data is divided into 3 subsets.
    - Training set: used for computing the gradient and updating the weights.
    - Validation set:
        - Error on this set is monitored during the training. It will normally decrease during the initial phase of training (as does training error).
        - When the networks begins to overfit the data, the error on the validation set will typically begin to rise.
        - Training is stopped when the error on the validation set increases for a pre-specified number of iterations and the weights and biases at the minimum of the validation set error are returned.
    - Testing set: not used during training but to compare different algorithms once training has completed.
- Problems with validation set approach is when the dataset is small - not enough data to provide a validation set, despite overfitting being most severe for small data sets.
- **K-fold Cross Validation**: perform k fold cross validation; determine the number of epochs that achieved best performance on the respective test partition; calculate the mean ep_mean, and on the final run, train the network for all examples with ep_mean epochs.

Error Surface and Convergence

- **Problem 1**: Path gets trapped within a local minimum - Try different initialisations!
- **Problem 2**: Path converges to the optimum solution, very slowly.

Speeding Up Convergence

- **Solution 1**: Increase the learning rate - faster on flat areas but unstable when falling into steep valleys that contain the minimum point i.e. overshoots minimum.



- **Solution 2**: Smooth out the trajectory by averaging the weight update.
    - Make the current update dependent on the previous by introducing an extra 'momentum' term in the weight adaptation equation.

Before:

$$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq}(t), \text{ where } \Delta w_{pq}(t) = \eta \delta_q o_p$$

Now:

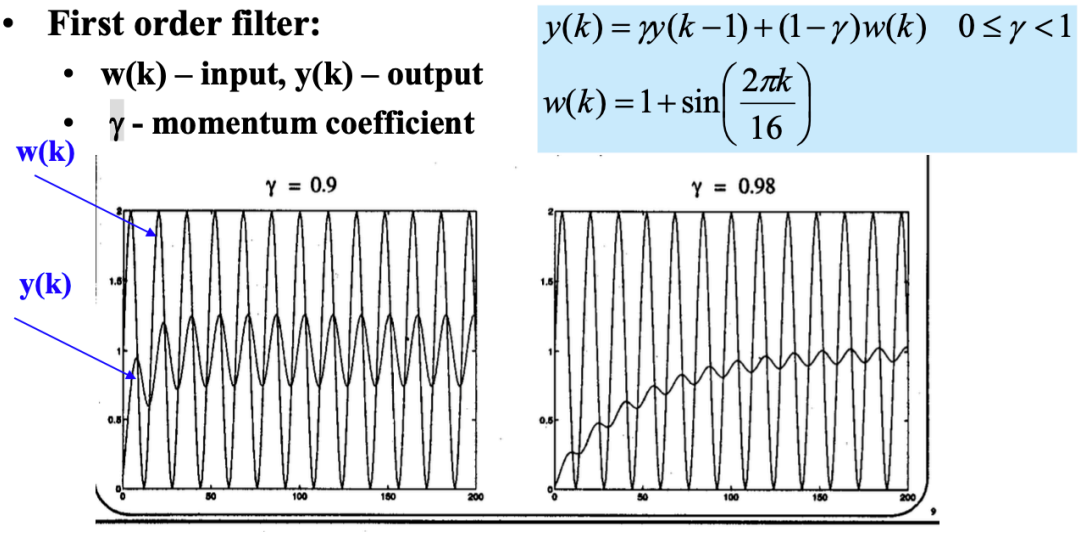
$$\Delta w_{pq}(t) = \eta \delta_q o_p + \mu\left(w_{pq}(t) - w_{pq}(t-1)\right)$$

momentum term

- If the previous change was large $\rightarrow$ the current change will be large and vice versa.
- Speeds up convergence along shallow gradient valleys (convergence is slow as the direction that has to be followed has a very small gradient $\rightarrow$ oscillations).
- The use of momentum typically smooths out the oscillations and produces a more stable trajectory.

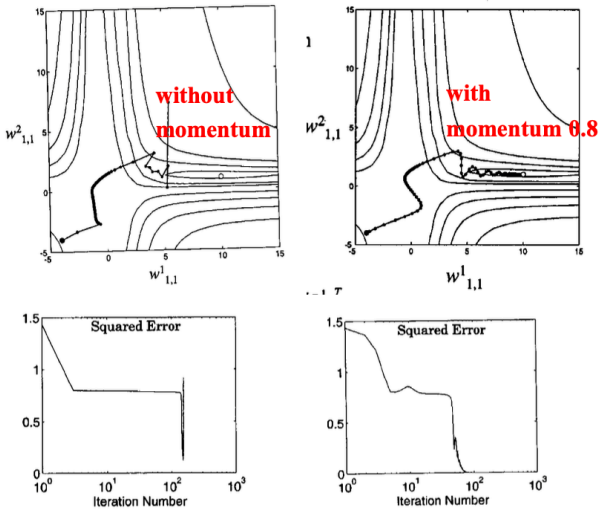
Momentum

- The theory behind momentum comes from linear filters. We observe that convergence might be improved by smoothing out the trajectory by averaging the updates to the parameters.



- In this case, the oscillation in the filter output $y(k)$ is less than the oscillation in the filter input $w(k)$. As the momentum coefficient increases, the oscillation in the output is reduced.
- The average filter output is the same as the average filter input (although as the momentum increases, the filter output is slow to respond) $\rightarrow$ the filter tends to reduce the amount of oscillation, while still tracking the average value.

- **Example – the same learning rate and initial position:**



- **Smooth and faster convergence**
- **Stable algorithm**
- **By the use of momentum we can use a larger learning rate while maintaining the stability of the algorithm**
- ***Try nnd12mo!***

- **Typical momentum values used in practice: 0.6-0.9**

Learning Rate η

- Constant throughout training for standard steepest descent.
- The performance is very sensitive to the proper setting of the learning rate. Too small - slow convergence, too big - oscillation and overshooting of the minimum.
- It is not possible to determine the optimum learning rate before training as it changes during training and depends on the error surface.
- Variable learning rate:
  - Goal: keep the learning rate as large as possible while keeping it stable.
  - Several algorithms have been proposed.

Some Practical Tricks

1. Stochastic (incremental) or batch learning
   - Use stochastic - it is faster and often finds better solutions.
2. Input examples
   - Shuffle the training set so that successive training examples never (rarely) belong to the same class.
   - Present input examples that produce a large error more often that examples that produce a small error.
   - Normalise the input variables to a standard deviation of 1.
   - If possible, use de-correlated input variables.
3. Transfer function
   - Use a tangent sigmoid, not the standard one – faster convergence.
4. Learning rate – a separate learning rate for each neuron so that all weights converge roughly with the same speed
   - Learning rate should be proportional to the square root of the number of inputs to the neuron.
   - Learning rates in the lower layers should be higher than in the higher layers.
5. Training - useful variations and extensions of the gradient descent method – Lavenberg-Marquardt method and conjugate gradient.

    - If the training set is large (> few hundred examples) and the task is classification, use stochastic gradient descent with careful tuning or the Lavenberg-Marquardt method.
    - If the training set is not too large or the task is regression, use the conjugate gradient method.

Limitations and Capabilities

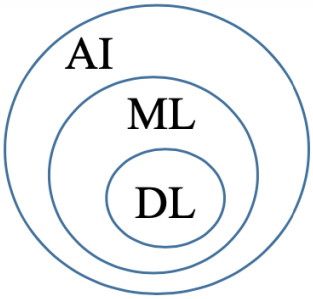
- Backpropagation NNs are used for supervised learning (classification or regression). Theoretically, they can:
    - Form arbitrary decision boundaries (i.e. both linear and non-linear).
    - Are universal approximators - can approximate any function arbitrarily well.
- In practice:
    - May not always find a good solution - trapped in local minimum.
    - Performance is sensitive to the start conditions i.e. weights initialisation, which is randomly done and adjusted by gradient descent.
    - Sensitive to the number of hidden layers and neurons.
        - Too few neurons - underfitting, unable to learn what you want it to learn.
        - Too many - overfitting and slow learning.
        - i.e. the architecture of a MLP network is not completely constrained by the problem to be solved as the number of hidden layers and neurons are left to the designer.
    - Sensitive to the value of the learning rate.
        - Too small - slow learning.
        - Too big - instability or poor performance.
    - The proper choices depends on the nature of examples.
        - Trial and error. Refer to the choices that have worked well in similar problems $\rightarrow$ successful application of NNs requires time and experience.
- Other issues include:
    - Training is slow and requires many epochs.
    - The NN is typically fully connected - too many parameters to adjust.
    - With many hidden layers, the learning becomes less effective - The **vanishing gradient problem** states that the weight changes for the lower levels become very small, and these layers end up learning slower than the higher hidden layers.
    - Require a large dataset of labeled data that may not be available, or may be difficult to obtain.
    - Require feature engineering to select useful features and represent them properly.

interesting Classical NN Applications 8b,9a

- This is before deep learning. Network design was the result of several months trial and error experimentation.
- Moral: NNs are widely applicable but they cannot magically solve problems, and wrong choices lead to poor performance i.e. NNs provide passable performance on many tasks that would be difficult to solve explicitly with other techniques.
- NETtalk: NN that tries to choose the correct English pronunciation i.e. cat vs century, differing c's.
- Driving Motor Vehicles, ALVIN: Learns to drive a van along a single lane on a highway.

Deep Learning

- Part of AI that focusses on creating large NNs that are capable of making accurate data-driven decisions.
- It is particularly suited for applications where the data is complex and large datasets are available.



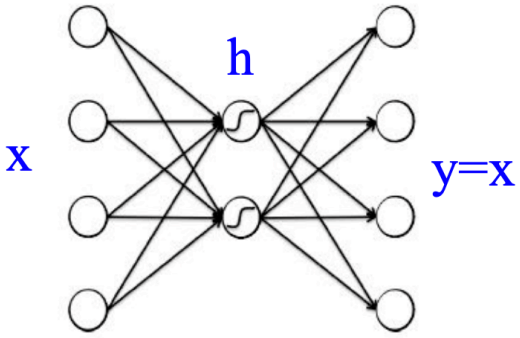
- Used by Facebook to analyse text in online conversations; Google, Baidu and MIcrosoft for image search, machine translation; smart phones for speech recognition and face detection; self-driving cars for localisation, motion planning, steering, tracking driver state; healthcare for processing medical images e.g. X-ray, CT, MRI.
- Used to produce AlphaGo, which beat Lee Sedol, the Go world champion in 2016.

More Specific Definitions

- The NN has more than 1 hidden layer.
- No need for human-invented and pre-selected features - it is able to learn the important features automatically.
- Some deep learning architectures use unlabeled data for pre-training of the NN layers, followed by supervised learning.
- Combined: Deep Learning is related to NNs that learn hierarchical feature representations.
- Some **Deep Learning architectures** include recurrent neural networks e.g. Long Short-term Memory (LSTM), Gated Recurrent UNit (GRU) which are used for sequences and text sequences, as well as restricted Boltzmann machines. The two that we are interested in are:
    - Stacked autoencoder networks
    - Convolutional networks

Autoencoder Neural Networks

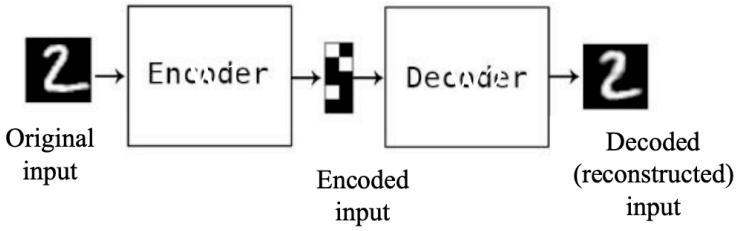
- First mentioned by Rumelhart, Hinton and Williams in 1986, they are typically used for dimensionality reduction, and image/data compression; more recently for pre-training networks (weight initialisation) in deep NN.
- We have a set of input vectors without their class (unlabelled data): $x = \{x_1, x_2, x_3 \dots\}$.
    - Each x_i is an n-dimensional vector representing 1 input vector.
- An autoencoder NN:
    - Sets the target values to be the same as the input values ($y_i = x_i$) and uses backpropagation to learn this mapping $\rightarrow$ the number of input and output neurons is the same.
    - Has one hidden layer with a smaller number of neurons than the input neurons.



- We are interested in the **outputs of the hidden neurons**, where h_i is the vector at the hidden layer for input vector x_i.
- The hidden layer can be seen as trying to learn a compressed version of the input vector.

For Encryption

- Also used in encryption, where w_1 performs encoding and w_2 performs decoding. The receiver needs w_2 to decode the encrypted input.

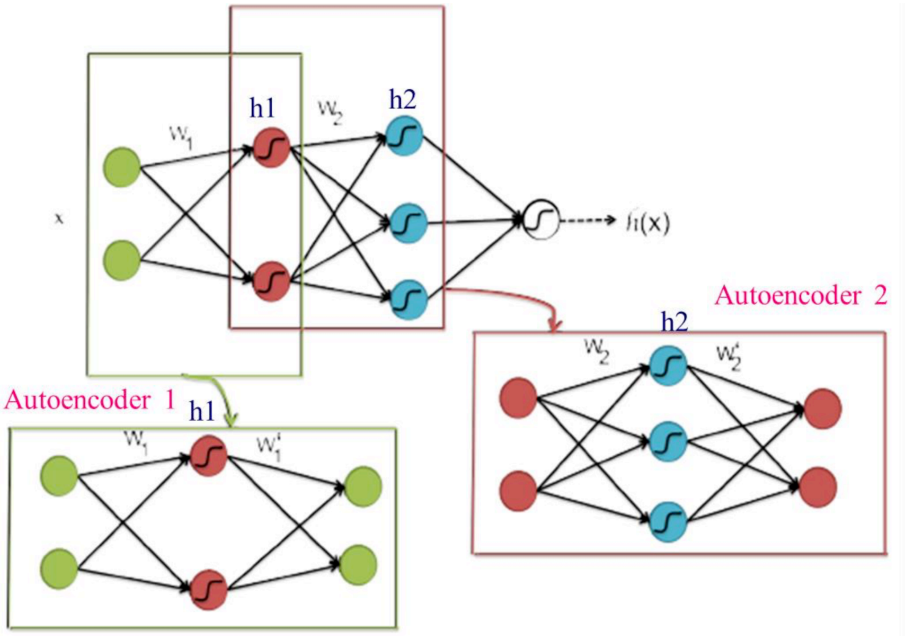


As Initialisation Method for Deep NN

- Can be used to pre-train the layers of a deep NN in advance - 1 layer at a time, 1 autoencoder for each layer.
- The training of a deep NN includes 3 steps:
  - Pre-training step: train a sequence of autoencoders, 1 for each layer (unsupervised).
  - Fine tuning step 1: train the last layer using backpropagation (supervised).
  - Fine tuning step 2: train the whole network using backpropagation (supervised).

Example

Pre-training step: Train a sequence of autoencoders, 1 for each layer (unsupervised) = 2 autoencoders for our example



- Pre-training means finding w_1 and w_2. To find w_1, we train Autoencoder 1 unsupervised with input vectors x only.
  - The learned w_1 is set in the deep NN as the values for the weights between the input and first hidden layer.
  - w_1' is discarded.
- We then find w_2, by using Autoencoder 2 with the now trained $h_1(x)$ as the input. The same process is followed as above.
- $h_1(x)$ can be seen as a different/compressed representation of the training data (a transformation has been applied), in which we hope that h_1 has discovered and extracted useful structure/pattern.
- **Fine-Tuning Step 1**: Train the last layer using backpropagation (supervised).
  - We compute the input for the training $h_2(h_1(x))$.
- **Fine-Tuning Step 2**: Train the whole network using backpropagation (supervised) as usual.

Stacked Autoencoders

- Stacking autoencoders is the process of using several autoencoders for pre-training.
  - Each layer of the networks learns an encoding of the layer below.
  - The network can learn hierarchical features in an unsupervised way.
- The network is called a stacked autoencoder.

