Agents

Anything that perceive its environment through sensors and act upon it through actuators. E.g Eyes for sensors, Hands for actuators.

- Maps percept histories to actions
- Agent = Architecture + Program
- Autonomous - behavior determined by own experience

Performance Measure

How agent know its doing the right thing?

- Best for whom?
- What are we optimizing?
- What info is available?
- Unintended effects?
- Costs?

Rational Agent

Choose an action that's expect to maximize performance measure, given evidence provided by :

- percept sequence
- built-in knowledge

Defining the Problem

- Performance Measure - how well something was done
- Environment - place that we are in
- Actuators - controls the system
- Sensors - measures things in the surrounding

Characterising the Environment

- Fully observable - sensors give complete state of environment (vs partially observable)
- Deterministic - next state determined by current state and action executed by agent (vs stochastic). If environment is deterministic except for actions of other agents, then its strategic.
- Episodic - experience divided into different time periods, choice independent of episodes (vs sequential)
- Static - environment unchanged while agent is deliberating (vs dynamic). If environment is unchanged but agent performance score change with time then its semi-dynamic.
- Discrete - limited number of distinct, clearly defined percepts and actions (vs continuous)
- Single agent - an agent operating by itself in an environment (vs multi-agent)

Agent Types

- Simple Reflex - based on some set of specified rules
- Model-based - keep state and decide what to do next based on state
- Goal-based - figure out goals instead of using if-else
- Utility-base - given current state, how happy am I?
- Learning agent - learn about the world and generate new problems to be solved. Might not maximise performance.

Exploitation vs Exploration

Tradeoff between current knowledge (best of current) vs unknown (possibility of better)

Problem Types

- Single State - agent know which state it would be in, solution is a sequence
- Sensorless - agent don't know where it is, solution is a sequence
- Contingency - percepts provide new information about current state, interleave search and execution
- Exploration - unknown state space

Single State

Problem formulation can follow these guides :

- Initial state
- Actions or successor functions as a set of action-state pairs,
  $< action, state\ as\ a\ result\ of\ action >$
- Goal test - check if have reached goal. Explicit or implicit checking
- Path cost - cost of getting to solution

State space abstracted for problem solving. Each abstract action should be easier than the original. Solve abstract and map back to world.

General Tree Search

Similar to BFS, but check if each item in fringe passes the goal test. **State** is a representation of physical configuration (like the actual board). **Node** is a data structure that includes **State**, **Parent node**, **Action**, **Path cost**, **Depth**.

Uninformed Search Strategies

Only can differentiate goal states and not the transition states.

- Completeness - does it always find a solution
- Time and Space complexity - measured using branching factor(b), depth of least cost (d) and max depth of tree (m), may be ∞
- Optimality - does it always find least-cost

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Breadth First Search

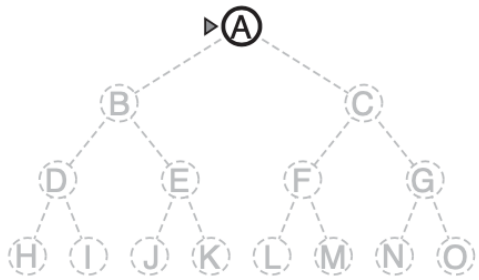
Implemented using a queue (new successors go to the back). If b is finite, its complete. **Optimal if path cost is a increasing function of depth - no negative weights**. Main issue is space - exponential increase in space is undesirable. $O(b^d)$ for time if we check when generated instead of when selected.

Uniform Cost Search

Expand **least-cost** unexpanded node. Fringe is a priority queue ordered by cost. Since step costs are positive and cost $\geq \varepsilon$, then the number of nodes needed to reach the goal would be $\frac{C^*}{\varepsilon}$. If the costs for each step is the same, then $\frac{C^*}{\varepsilon} = d$. Performs slightly worse then BFS.

Depth First Search

Expand **deepest** unexpanded node. Fringe is a stack or a LIFO queue. Not optimal because of infinite depth possibility.