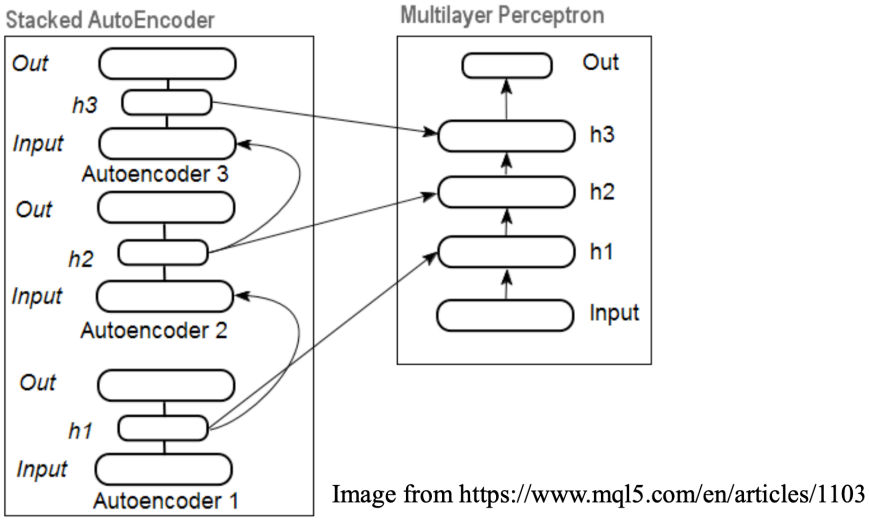


Image from https://www.mql5.com/en/articles/1103

Other Types of Autoencoders

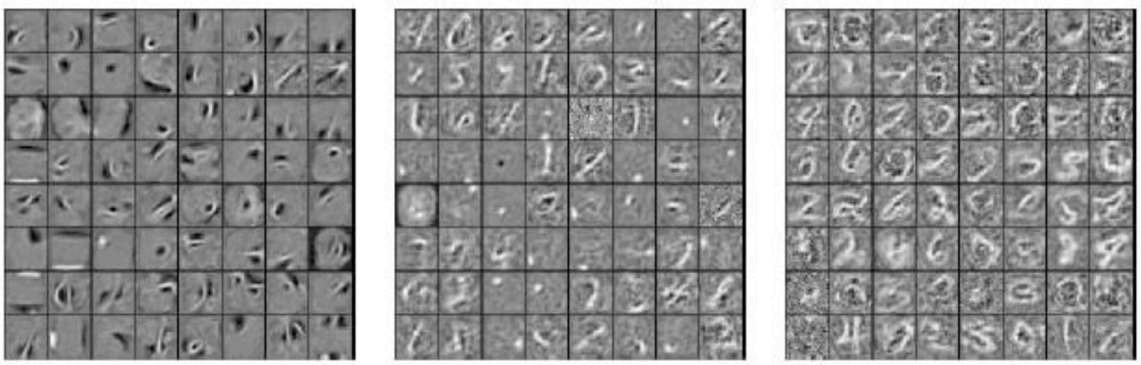
- **Spare Autoencoder** - an autoencoder with more hidden neurons than inputs (as in Autoencoder 2 above).
- **Denoising Autoencoder** - a percentage of the data is randomly removed, forcing the autoencoder to learn robust features that generalise better.

Visualising a Trained Autoencoder

- Consider image processing, we have trained the autoencoder on 20x20 images and have 100 hidden neurons.
- After the training has completed, we visualise what the autoencoder hass learnt (i.e. the function computer by each hidden neuron h_i) by showing an image that represents the maximal activation of each of the 100 hidden neurons.
- It can be shown that this image is formed by pixels computed as:

$$x_j = \frac{w_{ij}}{\sqrt{\sum_{j=1}^{400} w_{ij}^2}}$$

- Some of the hidden neurons have learnt to detect edges at different positions and with different orientations (useful for image recognition).
- A stacked autoencoder can learn a hierarchy of features e.g. 1st hidden layer - stroke-like features, 2nd - digit parts, 3rd - entire digits on MNIST dataset.

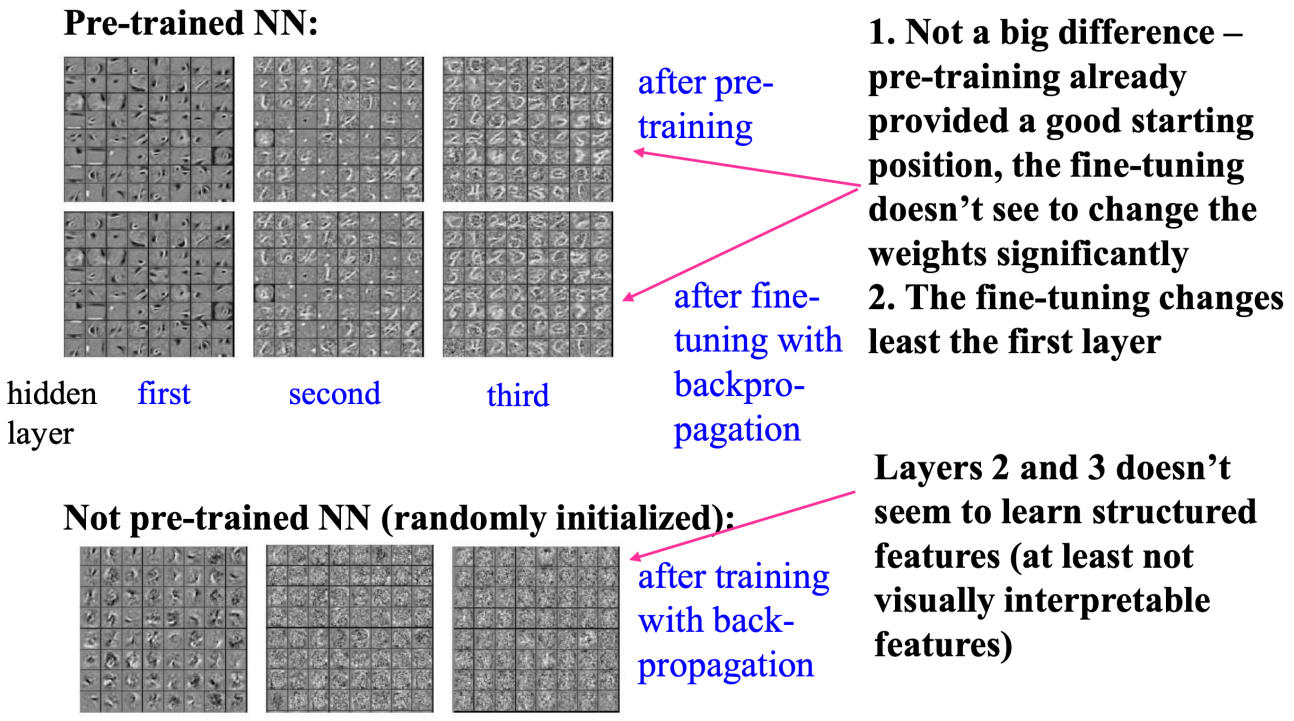


Autoencoders - Advantages

- Able to automatically learn features from unlabeled data.
    - Especially important for sensory data applications such as computer vision, audio processing, NLP (where researchers have spent many years manually devising good features).
    - In many domains, autoencoder learnt features are still not superior than the best hand-engineered featured, but more emerging cases where sophisticated autoencoders are.
    - Learned features can be used in conjunction with other ML/NN algorithms.

Unsupervised Pre-training Helps Deep Learning

- Compared deep NNs with and without pre-training experimentally on several big datasets.
    - Pre-trained converge faster and have better generalisation as opposed to slow training and easily stuck in poor local minima for random initialisation.
    - NN's with have better accuracy on test data.
    - NN's without, the porbability to find poor local minimum increases as the number of hidden layers increases. NN's with are robust to this.
    - NN's with provide a better starting position for the NN - in a "basin" with better local minimum.

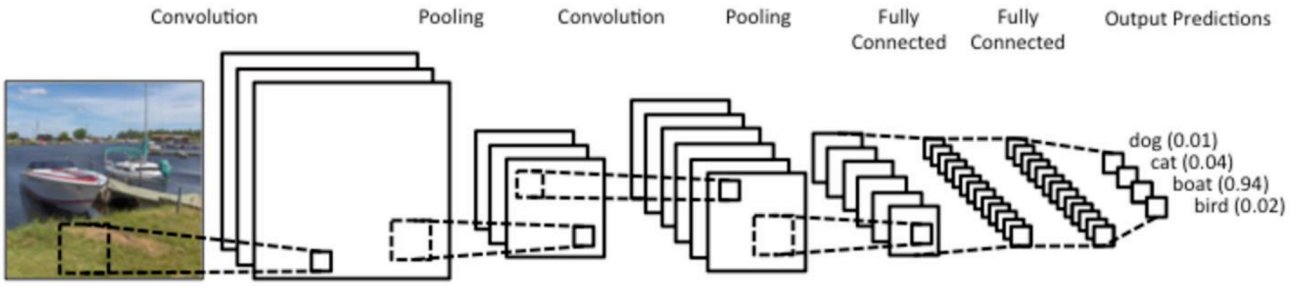


Convolutional Neural Networks

- Inspired by LeCun et al in 1989, it is a special type of multilayer NN, trained with backpropagation (as most other multilayer NNs) but with a different architecture.
- Designed to recognise visual patterns directly from pixel images with minimal pre-processing. Used in speech and image recognition with excellent performance.
- Can recognise patterns with high variability e.g. handwritten characters, and are robust to distortions and geometric transformations such as shifting.

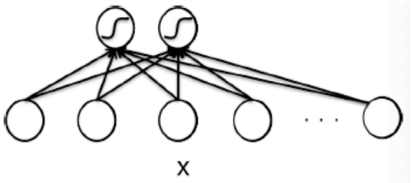
Summary

- Convolutional NNs contain a convolutional layer followed by a max-pooling layer and sometimes a LCN layer.
- This can be repeated several times i.e. output of max-pooling = input for convolutional, following by another max-pooling and LCN.
- Finally, there is 1 or 2 fully connected hidden layers. Backpropagation can be easily modified to train these networks.



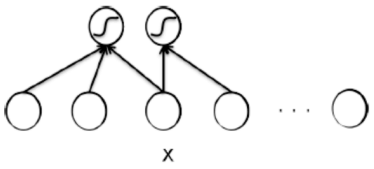
Main Idea 1 - Local Connectivity

- **Fully connected network** - each neuron is connected with each neuron in the next layer. Too many connections per neuron make it too computationally expensive e.g. 100x100 pixel image, 10^4 input vector = each hidden neuron in first layer has 10^4 connections + 1 bias weight.



Fully connected neuron

- **Locally connected network** - each hudden neuron is connected to a small subset of inputs, corresponding to adjacent pixels (a patch, continuous region in the image). Inspired by biological neural systems where neurons may have e.g. localised receptive fields.

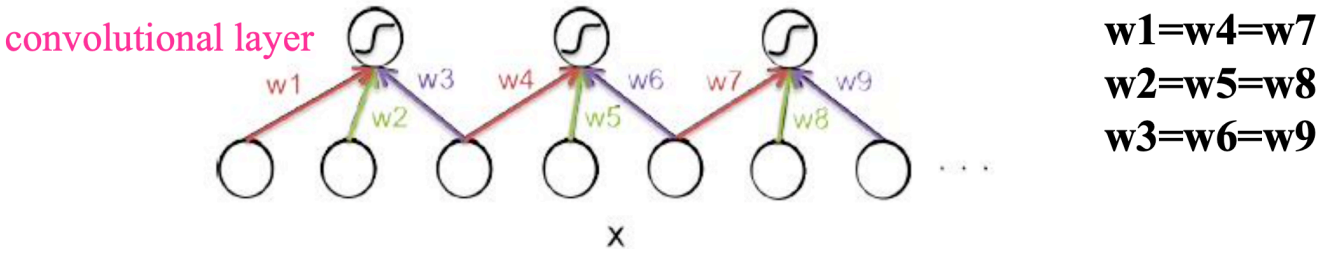


Locally connected neuron

- With local connectivity, each neuron is response to changes in its inputs only i.e. its receptive field. We can extend this idea to all layers.
- We can also easily modify the backpropagation algorithm to work with local connectivity:
  - Forward pass - assume that missing connections have weights 0.
  - Backward pass - no need to compute the gradient for the missing connections.

Main Idea 2 - Sharing Weights

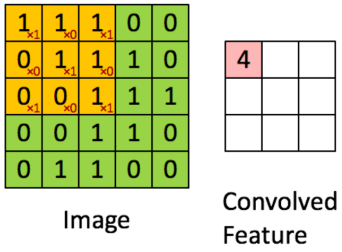
- The numer of connections can be further reduced by weight sharing - some of the weights are constrained to be equal to each other.



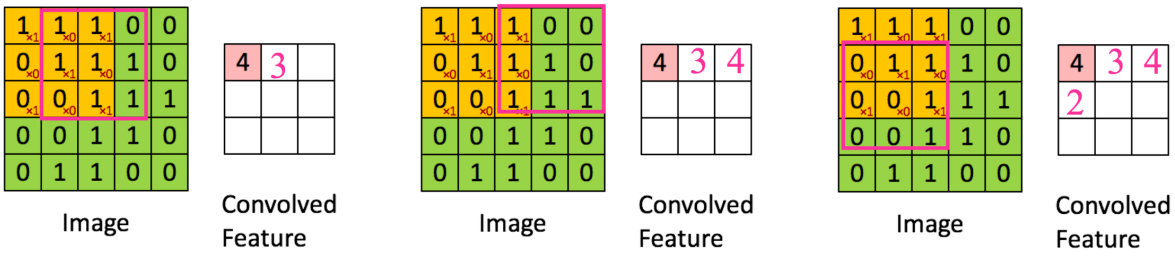
- In this case, we only need to store w_1, w_2, w_3 instead of all nine.
- Weight sharing means using the same weights for different parts of the image, which is similar to the convolution operation in signal processing - filter (set of weights) applied to different positions in the input signal.

- **Convolution is like applying a sliding window to a matrix**
- **The corresponding elements are multiplied and summed**

Demo at http://deeplearning.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/



- **image of black and white values: 0 is black 1 is white**
- **3x3 sliding window (filter, kernel) with values shown in red**
- **4 = 1*1+1*0+1*1+0*0+1*1+1*0+0*1+0*0+1*1**
- **Then the window is shifted as shown**

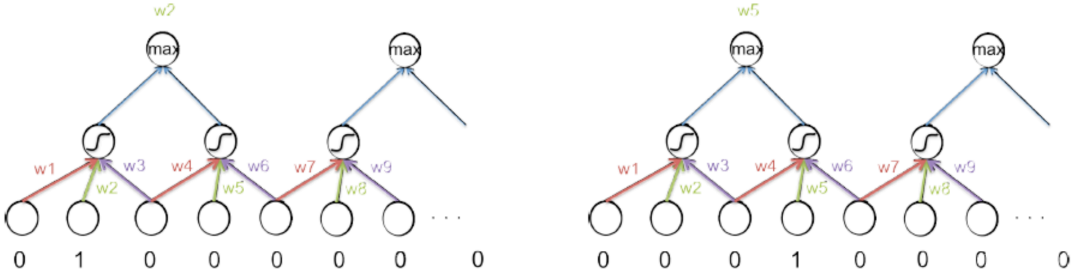


Main Idea 3 - Pooling

- The convolutional layer is used together with max-pooling layer, which takes the maximum value of a set of selected neurons from the convolutional layer.
- The pooling layer is called a subscampling layer because it reduces the size of the input data.
- Property: the output of a max-pooling neuron is invariant to shifts in the inputs/translation. Translational invariance is important for natural data such as images and sounds, as translation is a major source of distortion.

Example: 2 input images (1-dim), each with a white dot which got shifter 2 pixels to the right:

x1=[0,1,0,0,0,0,0…] **x2=[0,0,0,1,0,0,0…]**



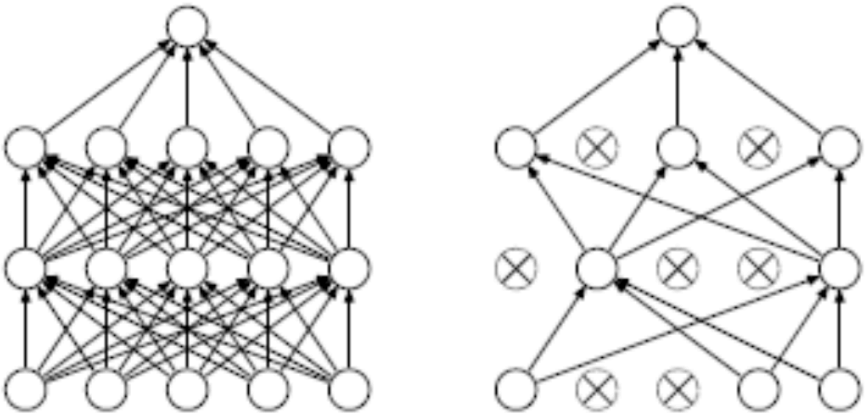
output first max-pooling neuron: w2 for x1 and w5 for x2
But w2=w5, so the value of the neuron is the same

Main Idea 4 - Local Contrast Normalisation

- Sometimes the max-pooling layer is following by a LCN layer.
- It normalises the output of each max-pooling neuron by subtracting the mean of their incoming neurons and dividing by the standard deviation of these neurons.
- LCN allows for **brightness invariance** which is useful for image recognition.

Main Idea 5 - Dropout

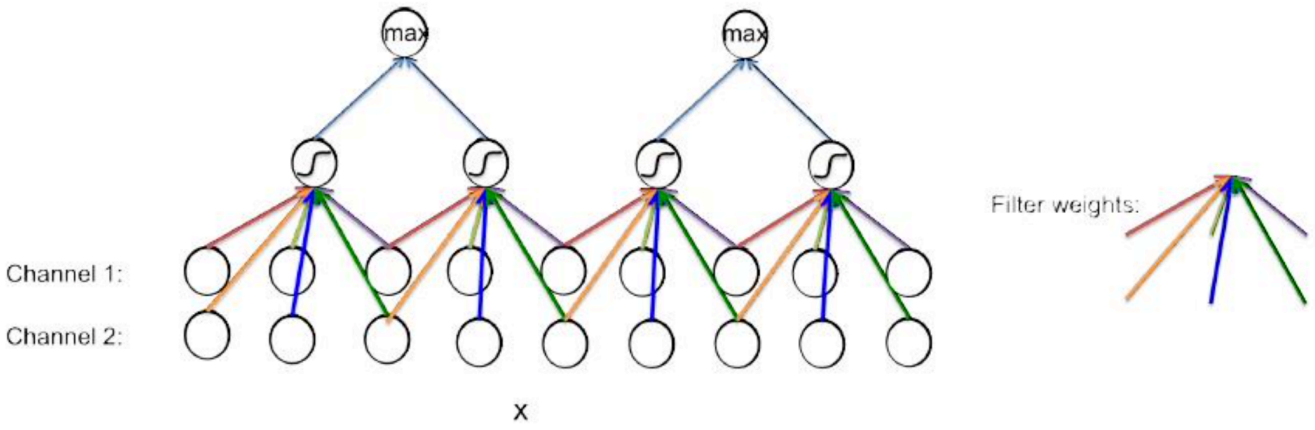
- Used in the fully connected layers to prevent overfitting.
    - During training, at each iteration of the backpropagation, we randomly select neurons in each layer and set their values to 0 (i.e. we drop them out from the weight adjust them, temporarily disabling them).
    - During testing, we do not drop out any neurons but scale their weights.
- Example: neurons at layer l have probability p to be dropped out. During testing, the incoming weights to layer l are multiplied by p.
- Dropout forces the NN to be less dependent on certain neurons, and collect more evidence from other neurons, making it more robust to noise.



(a) Standard Neural Net (b) After applying dropout.

Convolutional NNs with Multi-Channel Inputs

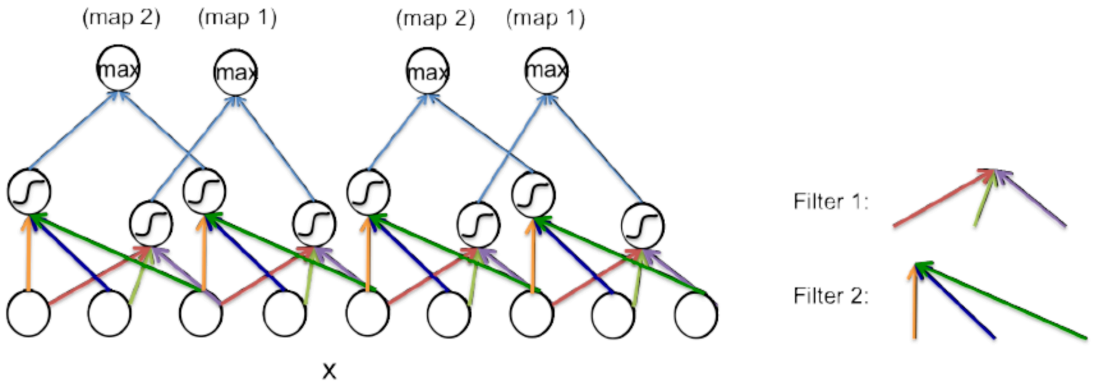
- Images with multiple channels e.g. RGB. We can modify the convolutional NN architecture to work with multiple channels.
    - A filter that looks at multiple channels.
    - Weights are not shared across a channel, only within one.



Convolutional NNs with Multiple Maps

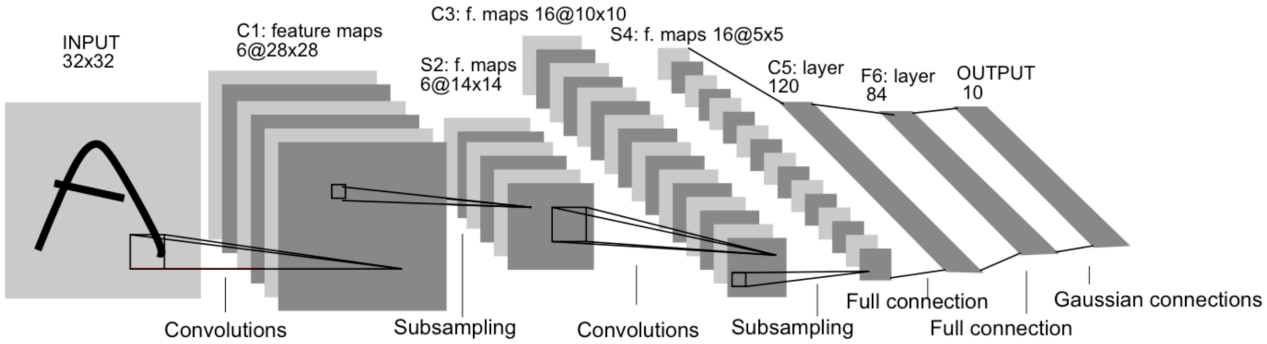
- We can have more than 1 filter for an input e.g. 2 filters looking at 1 pixel.
- The output produced by each filter is called a map.

Map 1 is created by Filter 1, Map 2 is created by Filter 2



Applications of Deep NNs

- LeNet-5 for Handwritten Digit Recognition - gradient-based learning applied to document recognition using 2 convolution, 2 max-pooling, 3 fully connected and local connectivity.



- ImageNet Image Classification - 5 convolutional layers, 3 max-pooling layers, 3 fully connected with 600 million weights, 650 000 neurons, dropout, rectified linear activation function, efficient GPU implementation of convolution.

Why the Dramatic Improvement in Deep Learning Now?

- The main ideas and algorithms have been around for a long time. Reasons:
    - Computational power - fast and powerful computers with powerful GPUs.
    - Availability of much larger datasets, especially labelled datasets - millions of examples.
    - Some new ideas e.g. dropout, autoencoders for pre-training, visualisation of what hidden layers learnt.
- HOWEVER, they are not a panacea that will solve all ML problems. Classical shallow NNs and other ML algorithms may do even better $\rightarrow$ you may not even need ML or NN to solve your problem.

Interpretability

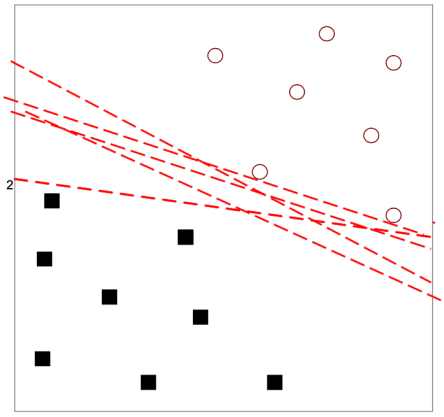
- Often we need not only a decision (predicted class) but also a reasoning behind it.
- This is especially important when the decision concerns a person e.g. medical diagnosis and credit assessment.
- Privacy and ethics regulations - individuals affected by decisions made by automated systems have the right for an explanation how the decision was made.
- Different algorithms provide different level of interpretability, but deep learning models are probablyt the LEAST interpretable.
- Research on interpretability is becoming more important.

Lecture 10 - Support Vector Machines

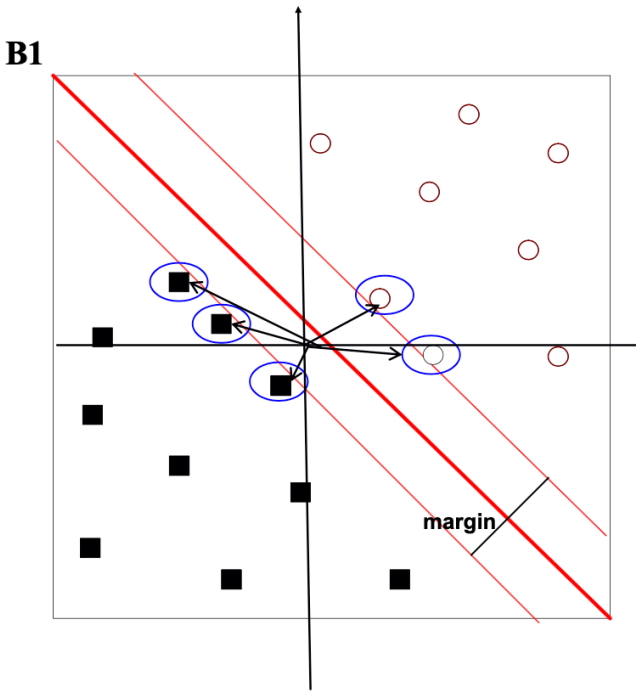
- Very popular, "off-the-shelf" classification method rooted in statistical learning theory, as it only has a few parameters that are easy to tune.
    - Maximise the margin of the decision boundary.
    - Transform data into a higher dimensional space where it is more likely to be linearly separable.
    - Kernel trick allows you to do calculations in the original, not the higher dimensional space.
- **Advantages**:
    - Can form arbitrary decisions both linear and non-linear.
    - The decision boundary is a maximum margin hyperplane i.e. has the highest possible distance to the training examples from the two classes $\rightarrow$ this helps to generalise well on new examples.
    - The decision boundary is defined by a subset of the training vectors called support vectors $\rightarrow$ resistant to overfitting.

Separation by a Hyperplane

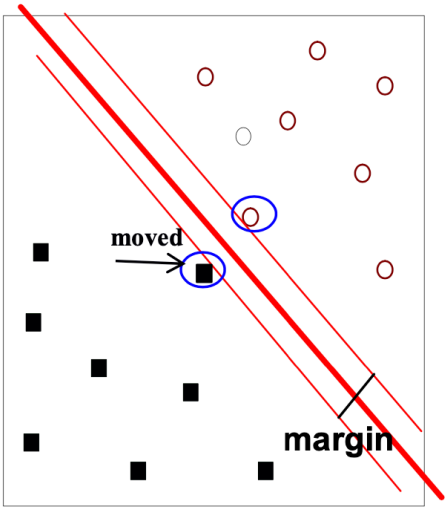
- Given a 2 class problem, squares and circles, find a linear boundary that separates the data.



- **Given a decision boundary B1:**
  - Support Vectors: the data points that lie closest to the decision boundary. It is possible to have more than 1 support vector for each class e.g. 5 total, 3 square, 2 circle → remember that training examples are given as input vectors, where each dimension corresponds to 1 feature.
  - Margin: the separation between the boundary and the closest examples (or the width the boundary can be increased before touching an example). The boundary is in the middle of the margin.



- Support vectors define the decision boundary. If we move a support vector, the decision boundary changes. This is not the case when a non support vector is moved.



- The hyperplane with the bigger margin is "better" and should be selected.

Maximum Margin Hyperplane

- The hyperplane (separator) with the biggest margin is called the **maximum margin hyperplane (separator)**.
    - A SVM selects the maximum margin hyperplane, since they are typically more accurate on new examples.
- Intuitively, it feels safer. We don't know where new examples will be but we assume they will be drawn from the same distribution as training examples.
- If we make small errors in the location of the boundary, the chances of causing misclassifications will be smaller if the margin is big $\rightarrow$ big margin = less sensitive to noise and overfitting.

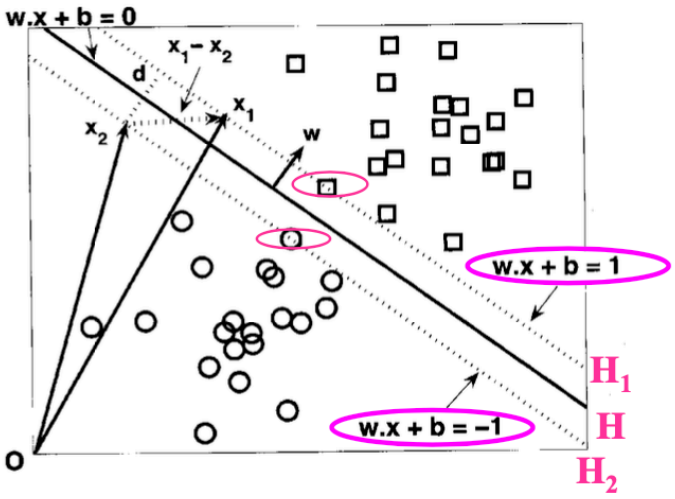
Maximum Margin - Formal Justification

- Structural risk minimisation principle from computatinal learning theory:
    - Generalisation error (error on new examples) depends on the training error and model complexity (also called model capacity).
    - If the training error is the same, the classifier with the lower complexity will generalise better.
- Small/big margin = high/low complexity = higher/lower generalisation error

Quick Revision - Linear Decision Boundary

- A decision boundary of a linear classifier is $w \cdot x + b = 0$ where w and b are parameters.
- Determining the sign will give us the classification of an example x_i.

Problem Statement for SVM



- Our separating hyperplane is H, which is between two others H_1 and H_2 defined as:
  - $H_1 : w \cdot x + b = 1$
  - $H_2 : w \cdot x + b = -1$
- d is the margin of H, with the point lying on the two hyperplanes above being the support vectors.
- It can be shown by calculating the distance between a point from H and the hyperplane H_1 and then multiplying by 2, that d can be expressed as:

$$d = \frac{2}{||w||}$$

- To maximise the margin d, we must minimise $||w||$, i.e. minimise $\frac{1}{2}||w||^2$.
- The optimisation problem can be transformed into its dual optimisation problem:

$$\max \mathbf{w}(\lambda) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{N} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

Dot product of pairs of training vectors

Target value (class value) of the training vectors

$$subject\ to\ \lambda_i \geq 0, \sum_{i=1}^{N} \lambda_i y_i = 0$$

- This is a Quadratic Programming (QP) problem and can be solved (i.e. the λs can be estimated) using a standard numerical procedure.
  - Global maximum of the λs (Lagrange multipliers) always exist.
  - w i.e. the optimum decision boundary:

$$w = \sum_{i=1}^{N} \lambda_i y_i x_i$$

All Combined

- Given N labelled training examples

$$(x_i, y_i), i = 1, \ldots, N$$
$$x_i = (x_{i1}, \ldots, x_{im})^T, y_i = \{-1, 1\}$$

- Minimise $\frac{1}{2}||w||^2$ subject to the linear constraint $y_i(w \cdot x_i + b) \geq 1, \forall i$, explained further 10a 18.

    - Learn the maximum margin hyperplane such that all training examples are classified correctly.
- Using QP, where λ are Lagrange multipliers:

$$w = \sum_{i=1}^{N} \lambda_i y_i x_i$$

- Since many λs are 0, the optimal decision boundary w is a linear combination ^ of support vectors, which have a non-zero λ_i.
- To classify a new example z:

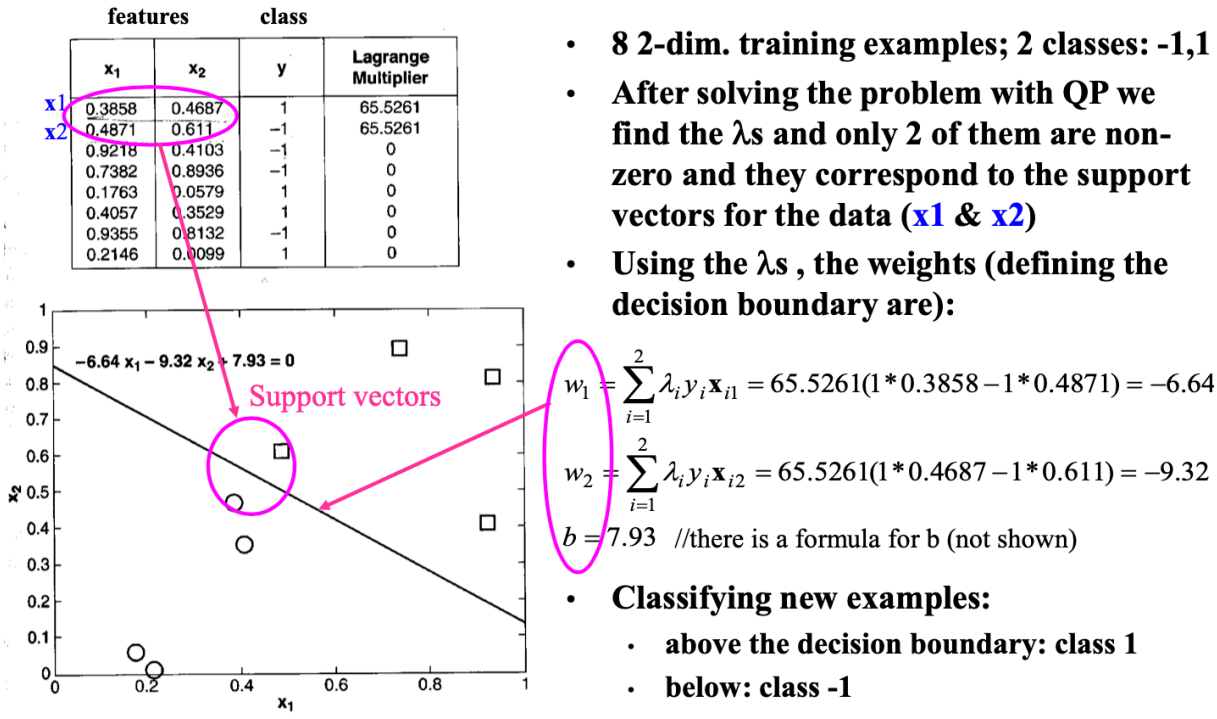
$$f = \mathbf{w} \cdot \mathbf{z} + b = \sum_{i=1}^{N} \lambda_i y_i \mathbf{x}_i \cdot \mathbf{z} + b$$