In the figure above, if J and C are possible solutions, DFS returns J first which is not an optimal solution. Hence its not optimal. DFS may generate all the nodes at m. If $m \gg d$, then run time is bad. If solutions are dense, may be better than BFS.If $m \approx d$, may be better if we some how hit a solution on the first dive.

Depth-Limited Search

DFS but we limit the depth, l. Nodes at depth have no successors. Can see this as DFS with $m = l$. Choose a depth before starting the algorithm, creating possible incompleteness or sub-optimal solutions.

- Incomplete - $l < d$, haven't reach d, answer not found
- Not optimal - $l > d$, similar to normal DFS reasoning.

Fails with either limit, or not found.

Iterative Deepening Depth First Search

Manually limit the depth that DFS can go. Generates the nodes at d once, $d-1$ twice and the root eventually gets generated d times, resulting in b^d run time. Space is the same as DFS

Repeated States

Repeated states can be prevented using **memoisation**.

Bidirectional Search

Search that start from the top and bottom. Exploit the fact that $O(b^{\frac{d}{2}}) \times 2 < O(b^d)$, same for space. Stop when the frontiers intersect if we use iterative deepening. To determine whether bidirectinal search is better, need to look at the branching factors when going backwards and forwards. If the overall time and space is worse than just going forward, bidirectional search may be worse.

- Require method for computing predecessors - going upwards
- Multiple goal states - construct a goal state containing all goals
- Check if node appears in other tree - hashtable.

Informed Search

Access to heuristic function, and performance is affected by quality of heuristics.

Terminology

- Evaluative function, $f(n)$ - cost estimate from current to goal
- Heuristic function, $h(n)$ - estimated **cheapest** cost from current to goal
- Admissable - **never over-estimates** cost to reach goal, **optimistic**
- Consistent - **triangle inequality** is satisfied, $f(n)$ is non-decreasing along any path.

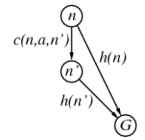
$h(n) \leq c(n,a,n') + h(n')$

If h is consistent, we have

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

$= g(n) + c(n,a,n') + h(n')$

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



Greedy Best-First Search

Use an evaluative function, $f(n)$ for each node to estimate desirablity. Order in **priority queue** based on function. Expand the nodes that are closest to the goal. If $f(n) = -depth(n)$, then DFS is a special case of Best-First Search.

- Incomplete - can get stuck in loops
- Time - b^m, may need to go through entire tree
- Space - b^m, every node is store in the PQ in the worse case
- Optimal - no, expanding closest to goal may not always be the right way to do so, may lead to a longer path

A* search

Uses the function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost so far to reach n, $h(n)$ is the estimated cost from n to goal, $f(n)$ is the estimated **total** cost from n to goal. Goal checking done only when **popped** from frontier. Consistency is a stricter requirement that admissibility. Uniform Cost Search is A* using $h(n) = 0$.

- If $h(n)$ is admissable, then $f(n)$ never over-estimates.
- If $h(n)$ is admissable, then A* using **Tree-Search** is optimal.
- If $h(n)$ is consistent, then A* using **Graph-Search** is optimal.

Since all the nodes are stored in memory using priority queue, space of A* is the number of node. Run time of A* is exponential however, exponential in the number of nodes. Every node with $f(n) < C*$, where $C*$ is the cost from start to goal, will be expanded.

Heuristics

- Manhatten Distance - distance from current to any square without moving along diagonals.

Optimizing A*

Prune paths that are less likely to be useful **Relaxed Problems** relaxes certain restrictions of the current problem for it to be solved more easily. Then map it back to the original - cost of optimal solution in relax is an admissable heuristic for the original.

Dominance

If 2 heuristics are admissble, then the one that is dominating is the one that is better used for search, $h_2(n) \geq h_1(n)$, then $h_2(n)$ is better for search

IDA*

Cut off uses $g + h$ instead of depth. Cut off value is the smallest f of any node that exceeded the cutoff of the previous iteration.

Recursive Best-first Search

Keep track of second best f-value seen so far. Parent node keeps track of best f-value for children. Backtrack when we end up at a node with children worse than the second best f-value. Basically ensure that we check out the possibly optimal paths to find the goal. Optimal if $h(n)$ is admissable. Time complexity depends on: how the path changes based on the nodes. Remembers too little, linear space uses too little memory.

Simplified MA*

Tries to solve memory problem for A*. As long as theres memory, we move. Once memory full, drop the worst f-value, and update the parent on dropped sub-tree

Local Search

Keep track of current state, only update the current with better state until the current becomes the goal.

Hill Climbing Search

When the path to the goal does not matter. Complete algorithm finds a goal if one exists. Optimal one finds the global maxima/minima. Continuously finds a better value, thereby climbing up the hill. But the peak of that hill may not be gloabl maxima. And because we are at a local maxima, we may get stuck there because theres no greater neighbors. Can get lost at shoulders (flat areas)

Simulated Annealing Search

Escape local maxima by allowing some bad moves. If frequency of bad moves decrease slowly enough, then we will find a global optimum with probability approaching 1 (understand this!)

- Pick a random move
- If situation improve, make the move
- Otherwise, accept the move with some probability.
- Probability decreases with badness of move. If the move is damn bad, the probability would be lower.

Beam Search

Perform multiple hill-climbing searches in parallel. Basically instead of 1 best state, keep multiple best states.

- Local Beam Search : k threads share information
- Stochastic Beam Search : k independent threads

Genetic Algorithm

Successor generated by combining 2 parent states. Start with a population of states. And then rated with an evaluative (fitness) function, higher values for better states.

Games

Specify a move for every possible opponent reply. Time limits means that it is unlikely to find a goal, have to approximate.

Minimax

Alternating turns between **min** and **max**. Min aims to minimise values while max does the converse. Each node has a **minimax** value and that is the utility of being in the corresponding state, **assuming both players play optimally**. Playing optimally means maximising or minimising the minimax values on their turn.

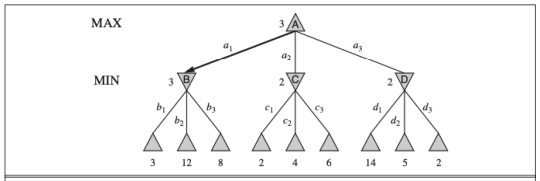


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Algorithm

Go all the way down the tree and then the minimax values are backed up through the tree as recursion unwinds.

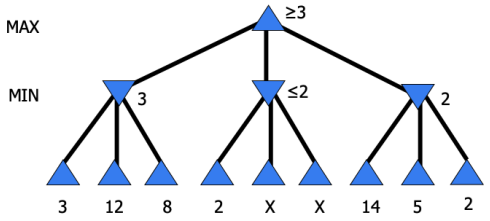
Properties

- Complete - if tree is finite, because we go down to the max depth of the tree before coming up. If infinite, will never find answer
- Time - $O(b^m)$, goes down all the way to max depth
- Space - $O(bm)$, stores only the current nodes along the path
- Optimal - if playing against optimal opponent, yes.

α-β Pruning

Ignore paths that will definitely not be chosen. Node ordering matters because of DFS ordering. Initially $\alpha = -\infty$ and $\beta = \infty$ Pruning branches as soon as the value of the current node is known to be worse than current α(for MAX nodes) or β(for MIN nodes).

- Going from the leaves up, we get a range of values for each node.
- If the node will never be chosen because there is a better alternative, prune it away.
- If however there is a possibility of a better node, we need to expand that node to determine if its truly better.



Going from left to right, we can prune the middle branch because at that point, 3 is our best value and MAX will never choose that branch. The last branch had to expanded all the way because of the ordering of the nodes(14,5,2). If it were the other way, can prune this branch also.

Properties

- Pruning doesn't affect final result
- Good move ordering improves performance
- With perfect ordering, improves to $O(b^{\frac{m}{2}})$, in the worse and average case - $O(b^{\frac{3m}{4}})$

Resource Limits

- Cutoff test - decides when to apply evaluation function
- Evaluation function - estimated desirability of position, turning nonterminal nodes into terminal leaves.

Evaluation Function

Returns an estimate of the expected utility(in terms of expected value) of the game from a given position.