Dot product of the new vector and the support vectors

$$sign(f)$$

i.e. the new example belongs to class 1, if f>0 or class -1 if f<0

- For example:



features		class	
x_1	x_2	y	Lagrange Multiplier
0.3858	0.4687	1	65.5261
0.4871	0.611	−1	65.5261
0.9218	0.4103	−1	0
0.7382	0.8936	−1	0
0.1763	0.0579	1	0
0.4057	0.3529	1	0
0.9355	0.8132	−1	0
0.2146	0.0099	1	0

- **8 2-dim. training examples; 2 classes: -1,1**
- **After solving the problem with QP we find the λs and only 2 of them are non-zero and they correspond to the support vectors for the data (x1 & x2)**
- **Using the λs , the weights (defining the decision boundary are):**

$$w_1 = \sum_{i=1}^{2} \lambda_i y_i \mathbf{x}_{i1} = 65.5261(1*0.3858 - 1*0.4871) = -6.64$$

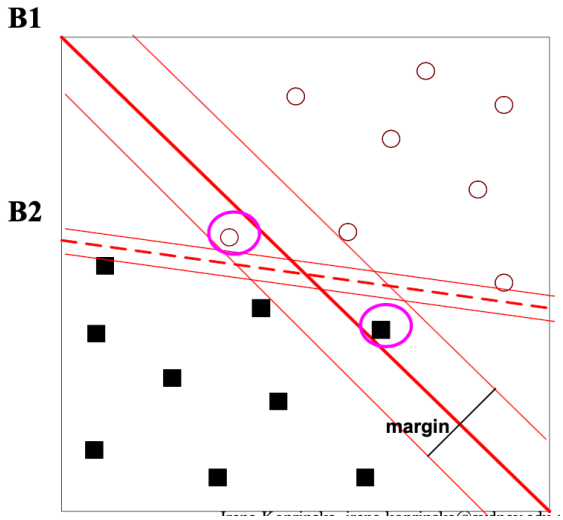
$$w_2 = \sum_{i=1}^{2} \lambda_i y_i \mathbf{x}_{i2} = 65.5261(1*0.4687 - 1*0.611) = -9.32$$

$b = 7.93$ //there is a formula for b (not shown)

- **Classifying new examples:**
    - **above the decision boundary: class 1**
    - **below: class -1**

Soft Margins

- The method above constructs decision boundaries that are free of misclassifications. We can allow some misclassifications.

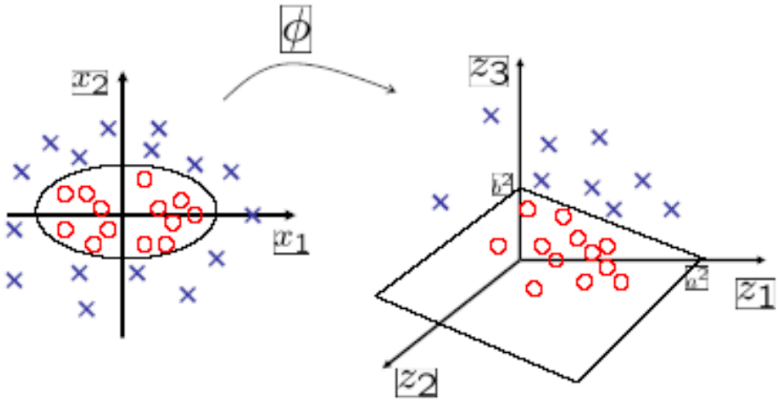


- B_2 classifies these new examples correctly unlike B_1, but these examples may be noise, and still prefer the wide margin and lesser sensitivity to overfitting and noise of B_1.
- The optimisation problem formulation is similar for a **soft margin solution**, but there is an extra parameter C in the function that we want to maximise, corresponding to the tradeoff between error and margin.
    - The solution is the same as the hard margin, but there is an upper bound C on the values of λs.
- The modified method will still construct a linear boundary even if the data is not linearly separable.

Non-Linear SVM

- Most problems are linearly non-separable. In order to find a non-linear boundary, we want to transform the data from its original feature space to a new space where a linear boundary can be used to separate the data.
    - Cover, 1965: A complex classification problem cast in high dimensional space non-linearly is more likely to be linearly separable than in a low dimensional space.
- The transformation has two properties:
    - It is non-linear.
    - It is to a higher dimensional space.

Non-linearly separable data $\mathbf{x} = (x_1, x_2)$ in the original space and linearly separable in the new space



$\phi : (x_1, x_2) \longrightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2)$ transformation from old to new space

$\left(\frac{x_1}{a}\right)^2 + \left(\frac{x_2}{b}\right)^2 = 1 \longrightarrow \frac{z_1}{a^2} + \frac{z_3}{b^2} = 1$ decision boundaries in old and new space

original space (2-dim): (x_1, x_2)

new space (3-dim): $(x_1^2, \sqrt{2}x_1x_2, x_2^2)$

- Intuitively for example above, we transform our 2D shape into a hill, and use a plan to cut through.

Issue 1

- How do we choose the dimensionality of the new space so that the data is linearly separable in the new space?
  - Theorem: If a dataset is mapped to a space of sufficiently high dimension, it will always be linearly separable.
  - If we use a space of infinite dimensionality though, we run the risk of overfitting.
  - A line in a d-dimensional space is defined by an equation with d parameters $\rightarrow$ if $d \approx N$, overfitting i.e. restrictions on dimension defined by data.
- We also deal with overfitting by constructing the maximum margin hyperplane in the new space (structural risk minimisation principle).

Issue 2

- Even if we know what the transformation Φ should be, solving a QP task in a new, higher dimensional space is computational expensive.

  1) Transform the 2-dim training data into 3-dim using Φ.

$$\mathbf{x}_i \overset{\Phi}{\rightarrow} \Phi(\mathbf{x}_i), \mathbf{x}_j \overset{\Phi}{\rightarrow} \Phi(\mathbf{x}_j)$$

$\mathbf{x}_1, \mathbf{x}_2$ – a pair of training vectors in the original 2-dim feature space

  2) Feed the 3-dim vector to the QP solver. Recall that QP computes dot product pairs of training vectors:

$$\max W(\lambda) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{N} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

Before (in the original 2-dim space)

$$\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$$

Now (in the new 3-dim space)

- The computation increases dramatically for even more dimensions.

Kernel Trick

- We learn in higher dimensional space by computing kernel functions in lower dimensional space, instead of transforming each vector into the higher dimensional space and computing dot products between vectors.
- We need the dot product of the features in the new space (the transformed features):

$$\Phi(x_i) \cdot \Phi(x_j)$$

- We do NOT want to do the following:

$$1) x_i \overset{\Phi}{\rightarrow} \Phi(x_i), x_j \overset{\Phi}{\rightarrow} \Phi(x_j)$$
$$2) \Phi(x_i) \cdot \Phi(x_j)$$

- Instead, we want to specify K in the original space i.e. indirectly specifying Φ, and perform the dot product computation in the original space which has smaller dimensionality to find a linear boundary in the new higher dimensional space.

$$K(u, v) = \Phi(u) \cdot \Phi(v)$$

Method

2 dim original and 3 dim new space, $\Phi : (x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$

Consider a pair of vectors in original space: u, v (they are 2-dim). They were transformed into the (3 dim.) vectors Φ(u), Φ(v) in the new space, i.e.

$$\mathbf{u} \overset{\Phi}{\rightarrow} \Phi(\mathbf{u}), \mathbf{v} \overset{\Phi}{\rightarrow} \Phi(\mathbf{v})$$

Let's calculate the dot product of Φ(u) and Φ(v)

$$\Phi(\mathbf{u}) \cdot \Phi(\mathbf{v}) = (u_1^2, \sqrt{2}u_1u_2, u_2^2) \cdot (v_1^2, \sqrt{2}v_1v_2, v_2^2) =$$
$$= u_1^2v_1^2 + 2u_1u_2v_1v_2 + u_2^2v_2^2 = (u_1v_1)^2 + (u_2v_2)^2 + 2u_1u_2v_1v_2 =$$
$$= (u_1v_1 + u_2v_2)^2 = (\mathbf{u} \cdot \mathbf{v})^2$$

The dot product in the new space can be expressed via the dot product in the original space! $\Phi(\mathbf{u}) \cdot \Phi(\mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^2$ **Nice property!**

This means that the dot product in the new space can be computed without first computing Φ for each input vector! Instead we will compute a function in the original space to evaluate the dot product in the new space!

- In this case, the circled expression is called a kernel function $K(u, v)$.
- NOTE: functions K for which this is true need to satisfy Mercer's Theorem, which restricts the class of functions K we can use:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^p - \textit{polynomial kernel}$$
$$K(\mathbf{x}, \mathbf{y}) = e^{-\frac{\|x-y\|^2}{2\sigma^2}} - RBF$$
$$K(\mathbf{x}, \mathbf{y}) = \tanh(k\mathbf{x} \cdot \mathbf{y} - \theta) - \textit{tanget hyperbolic}$$
$$(\textit{satisfies Mercer's Th. only for some } k \textit{ and } \theta)$$

- In terms of **TRAINING**:
  - **Original, without kernel function**

$$\max \mathbf{w}(\lambda) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{N} \lambda_i \lambda_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$subject\ to\ \lambda_i \geq 0, \sum_{i=1}^{N} \lambda_i y_i = 0$$

solution

$$\mathbf{w} = \sum_{i=1}^{N} \lambda_i y_i \mathbf{x}_i$$

Optimal hyperplane in the original space

  - **With kernel function**

$$\max \mathbf{w}(\lambda) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{N} \lambda_i \lambda_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$subject\ to\ \lambda_i \geq 0, \sum_{i=1}^{N} \lambda_i y_i = 0$$

$$\mathbf{w} = \sum_{i=1}^{N} \lambda_i y_i \Phi(\mathbf{x}_i)$$

Optimal hyperplane in the new space

- When classifying **NEW EXAMPLES** (note it's still a summation over support vectors):

Classify new example z as sign(f), i.e. class 1 if f>0 and class -1 if f<0:

Original, without kernel function

$$f = \mathbf{w} \cdot \mathbf{z} + b = \sum_{i=1}^{N} \lambda_i y_i (\mathbf{x}_i \cdot \mathbf{z}) + b$$

Dot product of the new vector and the support vectors

With kernel function

$$f = \mathbf{w} \cdot \mathbf{z} + b = \sum_{i=1}^{N} \lambda_i y_i K(\mathbf{x}_i, \mathbf{z}) + b$$

Kernel function of the new vector and the support vectors

Furthermore

- Kernels have been designed for strings, trees and non-numeric data.
- Kernalisation can be applied to all algorithms that can be re-formulated to work only with dot products (this is then replaced with a kernel function) e.g. k-nearest neighbour etc.

SVM Summary and Comparison

- Linear SVM - no kernel function, finds the optimal linear decision boundary between the positive and negative examples in the ORIGINAL feature space.
- Non-Linear SVM - uses kernel function, finds the optimal linear boundary in the NEW space.
  - This boundary when mapped back to the original feature space corresponds to a non linear decision boundary between positive and negative examples.
  - Scales well in high dimensions.
  - Multi-class problems need to be transformed into 2-class problems.
- Perceptrons: simple and efficient training algorithm but can only learn linear decision boundaries.
- Backpropagation NNs: hard to train with too many parameters, gets stuck in local minima, but can learn non-linear boundaries.
- SVM: relatively efficient training algorithm that can learn non-linear decision boundaries.

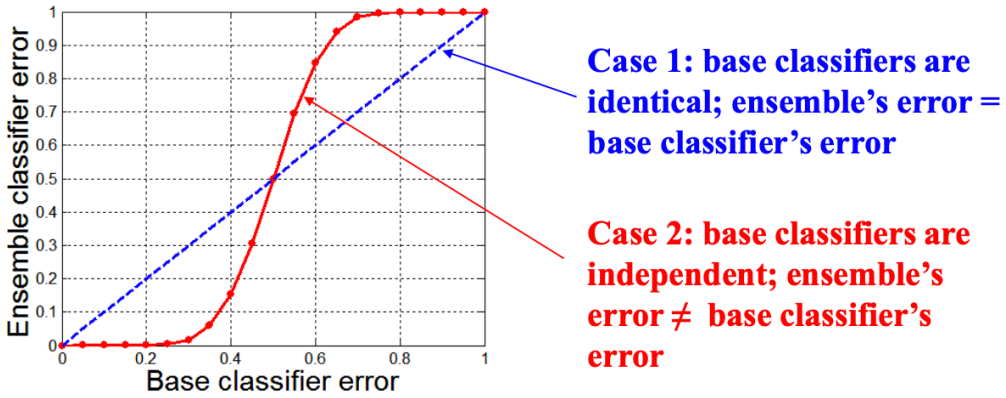
Ensemble of Classifiers

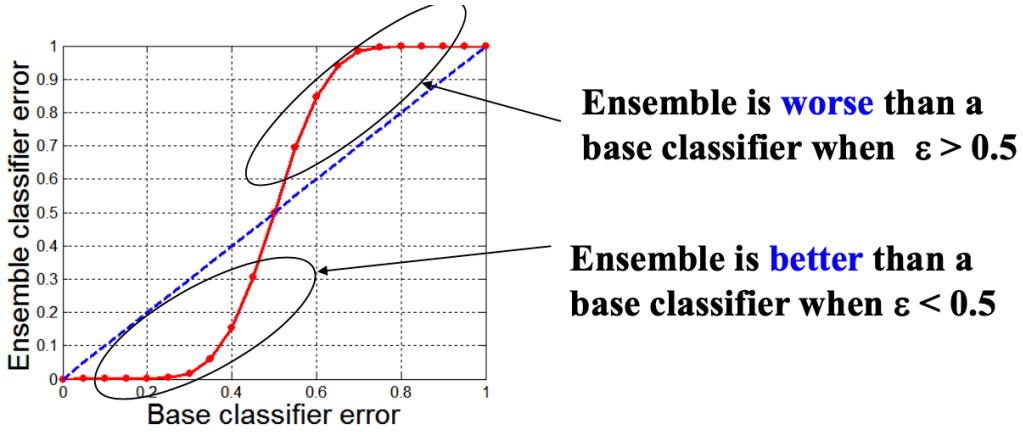
- An ensemble of classifiers combines the predictions of multiple classifiers (called base classifiers) in some way to classify new instances i.e. a **committee** of classifiers.
    - An ensemble of 50 backpropagation NNs, all trained on the same dataset but with different parameters, architecture, initialisation, learning rate etc.
    - To classify a new example, the individual predictions are combined by taking the majority vote.
- In real life:
    - If a certain medical diagnosis occurs more than the others, he chooses it as the final diagnosis i.e. the majority vote (intuition behind bagging).
    - Instead of trusting equally, weights are assigned to the value of each diagnosis, based on the accuracy of past diagnoses. The final one is a weight combination (intuition behind boosting).

Motivation Behind Ensembles

- Combining multiple classifiers helps when:
    - The base classifiers do not make the same mistakes i.e. they complement each other, each is an expert in a part of the domain where the others don't perform well.
    - Each base classifier is reasonably accurate.
- Given 25 base classifiers, binary classification task, an error rate of $\epsilon = 0.35$ on the test set for each base classifier, and majority voting:
    - If base classifiers are identical and make the same mistakes, the error rate for the ensemble is $\epsilon = 0.35$.
    - If the base classifiers are independent (errors not correlated), a new example will be misclassified if more than half of the base classifiers predict incorrectly. Mathematically, however, the error is much less than the individual error rate:

$$e_{\text{ensemble}} = \sum_{i=13}^{25} \binom{25}{i} \epsilon^i (1-\epsilon)^{25-i} = 0.06$$





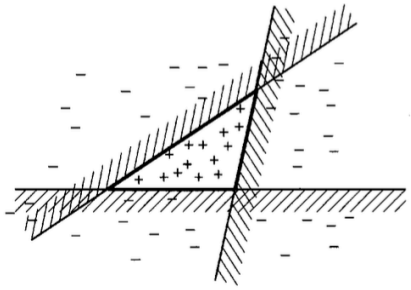
- In essence, if the base classifier is better than a random guess, an ensemble will perform better.

Summary of Conditions

- Ensembles perform better than a single classifier when:
  - The base classifiers are good enough/highly correct i.e. better than random guessing.
  - The base classifiers are independent of each other. Although this is difficult in practice, good results have been achieved with slightly correlated classifiers.

Ensemble Enlarges Hypothesis Space

- Ensembles do well because they are able to learn much more expressive hypotheses without much additional computational expense compared to an individual classifier.