- Order the terminal states in the same way as the true utility function - states that are wins must evaluate better than draws, etc.
- Cannot take too long
- For nonterminal states, evaluation function should be strongly correlated with actual chance of winning.

Calculates various features of a state, and return the expected value of the function. Features should be independent

Cutting Off Search

Modified Minimax by replacing

- Terminal nodes with the cutoff nodes
- Replacing utility function with evaluation function

Intro to ML

- Data collection is expensive - dirty data need to be cleaned.
- If machine has more exp, less code to write

Feedbacks

- Supervised - correct answer given for each answer
- Unsupervised - no answers given, deduce answers from data
- Weakly supervised - correct answer given but not precise
- Reinforcement - occasional rewards are given for right answers

Supervised Learning

Given a data set, know what we are looking for and have some idea of a relationship between input and output

- Regrssion - mapping input to output
- Classification - Discrete output, categorising

Unsupervised Learning

- No feedback on the predicted results
- Little to no idea on what results should look like
- Want to derive structure from data but don't necessarily know the effect of variables
- Try to derive structure by clustering data based on relationship among variables in data

Inductive Learning

Find a hypothesis such that agrees with f - consistent(matches output), curve fitting. Basically finding a best fit line for the data points.

Linear Regression

Given a set of data, how do we find a hypothesis $h_\theta(x) : \theta_0 + \theta_1 x$ that fits the data well? Choose values of θ_0 and θ_1 so that hypothesis is close to training examples.

Squared Error Function

Cost function to measure how close hypothesis is to training data. Squared because its optimized for math.

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2$$

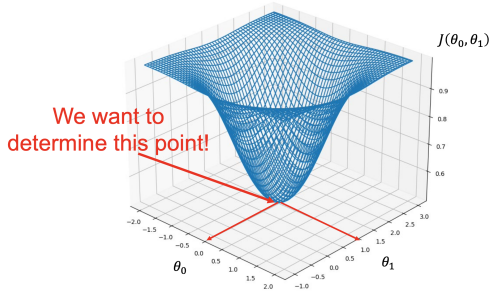
$h_\theta(x^i)$ refers to the value of the hypothesis at the ith data point. Aim to minimise cost function by picking the smallest value of θ_0 and θ_1

Simplification

Simplfying the hypothesis to 1 variable allows for easier visualisation of cost that is just based on gradient.

Gradient Descent

By plotting cost against the respective variables, we get a bowl like 3D structure. To minimise the cost function, we want all minimum of the cost function.



To find this minimum point, use hill climbing search. Find values that reduces the effective cost. But this may end with local minimum instead.

$$\theta_j := \theta_j - \alpha\frac{\partial J(\theta_0,\theta_1,...)}{\partial\theta_j}$$

α is the learning rate and values of θ have to be updated only after all calculations are done. Otherwise, wrong values may be used.

Proof of Correctness

Consider a parabola in a U shape on the graph $J(\theta_0)$ (y-axis) against θ_0 (x-axis). If we are on the side that is declining, then $\frac{\partial J(\theta_0,\theta_1,...)}{\partial\theta_j} > 0$ because gradient is positive. Since α is positive, the value of θ decreases and approaches the minimum. On the otherhand, if gradient is negative, $\frac{\partial J(\theta_0,\theta_1,...)}{\partial\theta_j} < 0$ then θ increases and approaches the minimum as well.

Size of alpha

- Too small - too many steps
- Too big - overshoot and will begin to oscillate

Steps naturally get smaller if α is constant

Gradient Descent for Linear Regression

$$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)} - y^{(i)})^2$$

$$\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_j} = \frac{\partial}{\partial \theta_j}\frac{1}{2m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)} - y^{(i)})^2$$

$$\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)} - y^{(i)})$$

$$\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)} - y^{(i)}).x^{(i)}$$

h'

Differentate with respect to each parameter of the cost function. Result of this is a convex function - bowl shaped. Then go to the bottom of the bowl while the line changes.

Gradient Descent Variants

- Batch - consider all training points
- Stoichastic - consider one at a time, faster and more randomness
- Mini-Batch - reduce the side of training points given and work on that

Multi-Feature Gradient Descent

Multiple variables can exist in the hypothesis. Cost function changes to include more parameters but summation equation is unchanged. θ is a column matrix by default so we have to transpose it to be able to perform vector multiplication with x

Hypothesis:

$$h_\theta(x): \theta^T x$$

vector

Cost Function:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Gradient Descent:

$$\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial \theta_j}$$ simultaneously for all θ_j

Feature Scaling

Gradient Descent don't work well if features have significantly different scales. Solution: scale features so that they vary on roughly the same scale. **Aim for** $-1 \le x_i \le 1$ - forced around origin. In the case of exponentially size features, feature scaling must be done as well.

Mean Normalisation

Removing the mean from features so that they all have about 0 mean. σ refers to standard deviation.

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$

Normal Equation

Suppose m examples with n features; $(x^(1), y^(1))$ ... $(x^(m), y^(m))$. Then

$$x_i = \begin{bmatrix} x_0^i \\ \vdots \\ x_n^i \end{bmatrix}$$

$$X = \begin{pmatrix} (x^1)^T \\ \vdots \\ (x^m)^T \end{pmatrix}$$

We can also include a column of ones (bias column) in front of X, making X the **Design Matrix**. The normal equation is defined as

$$\theta = (X^T X)^{-1} X^T Y$$

Where Y is the vertical matrix of $y^(1)...y^(m)$. $X^T X$ **needs to be invertible**.

Comparing GD and NE

GD	NE
• Need to choose α	• $X^T X$ needs to be invertible - $O(n^3)$
• Many iterations	• Slow if there are too many features
• Works well even when there's many features	

Classification

Predict results in discrete output, map input variables into discrete categories

Decision Trees

Something like a flow chart and the leaves are the ultimate decision.

Expressiveness

Express any function of the input attributes. Tree will likely be the same for multiple runs unless the function is nondeterministic. Probably won't generalise to new examples. **Goal : find more compact trees**.

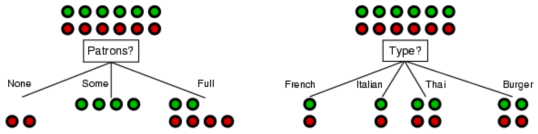
Hypothesis Space

Trees are very expressive. Refers to all possible trees that can lead to a plausible decision. More expressive spaces :

- increase chance that target function can be expressed
- increase number of hypothesis consistent with training set
- may get worse predictions

Choosing Attributes

A good attribute to choose splits examples into subsets that are either all good or all bad. Why? - easier to make decision.



The left attribute is better than the right one. Since there are distinct all positives and all negative sets, it would be easier to have a path that goes through those sets. If the sets are like those in the right, it would be difficult to proceed since we may have to go through each of the child.

Information Theory

Getting information based on entropy(unorderliness in a set, higher entropy means more disordered)

$$I(P(v_1)...P(v_n)) = -\sum_{i=1}^{n} P(v_i)\log_2 P(v_i)$$

For a training set with p positive examples and n negative examples :

$$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

$\mathbf{I(\frac{1}{2}, \frac{1}{2}) = 1}$

Entropy measures the number of bits to represent data. Higher entropy means that more bits can be used to represent data.

Info Gain

Entropy of current node - entropy of children. Entropy of children given by :

$$remainder(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

Then the information gain would be :

$$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$$

Then choose the attribute A with the largest information gain. Information gain is how much entropy we removed. **What if information gain is the same for 2 attributes? How to choose and does order affect?** - think it shouldnt affect since we are looking at the entropy difference between parent and children. If 2 children have the same entropy, picking either will result in the same result.

Do on calculator - use natural log, then divide by log(2) and then multiply by the fraction

Performance Measurement

Decision tree is the hypothesis approximation. On training set, deeper tree means more accurate. On test set, deeper tree decreases accuracy. **Why?** Hypothesis used for test data could only satisfy the data of the test. May not be generalisable. Leads to **overfitting** - where error of test data matches training data until a certain point.