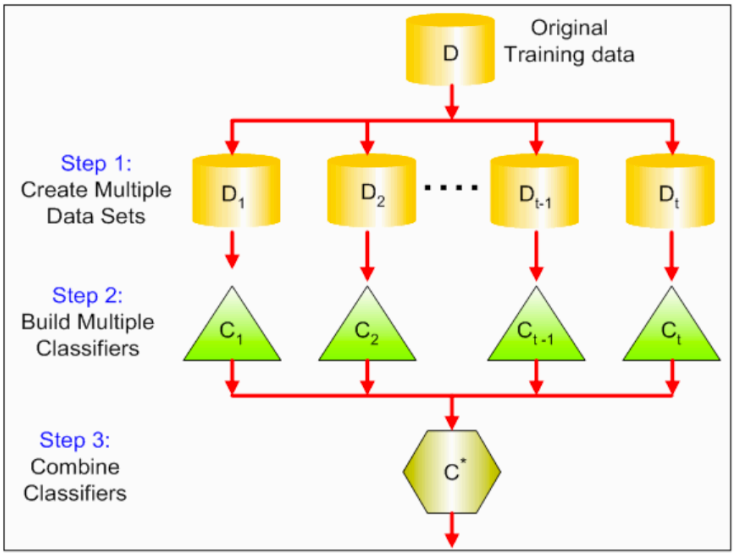
Example:

• 3 linear binary hypotheses, e.g. generated by 3 perceptrons

• An ensemble of 3 perceptrons can learn the resulting triangular region - a hypothesis not expressible in the original hypothesis space (of the perceptron)

Methods for Creating Ensembles

- Focus on generating disagreement among base classifiers by:
  - Manipulating the **training data** - creating multiple training sets by resampling the original data and creating a classifier for each training set i.e. Bagging and Boosting



- Manipulating the **attributes** - using a subset of input features i.e. Random Forest and Random Subspace.
- Manipulating the **class labels** - e.g. output coding.
- Manipulating the learning algorithm - e.g. building a set of classifiers with different parameters.

Bagging - Bootstrap Aggregation

- Create M bootstrap samples.
  - Given a dataset D with n examples, a bootstrap sample D' contains n examples, randomly chosen from D with replacement.
  - On average, 63% of examples in D will also appear in D'.

Dataset with 10 examples:

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

- Use each sample to build a classifier.
- To classify a new example: get the predictions of each classifier and combine them with a majority vote.

```
model generation
Let n be the number of instances in the training data.
For each of M iterations:
  Sample n instances with replacement from training data.
  Apply the learning algorithm to the sample.
  Store the resulting model (classifier).

classification
For each of the M models:
  Predict class of testing instance using model.
Return class that has been predicted most often.
```

When is Bagging Useful?

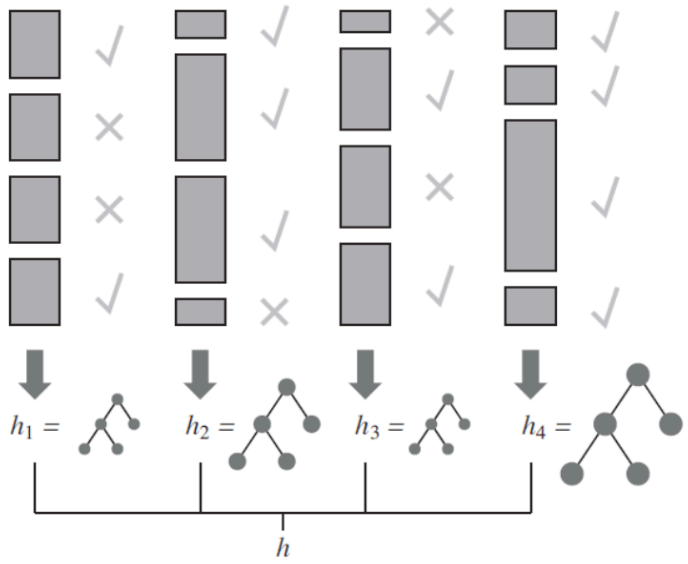
- Typically performs significantly better than the single classifier and is never substantially worse.
- Effective for unstable classifiers, in which small changes in the training set result in large changes in predictions e.g. DTs, neural networks.
- Bagging to numerical prediction/regression:
  - Individual predictions are averaged.
  - Shown theoretically that averaging over multiple models always reduces the expected value of the mean-squared error (not proved for classification).

Boosting

- The most widely used ensemble method, the idea is to make classifiers complement each other.
  - The next classifier should be created using examples that were difficult for the previous classifiers.
- This is achieved by using a weighted training set, where higher weights are examples that have proven difficult to classify previously, and thus should be selected with higher probability.

Algorithm

- Set equal weights for all example e.g. $1/m$ where m is the number of training examples.
- Generate first classifier $h_1 \rightarrow$ we want the next classifier to develop complementary expertise by classifying the misclassified examples.
- Increase the weights of misclassified and decrease the weights of correctly classified examples.
- There is a mechanism for selecting examples for the training set of the next classifier such that higher weights are more likely to be selected $\rightarrow$ generate h_2.
- Continue until K classifiers are generated, and combine using a weighted vote based on how well each classifier performed on the training set e.g. h_4 would have a higher weight vote than h_1.



1 rectangle corresponds to 1 example
The height of the rectangle corresponds to the weight of the example
√ and X show how the example was classified by the current hypothesis (classifier)
The size of the DT corresponds to the weight of that hypothesis in the final ensemble

AdaBoost Algorithm

AdaBoost ensemble generation

Input: the training set, T, consisting of m examples; and the user's choice of the induction technique

1. Let $i = 1$. For each $\mathbf{x}_j \in T$, set $p_1(\mathbf{x}_j) = 1/m$.
2. Create subset T_i consisting of m examples randomly selected according to the given probabilities. From T_i, induce C_i.
3. Evaluate C_i on each example, $\mathbf{x}_j \in T$.
   Let $e_i(\mathbf{x}_j) = 1$ if C_i misclassified $\mathbf{x}_j$ and $e_i(\mathbf{x}_j) = 0$ otherwise.
   i) Calculate $\epsilon_i = \sum_{j=1}^{m} p_i(\mathbf{x}_j) e_i(\mathbf{x}_j)$;
   ii) Calculate $\beta_i = \epsilon_i / (1 - \epsilon_i)$
4. Modify the probabilities of correctly classified examples by $p_{i+1}(\mathbf{x}_j) = p_i(\mathbf{x}_j) \cdot \beta_i$
5. Normalize the probabilities to make sure that $\sum_{j=1}^{m} p_{i+1}(\mathbf{x}_j) = 1$.
6. Unless a termination criterion has been met, set $i = i + 1$ and go to 2.

until K hypotheses (classifiers) have been generated

- Example Probability Recalculation ($m = 10$)
  - Initial Probabilities: $p(x_i) = 1/m = 1/10 = 0.1$

$p_1(\mathbf{x}_1)$	$p_1(\mathbf{x}_2)$	$p_1(\mathbf{x}_3)$	$p_1(\mathbf{x}_4)$	$p_1(\mathbf{x}_5)$	$p_1(\mathbf{x}_6)$	$p_1(\mathbf{x}_7)$	$p_1(\mathbf{x}_8)$	$p_1(\mathbf{x}_9)$	$p_1(\mathbf{x}_{10})$
0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

  - Training set T_1 created using probability distribution above and sampling with replacement. C_1 created, getting the last three wrong.
    - Weighted Error of Classifier: $\epsilon_1 = \sum_{j=1}^{10} p_1(x_j) \cdot e_1(x_j) = 0.3$ where $e = 1$ for incorrect, 0 otherwise.
    - β for Weight Modification: $\beta_1 = (\epsilon_1 / 1 - \epsilon_1) = 0.43$.
  - New probabilities modified with $p(x_j) = p(x_j) \cdot \beta_1$.

$p_2(\mathbf{x}_1)$	$p_2(\mathbf{x}_2)$	$p_2(\mathbf{x}_3)$	$p_2(\mathbf{x}_4)$	$p_2(\mathbf{x}_5)$	$p_2(\mathbf{x}_6)$	$p_2(\mathbf{x}_7)$	$p_2(\mathbf{x}_8)$	$p_2(\mathbf{x}_9)$	$p_2(\mathbf{x}_{10})$
0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.1	0.1	0.1

  - After normalisation done by dividing probability by sum of all probabilities.

$p_2(\mathbf{x}_1)$	$p_2(\mathbf{x}_2)$	$p_2(\mathbf{x}_3)$	$p_2(\mathbf{x}_4)$	$p_2(\mathbf{x}_5)$	$p_2(\mathbf{x}_6)$	$p_2(\mathbf{x}_7)$	$p_2(\mathbf{x}_8)$	$p_2(\mathbf{x}_9)$	$p_2(\mathbf{x}_{10})$
0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.17	0.17	0.17

Property of AdaBoost - Important Property

- If the base learning algorithm L is a weak learning algorithm, then AdaBoost will return an ensemble that classifies the training data perfectly for large enough K.
- A weak learner is a classifier which has a classification performance slightly better than random guessing.
- AdaBoost boosts the weak learning algorithm into a strong one on the training data, independent of how expressive the original hypothesis space is, or how complex the function being learnt is.

AdaBoost – Example 1

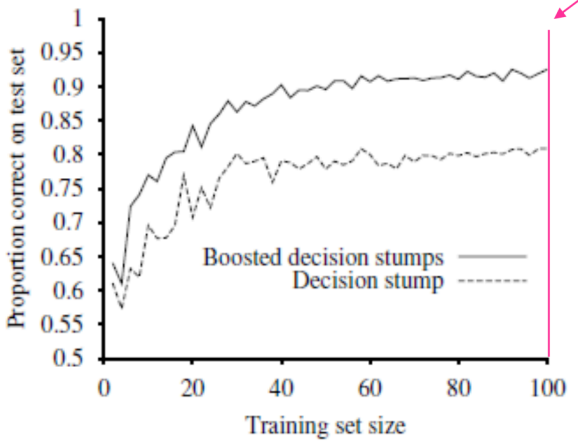
Ensembles typically perform better than individual classifiers

Example: boosted decision stumps (AdaBoost ensemble combining 5 decision stumps) vs 1 decision stump

- **Decision stump = 1-level DT (1 test node only)**

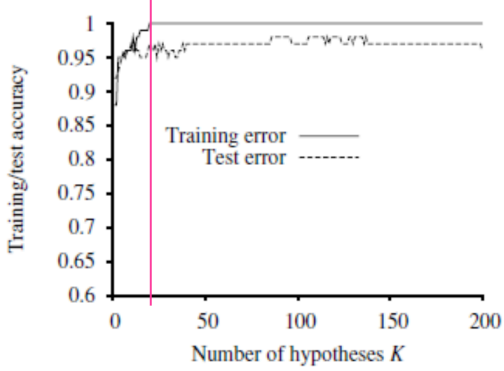
E.g. compare the performances at training set size =100 examples:

- **1 decision stump: 80% accuracy on test set**
- **Ensemble of 5 decision stumps: 93% accuracy on test set**



AdaBoost – Example 2

- **Accuracy vs number of hypotheses K (for a set of 100 training examples)**
- **As we add more base classifiers (decision stumps), the accuracy on the training data increases and reaches 100% for K=20 (consistent with the Boosting Theorem – a perfect fit on the training data is achieved)**
- **As we keep adding more base classifiers:**
  - **The accuracy on the training set remains 100%**
  - **But the accuracy on the test data continues to increase after the accuracy on training data has reached 100%!**
    - **e.g. K=20, accuracy on test set=95%; K=137, accuracy on test set=98%**



- **This result was found across different data sets and classifiers**
- **Contradiction! Ockham's razor – don't make the hypothesis more complex than necessary as this will lead to overfitting**
- **Here - accuracy improves as the ensemble gets more complex**
- **Various explanations, e.g. adding base classifier allows the ensemble to be more definitive in distinguishing between positive and negative ex.**

- It was shown that boosting on new data only fails if the individual classifiers are too complex for the training data size, or their training error become too large quickly.
  - Boosting allows you to build a very powerful combined classifier from very simple ones e.g. simple

DTs generated by 1R.

- Variations: LogitBoost, Gradient Boosting - adds a new model that minimises the error of the previous model.

Bagging and Boosting Comparison

Similarities

- Uses voting for classification and averaging for prediction to combine the outputs of individual learners.
- Combine classifiers of the same type e.g. DTs.

Differences

- The ensemble members are built separately in bagging; they are built iteratively in boosting (influenced by performance of previous ensemble members).
  - A new ensemble member is encouraged to become an expert that complement each other i.e. for examples incorrectly classified by previous ensemble members.
- Combing the opinions of individual ensemble members:
  - Bagging - equal weighting (all experts equally influential).
  - Boosting - weighed based on performance (i.e. some more influential than others).

Random Forest

Given:
n - number of training examples, m – number of all features, k – number of features to be used by each ensemble member (k<m), M – number of ensemble members (trees)

Create Random Forest of M trees:
For each of M iteration
1. Bagging – generate a bootstrap sample
Sample n instances with replacement from training data
2. Random feature selection for selecting the best attribute
Grow a decision tree without pruning. At each step select the best feature to split on by considering only k randomly selected features and calculating information gain.

Classification:
Apply the new example to each of the M decision trees starting from the root. Assign it to the class corresponding to the leaf. Combine the decisions of the individual trees by majority voting.

Discussion

- Performance depends on the accuracy of the individual trees and their correlation with each other.
  - Ideally, we would like highly accurate individual trees with little correlation.
- Bagging and random feature selection used to generate diversity and reduce correlation.
- As the number of features k increases, both the strength and correlation increase.
  - Rule of Thumb: $k = \log_2 m$ where m is number of features.
- Random Forest typically outperforms a single DT.
- Random Forests have been proven to not overfit.
- Random Forests are faster than AdaBoost and give comparably accurate results.

Lecture 11 - Bayesian Networks

Probabilistic Reasoning

- Random Variables
  - Domain of a RV e.g. Domain of Fair Dice = $\{H, T\}$
- Event (Proposition)
- Probability $P(A=a)$
  - Joint Probability $P(A=a, B=b)$
- Conditional Probability $P(A=a|B=b)$
- Probability Distribution $P(\text{Meningitis}) = <0.3,0.7>$
  - Joint Probability Distribution Table of Two Variables - k^N entries in table given N variables, k values per variable

Vomiting	Meningitis	P(Vomiting, Meningitis)
T	T	2/10
T	F	4/10
F	T	1/10
F	F	3/10

- Axioms of Probability
  - $P(A) \in [0,1]$
  - $P(T)=1$ and $P(F)=0$ i.e. valid and invalid propositions
  - $P(A \or B) = P(A) + P(B) - P(A \and B)$
- Theorems of Probability
  - $P(\neg A)= 1-P(A)$
  - For a RV X with k different values $x_1,...,X_k$: $P(X=x_1) + ... + P(X=x_k) = 1$ i.e. probability distribution of a single variable must sum to 1.
- Given the full joint probability table, we can compute the probability of any event in the domain by summing over the cells (atomic events) where the event is true.
  - This is called marginalisation, or summing out, because the variables that are not in question were summed out.

Conditional Probability

- $P(a|b) = {P(a,b) \over P(b)}$ given that $P(b) \neq 0$.
- Therefore, $P(a,b) = P(a|b)P(b) = P(b|a)P(a)$ with the commutative property.
- Chain rule can be used such that $P(a,b,c,d) = P(a|b,c,d)P(b|c,d)P(c|d)P(d)$.

$$P(\neg cavity \mid toothache) = \frac{P(\neg cavity, toothache)}{P(toothache)} =$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = \frac{0.08}{0.2} = 0.4$$

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

$$P(cavity \mid toothache) = \frac{P(cavity, toothache)}{P(toothache)} =$$
$$= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = \frac{0.12}{0.2} = 0.6$$

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

Good - 0.6 and 0.4 sum up to 1 as expected!

P(Cavity| toothache) = <0.6, 0.4>

Normalisation

- Since the denominators are the same, we can view it as a normalisatoin constant α.

$$P(cavity \mid toothache) = \alpha P(cavity, toothache) =$$
$$= \alpha * (0.108 + 0.012) = \alpha * 0.12$$
$$P(\neg cavity \mid toothache) = \alpha P(\neg cavity, toothache) =$$
$$\alpha * (0.016 + 0.064) = \alpha * 0.08$$

- The probability distribution with α:

$$\mathbf{P}(Cavity \mid toothache) = \alpha < 0.12, 0.08 >$$

- Computing the normalisation constant:

$$\alpha = 1/(0.12+0.08) = 1/0.2$$

- Final probability distribution without α:

$$\mathbf{P}(Cavity \mid toothache) =< 0.12/0.2, 0.08/0.2 >=< 0.6, 0.4 >$$

- Alternatively:

$$\begin{aligned}
&\mathbf{P}(Cavity \mid toothache) = \alpha \mathbf{P}(Cavity, toothache) = \\
&= \alpha[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)] = \\
&= \alpha[< 0.108, 0.016 > + < 0.012, 0.0064 >] = \\
&= \alpha < 0.12, 0.08 >= \quad < 0.12/(0.12 + 0.08), 0.08/(0.12 + 006) >= \\
&=< 0.6, 0.4 >
\end{aligned}$$

Inference by Enumeration Using Full Joint Distribution

General rule:

- **X – query variable (e.g. Cavity)**
- **E – evidence (e.g. Toothache)**
- **e – the observed values for E (e.g. true)**
- **H – the remaining (*hidden*) variables: H = X - E**

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

The summation is over the values of the hidden variables, i.e. we are *summing out* the hidden variables

$$\begin{aligned}
&\mathbf{P}(Cavity \mid Toothache = true) = \alpha \mathbf{P}(Cavity \mid Toothache = true) = \\
&= \alpha \sum_{h=true, false} \mathbf{P}(Cavity, Toothache = true, Catch = h) \\
&= \alpha[\mathbf{P}(Cavity, Toothache = true, Catch = true) + \\
&\quad \mathbf{P}(Cavity, Toothache = true, Catch = false)]
\end{aligned}$$

- This doesn't scale well:
  - For n Boolean variables, it requires a joint probability table of size $O(2^n)$ and $O(2^n)$ time to process it.
  - Impractical for real problems with hundreds of random variables - sum over too much.

Independence

- Two events A and B are independent if:

$$P(A, B) = P(A) \times P(B)$$

- This is equivalent to:

$$P(A|B) = P(A) \text{ and } P(B|A) = P(B)$$

- Indepedence is domain knowledge and makes the inference easier.

Independence - Example

product rule:
$P(A,B) = P(A|B)P(B)$

- **Dental example: Suppose that we have 1 more binary variable in the dental example: Weather with values *cloudy* and *sunny*. Task – compute:**

$$P(Weather = cloudy, toothache, catch, cavity) =$$

product rule

$$= P(Weather = cloudy \mid toothache, catch, cavity)P(toothache, catch, cavity)$$

$$= P(Weather = cloudy)$$

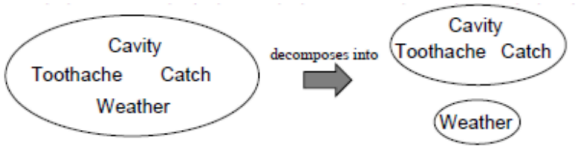
- **However, a person's dental problems do not influence the weather**

$$\Rightarrow P(Weather = cloudy)|toothache, catch, cavity) = P(Weather = cloudy)$$

independence rule:
$P(A|B) = P(A)$

$$\Rightarrow \quad P(Weather = cloudy, toothache, catch, cavity) =$$
$$= P(Weather = cloudy)P(toothache, catch, cavity)$$

⇒ **Instead of storing a table with 2^4 entries, we can store $2^3 + 2^1$. This also reduces the complexity of the inference problem.**



Conditional Independence

- Absolute independence is a very useful property but it is rare in practice. However, they are often conditionally independent.
  - The asserted statement below is because if I have a cavity, the probability that I have a toothache doesn't depend on whether the dentist catches my tooth.

Similarly, based on domain knowledge we can assert that Toothache is conditionally independent of Catch given Cavity:

$$\mathbf{P}(Toothache \mid Catch, Cavity) = \mathbf{P}(Toothache \mid Cavity) \quad (1)$$

Let's see how conditional independence simplifies inference:

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

chain rule

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

from (1)

So we need $2^2+2^2+2^1=10$ probability numbers or 5 independent numbers (as counterparts sum to 1) which is less than without conditional independence

In most cases, conditional independence reduces the size of the representation of the joint distribution from $O(2^n)$ to O(n)

- **n is the num. of variables conditionally independent, given another variable**

=> Conditional independence assertions: 1) allow probabilistic systems to scale up and 2) are more available than absolute independence assertions

Bayes Theorem for Reasoning

- $P(a, b) = P(a|b)P(b)$ i.e. $P(a, b) = P(b, a) = P(b|a)P(a)$ given commutativity.
- Equating right hand sides give:

$$P(a \mid b) = \frac{P(b \mid a)P(a)}{P(b)}$$ **Bayes Theorem**

$$\mathbf{P}(A \mid B) = \frac{\mathbf{P}(B \mid A)\mathbf{P}(A)}{\mathbf{P}(B)}$$ **<- More general form for multivalued variables using the P notation**

$$\mathbf{P}(A \mid B) = \alpha\mathbf{P}(B \mid A)\mathbf{P}(A)$$ **<- General form with α**

Answering Q1

After a yearly checkup, a doctor informs their patient that he has both bad news and good news. The bad news is that the patient has tested positive for a serious disease and that the test that the doctor has used is 99% accurate (i.e., the probability of testing positive when a patient has the disease is 0.99, as is the probability of testing negative when a patient does not have the disease). The good news, however, is that the disease is extremely rare, striking only 1 in 10,000 people.

Q1. What is the probability that the patient has the disease?

Variables: D – hasDisease {true, false) , T – positiveTest {true, false}

$P(d \mid t) = ?$ **This is a shorthand for:** $P(D = true \mid T = true) = ?$

$$P(d \mid t) = \frac{P(t \mid d)P(d)}{P(t)} = \frac{0.99 * 0.0001}{P(t)}$$

To calculate P(t) we will use the *Theorem of Total Probability*

Theorem (Rule) of Total Probability

- **The unconditional probability for any event X is:**

$$P(X) = \sum_i P(X \mid Y_i)P(Y_i)$$

where Y is another event and Y_i are all possible outcomes for Y

- **This is actually the summing out rule that we used before!**

$$P(t) = P(t \mid d)P(d) + P(t \mid \neg d)P(\neg d) =$$
$$= 0.99 * 0.0001 + 0.01 * 0.9999 = 0.0101$$

$$P(d \mid t) = \frac{P(t \mid d)P(d)}{P(t)} = \frac{0.99 * 0.0001}{0.0101} = 0.0098$$

=> The probability to have the disease, given that the test is positive is 0.98%. This is very low: <1%.

Answering Q2

Q2. Why is the rarity of the disease good news given that the patient has tested positive for it?

Because P(d|t) is proportional to P(d), so a lower P(d) means a lower P(d|t)

$$P(d \mid t) = \frac{P(t \mid d)P(d)}{P(t)} = \frac{0.99 * 0.0001}{0.0101} = 0.0098$$

In other words, the Bayes theorem explicitly includes the prior probability of an event when calculating the likelihood of that event based on evidence

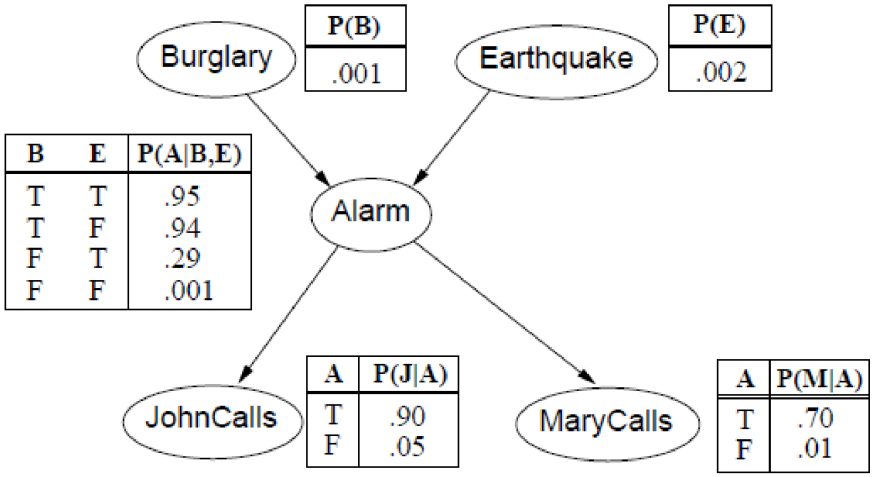
Hence, for our example, when the disease is much rarer than the accuracy of the test, a positive test result does <u>not</u> mean the disease is likely

Summary

- Probability is a rigorous formalism for representing uncertain knowledge.
- Given the full joint probability distribution, we can compute the probability of any event in the domain by summing over the cells (atomic events) where the event is true.
    - The full joint distribution is usually too large to store and use for reference.
- Independence and conditional independence allow probabilistic systems to scale up.
- We also saw how to use several rules and theorems for probabilistic reasoning e.g. conditional probability, product rule, chain rule, Bayes thereom, theorem of total probability.

Bayesian Networks

- To do probabilistic reasoning, we need to know the joint probability distribution of the variables in the domain.
    - If we have N binary variables, we need 2^N numbers to specify the joint probability distribution. We may not need all values if variables are conditionally/fully independent.
- Bayesian networks are graph-based models that encode structural relationships between variables such as direct influence and conditional independence - more compact than full join probability distribution.
    - Directed Acyclic Graph composed of nodes (one per variable), link (denotes direct influence/dependence between two nodes), conditional probability tables (CPT) for each node, give its parents.



- Each row in a CPT sums to 1 $\rightarrow$ for a binary variable X, we can similarly state the probability for $P(X=T|G)$. $P(X=F | G)$ is just 1 minus.

Compactness of Bayesian Networks

Consider a domain with *n* Boolean variables

We have created a Bayesian network

The CPT for a variable X with *k* parents will have 2^k rows

- **1 row for each combination of parent values, e.g. Alarm has 2 parents and will have 4 rows**
- **Each row has 1 number *p* for X=*true*, the number for X=*false* is 1-*p***

If each variable has at most *k* parents, the complete Bayesian network requires O(n* 2^k) numbers, i.e. it grows linearly with *n*

For comparison, the full joint distribution table requires O(2^n) numbers which is much bigger – grows exponentially with *n*

For our burglary example: 1+1+4+2+2=10 numbers in the CPT of the Bayesian net vs 2^5=32 in the full joint distribution table

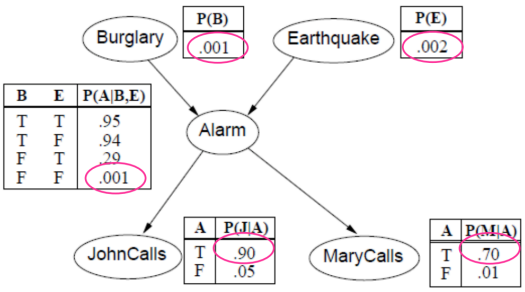
Joint Probability Rule

- Given a Bayesian network with n nodes, the probability of an event $x_1, x_2,...,x_n$ can be computed as:

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

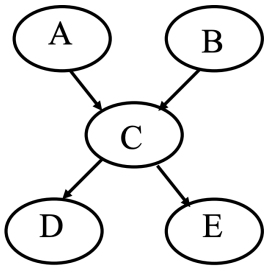
- **Example for the burglar net:**

P(j, m, a, ~b, ~e) =
= P(j|a) P(m|a) P(a|~b,~e) P(~b)P(~e) =
= 0.9*0.7*0.001*(1-0.001)*(1-0.002) =
= 0.006281



- **Notation: As before, j is a shorthand for JohnCalls=T, and ~j for JohnCalls=F**

- Why do we multiply probabilities together? What is the assumption made in Bayesian networks that allow us to do so?
  - **Bayesian Network Assumption**: a node is conditionally independent of its non-descendents (grandparents, great-grandparents, siblings etc NOT children, nor parents), given its parents i.e. if the values of its parents are known.



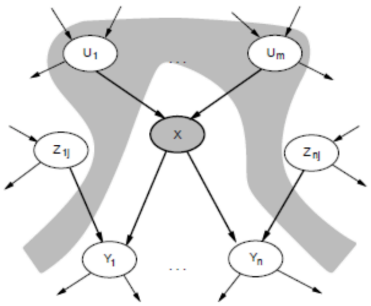
- **D is conditionally independent of A, given C**
- **D is conditionally independent of B, given C**

- **E is conditionally independent of A, given C**
- **E is conditionally independent of B, given C**

- **D is conditionally independent of E, given C**
- **E is conditionally independent of D, given C**

- **A is independent of B**

P(node | parents plus any other non-descendants)=P(node | parents)



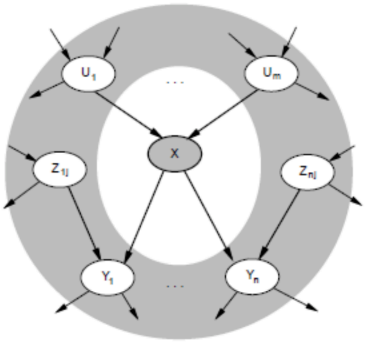
X is conditionally independent of its non-descendants (nodes Z_{ij}) given its parents (nodes U_i – the gray area)

- This is to say that the calculation of joint probabilities reduces to a multiplication of conditional probabilities from the BN.

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | x_{i-1}, \ldots, x_1) = \prod_{i=1}^{n} P(x_i | Parents(x_i))$$

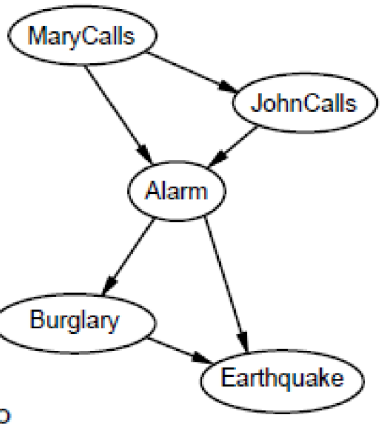
Markov Blanket

- Another important independence property: A node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents.
    - Given the grey area, X is conditionally indepdent of all other nodes in the network.



Construction of Bayesian Network 11, 48 by Domain Experts

Suppose we choose the ordering: M, J, A, B, E, A



$P(J|M) = P(J)$? No
$P(A|J, M) = P(A|J)$? $P(A|J, M) = P(A)$? No
$P(B|A, J, M) = P(B|A)$? Yes
$P(B|A, J, M) = P(B)$? No
$P(E|B, A, J, M) = P(E|A)$? No
$P(E|B, A, J, M) = P(E|A, B)$? Yes

Inference by Enumeration 11,54-63

- We can compute any conditional probability $P(X|E)$, and thus perform inference.
    - E.g. $P(b|j, m)$

Inference by Variable Elimination 11, 63-72

- The enumeration algorithm has poor space complexity and is inefficient due to space complexity issues.
    - We can improve this by doing the variable elimination algorithm, which does the calculation once and saves the results for later use.
    - The expression is evaluated right-to-left and the tree is evaluated bottom up.
    - The results are stored, the summations over each variable are done only for those portions of the

expression that depend on the variable.

$$P(b \mid j,m) = \alpha \sum_{ea} P(b)P(j \mid A_a)P(m \mid A_a)P(E_e)P(A_a \mid b,E_e)$$

Approximate Inference

Compared to enumeration, variable elimination allows us to save computation by re-using intermediate results but it is still computationally expensive

Alternative: *approximate inference by sampling* – idea:

- **Generate many samples – 1 sample is a complete assignment of all variables**
- **Count the fraction of samples matching the query and evidence**
- **If the number of samples is big enough, the fractions converge to P(query | evidence)**

There are 3 main sampling algorithms

- **Simple (direct) sampling**
- **Likelihood weighting**
- **Gibbs sampling (an example of Markov Chain Monte Carlo method)**

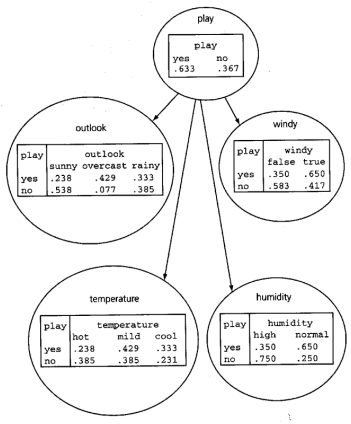
Naive and Bayesian Networks

Naïve Bayes and Bayesian Networks

A Naïve Bayes classifier is a special case of Bayesian networks

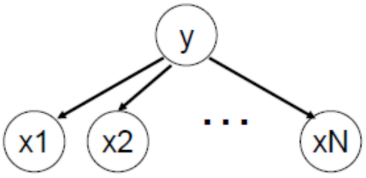
The topology represents the Naïve Bayes assumption of conditional independence - the values of the attributes are conditionally independent given the class value

- ***outlook*, *windy*, *temperature* and *humidity* are independent given *play***



More generally:

- **Class node *y* as a root**
- **Evidence nodes *x* as leaves (features)**
- **Assume conditional independence between features, given the class**



Where Do We Get CPT From?

- Ask domain expert or learn from data 76,77.

Summary

Bayesian networks are graphical models that provide a way to represent conditional independence relationships. Hence, they provide a more compact representation than the full joint probability distribution.

They assume that a node is conditionally independent of its non-descendants, given its parents

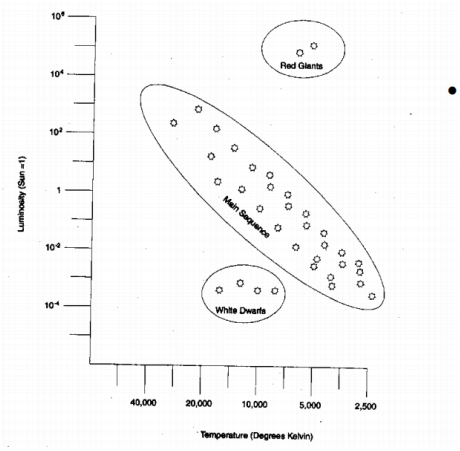
Inference in Bayesian networks means computing the probability distribution of a set of query variables, given a set of evidence variables

We studied two algorithms for inference: inference by enumeration and inference by variable elimination

A Naïve Bayes classifier is a special case of Bayesian networks

Lecture 12 - Unsupervised Learning (Clustering)

- Clustering - the process of partitioning data into a set of groups/clusters so that items from the same cluster are:
  - Similar to each other defined in terms of some distance measure.
  - Dissimilar to the items in other clusters.



- **Ex. Star clustering based on temperature and brightness (Hertzsprung-Russel diagram)**
  - **3 well-defined clusters**
  - **Now we understand that the groups represent stars in 3 different phases of their life**

Image from "Data Mining Techniques", M. Berry, G. Linoff, Wiley.

- Example: fitting troops in clothes that suited different body types rather than ordered set where all dimensions increased together.

Unsupervised Learning

- Given:
  - A set of UNLABELED examples (input vectors) x_i.
  - k - desired number of clusters (may not be given)
- Task: Cluster the examples into k clusters.

Clustering

- Given:
  - A dataset $P = \{p_1, \ldots, p_n\}$ of input vectors.
  - An integer k - number of clusters we are looking for (may not be specified e.g. agglomerative, SOM).
- Find:
  - A mapping $f : P \rightarrow \{1, \ldots, , k\}$ where p_i is assigned to 1 cluster K_j, $1 \leq j \leq k$. Note: some clustering algorithms (fuzzy, probabilistic) assign each example to more than 1 cluster with a certain probability.
- Result:
  - A set of clusters $K = \{K_1, K_2, \ldots, K_k\}$.

Typical Clustering Algorithms

- As a stand-alone tool to group data.
- As a building block for other algorithms e.g. pre-processing tool for dimensionality reduction - using the cluster centre to represent all data points in the cluster (called vector quantisation).
- Marketing (find groups of customers), biology (plant, animal taxonomies), land use (areas of similar land use), insurance (groups with high average claim cost), city-planning (group of houses according to type, value, location).

Taxonomy of Clustering Algorithms

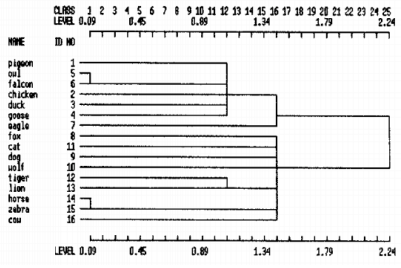
Partitional – k-means, k-medoids, nearest neighbor, SOM

- **Create only one set of clusters**
- **The number of clusters *k* is required for most algorithms (e.g. k-means and k-medoids) and not required for some (k-nearest neighbor and SOM)**

Eagle	Cow
Falcon	Zebra
Owl	Horse
Goose	Lion
Duck	Tiger
Chicken	Cat
Pigeon	Wolf
	Dog
	Fox

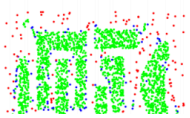
Hierarchical - agglomerative and divisive

- **Creates a nested set of clusters**
- ***k* does not need to be specified**



Density-based - DBSCAN

- **regions of high density form clusters**
- ***k* does not need to be specified directly**



Model-based (generative) – EM

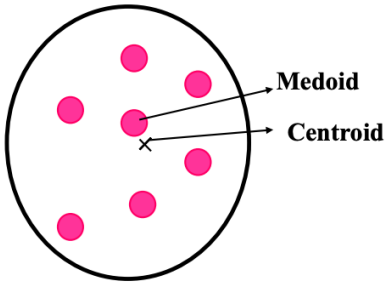
Fuzzy clustering

Cluster Definitions

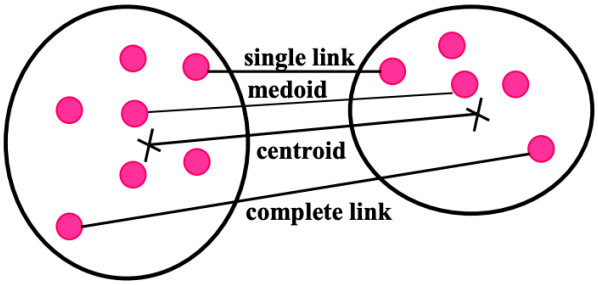
- Consider a cluster K of N points $\{p_1, \ldots, p_N\}$.
- Centroid (means) - the middle of the cluster (doesn't have to be an actual data point).

$$C = \frac{\sum_{i-1}^{N} p_i}{N}$$

- Medoid M - centrally located point in the cluster.



Distance Between Clusters



- Centroid - distance between centroids.
- Medoid - distance between medoids.
- Single link (MIN) - smallest pairwise distance between elements from each cluster.
- Complete link (MAX) - largest pairwise distance between elements from each cluster.
- Average link - average pairwise distance between elements from each cluster.

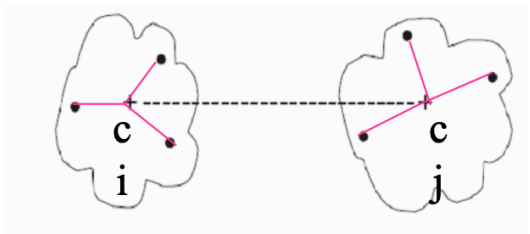
What is a Good Clustering?

- A good clustering will produce clusters with (measured with a distance function):
  - High cohesion (high similarity within the cluster).
  - High separation (low similarity betwen clusters).
- Various ways to combine them into 1 measure.
  - Davies-Bouldin (DB) Index
  - Silhouette Coefficient

Davies-Bouldin (DB) Index

- Heuristic measure of the quality of clustering that combines cohesion and separation.
- Each pair of clusters i and j (from the resulting clustering) are compared in pairs.

$$DB = \frac{1}{k}\sum_{i=1}^{k}\max_{j,j\neq i}\left[\frac{dist(\mathbf{x},c_i)+dist(\mathbf{x},c_j)}{dist(c_i,c_j)}\right]$$



k – number of clusters

c_i and c_j – centroids of clusters i and j

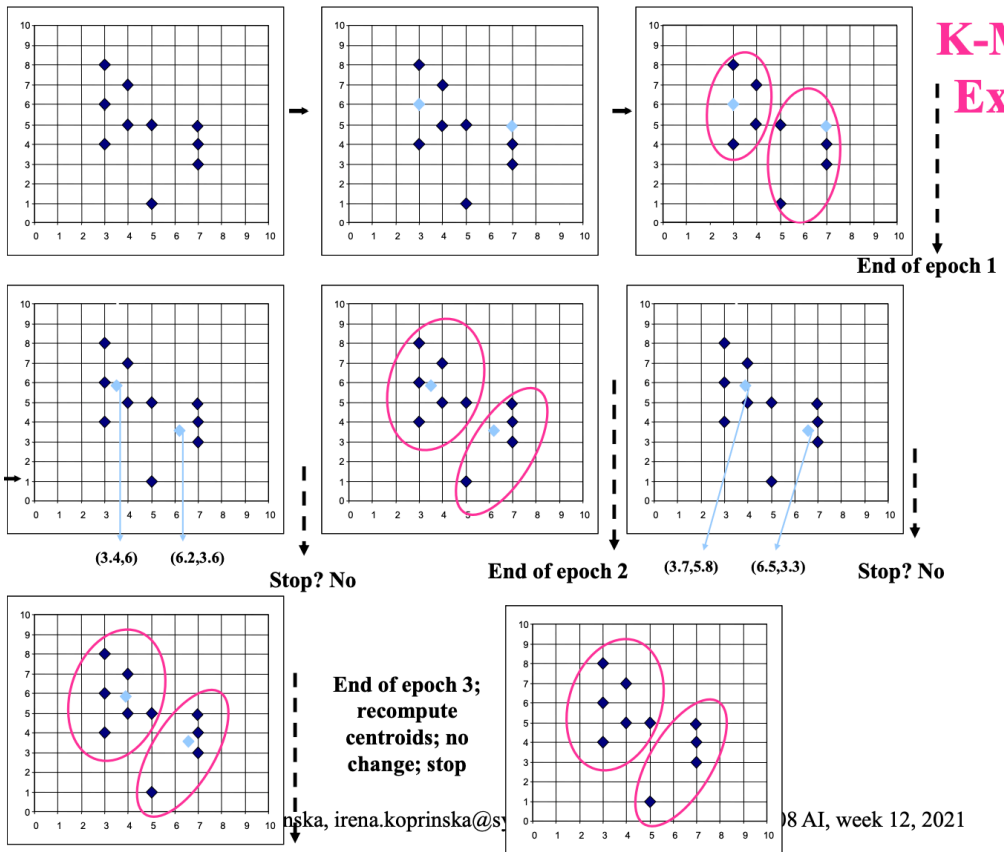
$dist(\mathbf{x}, c_i)$ – mean-squared distance from each item x in cluster i to its centroid

$dist(ci,c_j)$ – distance between the centroids of cluster i and j

- $dist(x, c_i)$ is the spread of items within cluster i, $dist(c_i, c_j)$ is the distance between clusters (centroid distance).
- $\max$ - for each cluster i, find the cluster j that maximises the fraction in brackets:
  - Denominator is low: small distance between centroids of i and j i.e. possibly overlapping clusters i and j AND/OR
  - Numerator is high: big spread within each of the clusters.
- DB index is the of max's (worst pairing for each cluster) and we would like to keep it to a minimum. Small DB index = clusters have small spread and are far from each other.

K-Means Clustering Algorithm

- Partitional clustering algorithm, with number of k specified in advance and only one set of clusters. It is iterative and distance-based.
- Implementation:
  - Choose k examples as the initial centroids (seeds) of the clusters.
  - Form k clusters by assigning each example to the closest centroid.
  - At the end of each epoch:
    - Re-compute the centroid of the clusters.
    - Stopping condition satisfied? Centroid do not change. If yes, stop. Otherwise, repeat 2 and 3 using the new centroids.



- Given a list of items with the distance between each other, cluster into 2 groups using B and C as initial centroids.

Solution

Item	A	B	C	D	E
A	0	1	2	2	3
B	1	0	2	4	3
C	2	2	0	1	5
D	2	4	1	0	3
E	3	3	5	3	0

seed1=B, seed2=C
epoch1 – start:
A:
d(A, seed1)=1
d(A, seed2)=2
=> A ∈ cluster1

B:
B is the seed of cluster1
=> B ∈ cluster1

C:
C is the seed of cluster2
=> C ∈ cluster2

- **D:**
  d(D, seed1)=4
  d(D, seed2)=1
  => D ∈ cluster2

- **E:**
  d(E, seed1)=3
  d(E, seed2)=5
  => E ∈ cluster1

- **End of epoch 1, clusters are:**
- **{A, B, E} and {C, D}**

As an Optimisation Problem

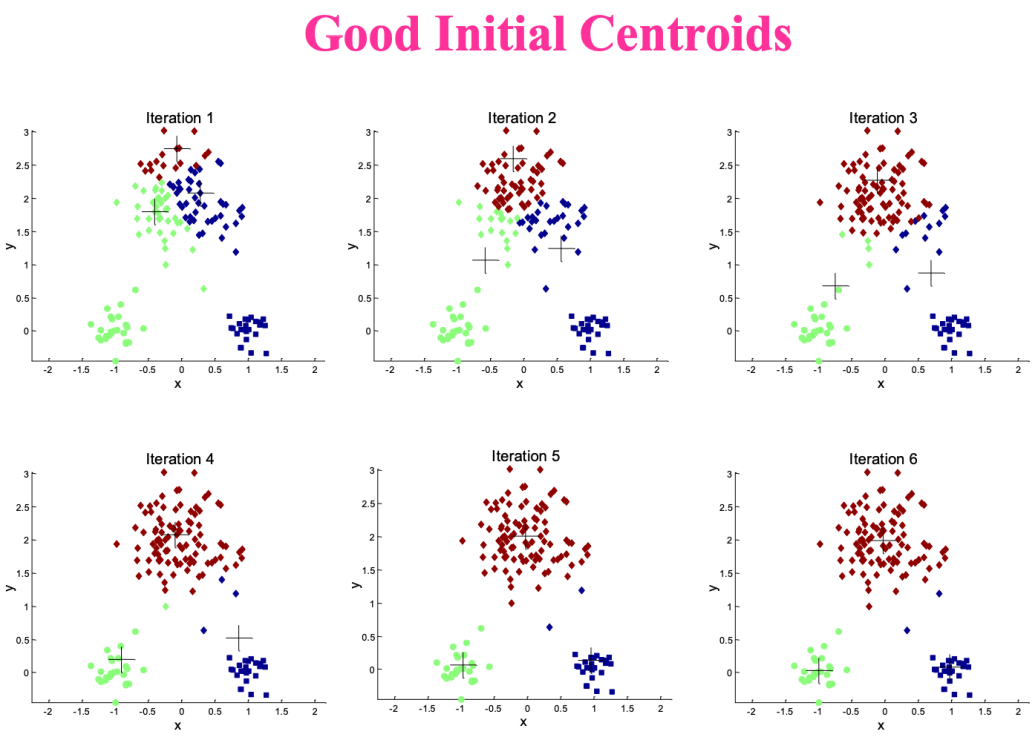
- k-means clustering can be viewed as an optimisation problem, find k clusters that minimise the sum-squared error (SSE), where c_i are centroids, and x are examples.

$$SSE = \sum_{i=1}^{k} \sum_{x \in K_i} dist(c_i, x)^2$$

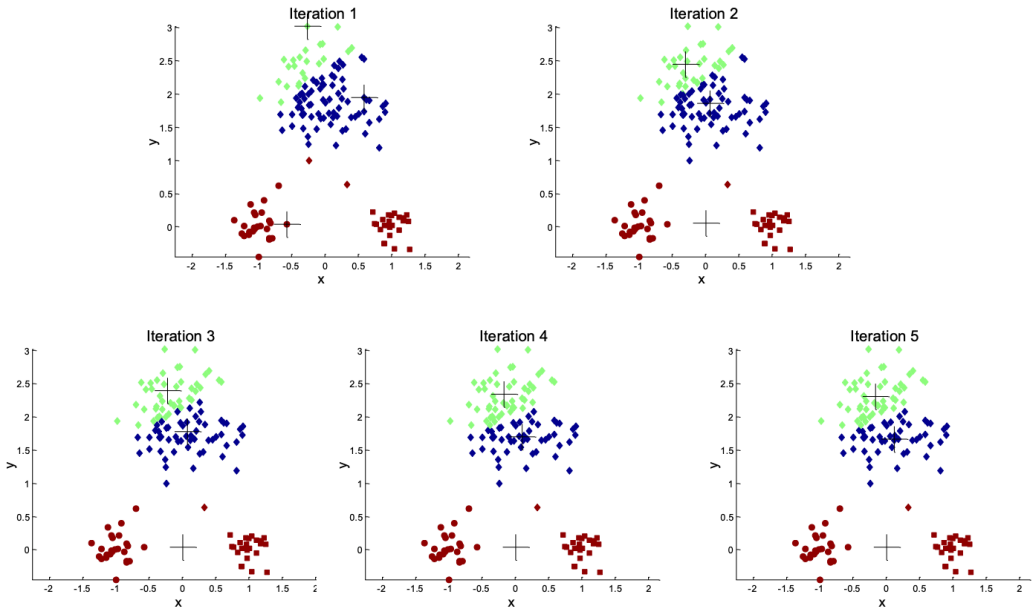
- Not guaranteed to find a global minimum, since it may be stuck in a local minimum.

Discussion of k-Means

- Typical values of k: 2 to 10.
- Different distance measures can be used, typically Euclidian distance.
- Data should be normalised and nominal data needs to be converted into numeric.
- The number of epochs for convergence is typically much smaller than the number of points i.e. quick convergence.
- Often the stopping criterion is changed to 'until relatively few points change clusters' e.g. $< 1\%$.
- **Sensitive to the choice of initial centroid seeds**. Different runs of k-means produce different clusters, although there are several approaches for choosing "good" initial centroids $\rightarrow$ run several times, pick best initialisation based off final clustering.



Poor Initial Centroids



Time and Space Complexity

- Space Complexity - modest
  - All examples and centroids need to be stored. This is done in $O((m+k)n)$ where m is the number of examples, n is the number of features, k is the number of clusters.
- Time complexity - expensive
  - $O(tkmn)$ where t is the number of iterations.
  - Involves finding the distance from each example to each cluster center at each iteration.
  - t is often small and can be safely bounded as most changes occur in the first few iterations.
- NOT practical for large datasets.

Strength and Weaknesses

Strengths

- Simple and very popular
- Relatively efficient, modest space complexity, high time complexity, typically fast convergence with small number of iterations.
- Not sensitive to the order of input examples.

Weaknesses

- Not guaranteed to find the optimal solution and sensitive to initialisation.
- Does not work well for clusters with non-spherical shape, non-convex shape.
- Does not work well with data containing outliers (pre-processing is needed).
- The best number of clusters k is not known in advance.
  - There is more than one correct answer to a clustering problem.
  - Domain expert may be required to suggest a good number of clusters and also to evaluate a solution.

    - Use clustering evaluation methods (DB-index, silhoette coefficient elbow method) to find a good number of clusters.
- Interpreting the semantic meaning of each cluster is difficult.
    - Applies for all clustering algorithms, not only k-means.
    - Why are the items in the same cluster - What are their common characteristics? What are the distinct characteristics in each cluster.
    - Domain expert is needed.

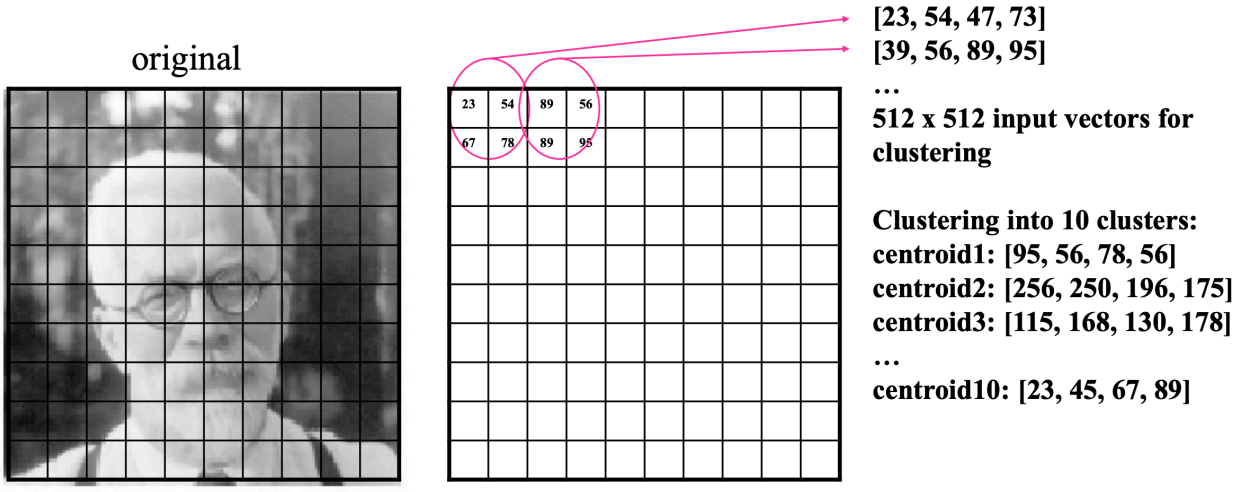
Variations

- Improving the chances of k-means to find the global minimum (different ways to initialise the seed centroids).
    - Using weights based on how close the example is to the cluster centre - Gaussian mixture models.
    - Allowing clusters to split and merge.
        - Split if the variance within a cluster is large.
        - Marge if the distance between cluster centers is smaller than a threshold.
- Make it scale better.
    - Save distance information from one iteration to the next, reducing the number of calculations.
- K-means can be used for hierarchical clustering. Start with $k = 2$ and repeat recursively within each cluster.

k-means for Image Compression

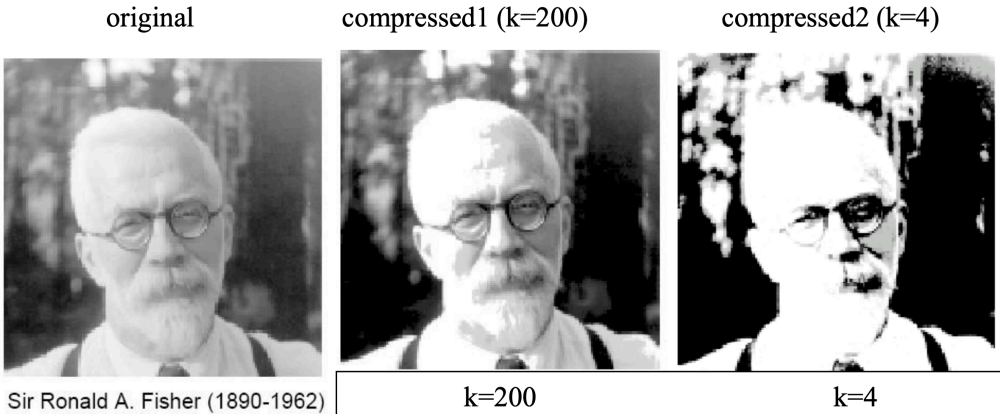
- Compression by Vector Quantisation (VQ) using k-means using an original 1024 x 1024 image, where each pixel contains a greyscale value $[0, 255]$.
    - $1024 \times 1024 \times 1B = 1MB$ needed to store the original image.

- **Break the image into 2x2 blocks of pixels**
- **Each of the 512x512 blocks of 4 numbers is regarded as a vector (4 dim)**
- **Cluster these vectors using k-means (e.g. k=10)**
- **In the compressed image represent each block with the centroid of its cluster (i.e. replace each vector with the centroid)**



- Now there are 10 centroids and greyscale vectors, 40 greyscale values.

- The clustering process is called encoding, centroids are called codewords and the collection of all centroids is scalled a codebook.



- In our example:
  - k centroids with dimensionality 4 = $k \times 4 \times 8bits$
  - If k is small (e.g. in this example), this is a very small storage requirement that can be ignored.
- For each bloke, the index of the closest centroid that will replace its values: $\log_2 k$ bits per block since we need to store k integers, and log that amount to encode them.
- Compression Rate (Compressed/Original) = $(k \cdot 4 \cdot 8 + 512 \cdot 512 \cdot \log_2 k)/(1024 \cdot 1024 \cdot 8)$ bits = 0.063.

k-Medoids Clustering Algorithm

- Similar to k-Means, but reduces sensitivity to outliers by using cluster medoids (centrally located point) instead of cluster means.
  - Also minimises the distance between a point in a cluster and the reference point (medoid instead of means in this case).
- Also called PAM (Partitioning Around the Medoids).

Pseudocode

- Select K points as the initial medoids.
- Repeat:
  - Form K clusters by assigning all items to the closest medoid.
  - Recompute the medoid for each cluster (search for better medoids). In essence, what's below is a hill-climbing search for the set of medoids minimising the distance from an example to a medoid.
    - Initial parent state = current set of medoids.
    - Generate the children states by swapping each medoid with other non-medoids.
    - Evaluate the children states - are they better than the parent based off an evaluation function? Sum of absolute distances between items and closest medoid.
    - Choose the best state (set of medoids with the smallest cost).
- Until medoids don't change or cost doesn't decrease.

Examples

- Given 5 items with pairwise distances, cluster using K-medoids, $k = 2$. Ties are broken randomly if distances are the same.

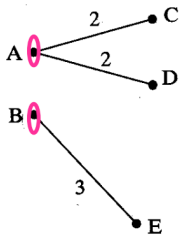
Item	A	B	C	D	E
A	0	1	2	2	3
B	1	0	2	4	3
C	2	2	0	1	5
D	2	4	1	0	3
E	3	3	5	3	0

1) Randomly select 2 items as initial medoids,

- **e.g. A (K1) and B (K2)**

2) Assign all items to the closest medoid:

- **K1= {A,C,D} , K2={B,E}**



Cost=7

3) Consider swapping medoids A and B with each non-medoid

A->C, B=B (swap A with C, don't change B; now the medoids are C and B)

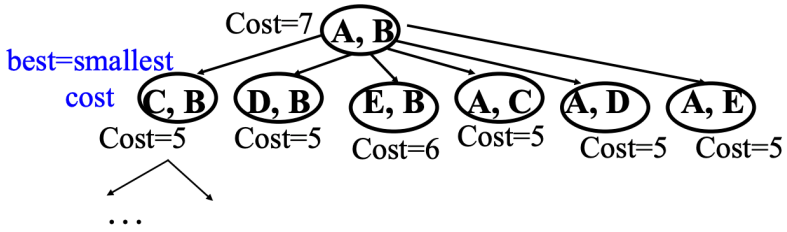
A->D, B=B

A->E, B=B

A=A, B->C

A=A, B->D

A=A, B->E



Given *n* items and *k* desired clusters , how many swaps need to be considered?

k(n-k)

- The following step is repeated for all swaps.

Evaluate child 1 (C, B) – the new medoids are C and B, resulting from the swap: A ->C, B=B

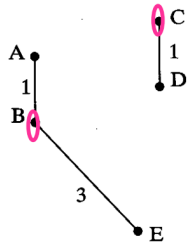
Assign each non-medoid to the closest medoid and calculate the total cost:

- From A to the new medoids: dist(A,C)=2 > dist(A,B)=1

- From D to the new medoids: dist(D,C)=1 < dist(D,B)=4

- From E to the new medoids: dist(E,C)=5 > dist(E,B)=3

cost=1+1+3=5



- The clustering with the best (smallest) cost is selected i.e. cost=5. We randomly choose the first child, although the other 4 could work.
- One epoch has finished; stopping criterion checked, but no change in medoids/cost doesn't decrease - not met.

Summary

- Less sensitive to outliers than k-means.
- Computationally expensive and not suitable for large databases.
  - Time complexity: $O(n(n-k))$
  - Space complexity: $O(n^2)$ - needs the proximity matrix.
- Does not depend on the order of examples.
- Unlike k-means, k-medoids can be used when only distances are given and not raw data.

Nearest Neighbour Clustering Algorithm

- Partitional algorithm suitable for dynamic data (data that changes over time) e.g. clustering web logs to find patterns of usage.
- Idea: Items are iteratively merged into clusters.
  - The first items forms a cluster of itself.
  - A new items is either merged with an existing cluster or forms a new cluster of itself depending on how close it is to the existing clusters.
  - Typically used distance measure is: Single Link (MIN).
- Space and time complexity: $O(n^2)$ where n is the number of items.
  - Each item is compared to each item already in the cluster.

```
Input:
   D = {t1, t2, ..., tn}    //Set of elements
   A      //Adjacency matrix showing distance between elements
Output:
   K      //Set of clusters
Nearest neighbor algorithm:
   K1 = {t1};  // t1 is placed in a cluster by itself
   K = {K1};
   k = 1;
   for i = 2 to n do      // t2-tn items: – add to an existing cluster or create a new cluster?
      find the tm in some cluster Km in K such that dis(ti, tm) is
         the smallest;
      if dis(ti, tm), ≤ t then
         Km = Km ∪ ti
      else
         k = k + 1;
         Kk = {ti};
```

- Given 5 items with the distance between them, cluster with a threshold of $t = 2$ i.e. max distance before starting a new cluster.

- **t=2**

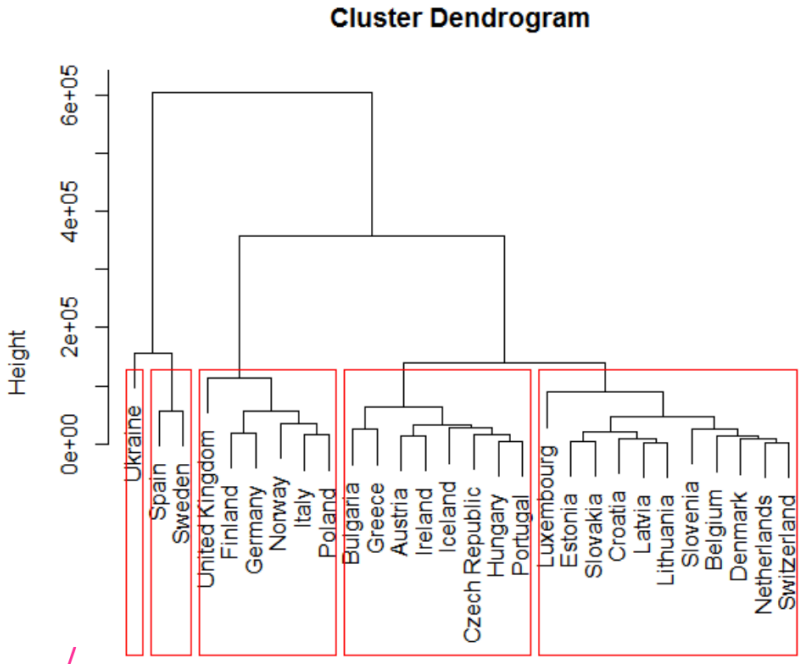
Item	A	B	C	D	E
A	0	1	2	2	3
B	1	0	2	4	3
C	2	2	0	1	5
D	2	4	1	0	3
E	3	3	5	3	0

-A: K1={A}
-B: d(B,A)=1 ≤ t => K1={A,B}
-C: d(C,A}=d(C,B)=2≤t=>K1={A,B,C}
-D: d(D,A)=2, d(D,B)=4, d(D,C)=1 =dmin ≤t => K1={A,B,C,D}
-E: d(E,A)=3, d(E,B)=3, d(E, C)=5, d(E, D)=3=dmin>t=>K2={E}

- This algorithm is sensitive to the order of examples, but it only requires a single pass through the data before settling on a clustering.

Hierarchical Clustering Algorithm

- Creates a hierarchical tree representation, called a dendrogram.
  - At the leaf level, each example is in a cluster of itself.
  - At the root level, all examples are in the same cluster.
  - The clusters at each level are produced by merging clusters from the previous level.
- Produces not a single set of clusters, but several sets - each hierarchy level is associated with a set of clusters.
  - In red: 5 clusters.
- Does not require the number of clusters k to be specified.



- Dendrograms provide a highly interpretable description of clustering → one of the main reasons for its popularity.

- Two types: agglomerative and divisive clustering.

Applicability and Complexity

- Hierarchical clustering is suitable for tasks with natural nesting relationships between clusters (taxonomies, hierarchies).
  - Biology tasks e.g. plant and animal taxonomies.
- Computationally expensive which limits applicability to high dimensional data.
  - **Space complexity**: $O(n^2)$, n - number of examples. The space required to store the distance matrix and the dendrogram.
  - **Time Complexity**: $O(n^3)$, n - levels; at each of them, n^2 distance matrices must be searched and updated.
    - Can be reduced to $O(n^2 \log n)$ if the distances are stored in a sorted list.
- Not incremental - assumes all data is present.

Agglomerative Clustering

- Bottom up - merges clusters iteratively.
  - Start with each item in its own cluster; iteratively merge clusters until all items belong to one cluster.
  - Merging is based on how close the clusters are to each other.
    - Distance threshold d: if the distance between two clusters is smaller or equal to d, merge them. More than 2 clusters may be merged.
  - Initially, d is set to a small value that is incremented at each level.

- **Example:**
  - **Given: 5 items with the distance between them**
  - **Task: Cluster them using agglomerative single link clustering**

	A	B	C	D	E
A	0	1	2	2	3
B		0	2	4	3
C			0	1	5
D				0	3
E					0

1) d=0; // initial distance threshold for merging
2) Compute the distance matrix
3) Let each data point be a cluster
4) Repeat
Increment the distance threshold with 1
Merge all clusters with distance <=d
Update the distance matrix
Until only a single cluster remains

Distance Level 0: each item is in a cluster of itself

Level 0

	A	B	C	D	E
A	0	1	2	2	3
B		0	2	4	3
C			0	1	5
D				0	3
E					0

Distance Level 1: merge A&B, C&D; update the distance matrix:

Level 1

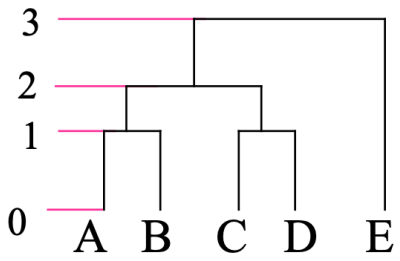
	AB	CD	E
AB	0	2	3
CD		0	3
E			0

Distance Level 2: merge AB&CD; update the distance matrix:

Level 2

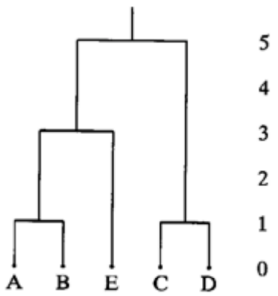
	ABCD	E
ABCD	0	3
E		0

- **Distance Level 3: merge ABCD&E; all items are in one cluster; stop**
- **Dendrogram:**



Single Link vs Complete Link Algorithm

- Single link (MIN) suffers from the 'chain effect'.
  - 2 clusters are merged if only 2 of their elements are close to each other. There may be elements in the 2 clusters that are far from each other but this has no effect on the algorithm.
  - Thus, the clusters may contain points that are not related to each other but simply happen to be near points that are close to each other.
  - More sensitive to noise and outliers.
- Complete link (MAX) - the distance between 2 clusters is the LARGEST distance between an element in one cluster and an element in another.
  - Generates more compact clusters that are less sensitive to noise and outliers.



Divisive Clustering

- Top down - splits a cluster iteratively.
- Place all times in one cluster; iteratively split clusters into two until all their items are in their own cluster.

    - Splitting based on the distance between clusters - split if the distance is greater or equal to a threshold d.
    - d is initially set to a big value that is decremented at each level.
- Less popular than agglomerative.