Occam Razor

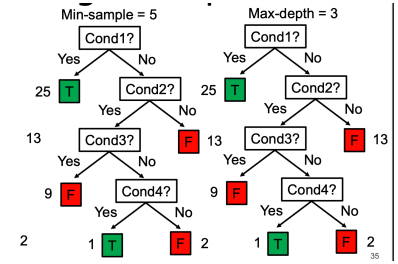
Prefers short hypothesis. Short hypothesis that fits data is unlikely to be coincidence, but long hypothesis that fits may be(Lucky that long hypothesis fits both test and training).

Overfitting

Tree too finely tuned to noise in training data, cannot be generalised.

Pruning

Prevent node from splitting even when it fails to cleanly separate examples. Check if attribute is relevant by statistical technique. Results in smaller tree with better accuracy. Look at nodes with only leaf nodes as descendents and determine if we can elminate that node and replace it with a leaf node instead.



For min-sample, split the node only if the set has more than the value of min-sample. Once we reach a node with only leaves, see if can combine by rounding it to the more dominent node. Cond4 can be rounded to 2 False nodes and replaced with 2 False nodes. Then when leaves are the same nodes, they can be combined with the parents. The resulting tree would just have cond1 with 25 yes and 25 no. Similarly for Max-depth. Once the depth exceeds, combine the nodes starting from the bottom. And similar to the min-sample, we can combine the node if their leaves are the same too.

Continuous-Valued Attributes

For values that have ranges, we can partition them into smaller sets with fixed ranges instead. Gives them some form of ordering and its branching factor dependent.

Attributes with Many Values

Problem - gain will select attribute with many values like dates. **Solution** - use gain ratio instead, bias against attributes with alot of values.

$$GainRatio(C,A) = \frac{Gain(C,A)}{SplitInformation(C,A)}$$

$$SplitInformation(C,A) = -\sum_{i=1}^{d}\frac{|E_i|}{|E|}\log_2\frac{|E_i|}{|E|}$$

Attribute A divides training set E into subsets $E1...Ed$ corresponding to d distinct values of A

Attributees with differing costs

Problem - How to learn a consistent DT with low expected cost? **Solution** - Replace Gain by :

$$\frac{Gain^2(C,A)}{Cost(A)}$$

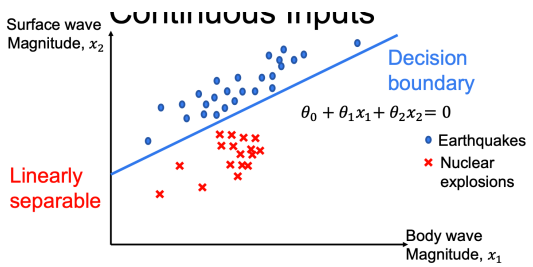
$$\frac{2^{Gain(C,A)}-1}{(Cost(A)+1)^w}$$

Solution will be biased against high cost.

Missing Attribute Values

Problem - what if some examples are missing values of attribute? **Solution** - Randomly assign based on current attribute values. Could use most common value is one way.

Continous Inputs



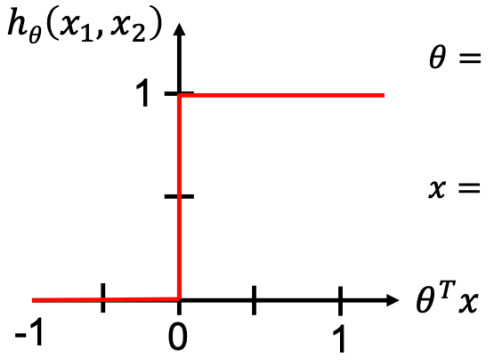
Decision boundary separates 2 classes of points. Linear decision boundary is a **linear separator**, meaning that the data is **linearly separable**.

Logistic Regression

Based on decision boundary, want to classify classes that are to the left and right of linear separator with values 0 and 1. The linear separator has a value of 0. So values above that have values greater than 0 and values below it have values smaller than 0. If we want to classify the values above the line to have values of 1 and below as 0, then

$$h_\theta(x_1, x_2) = \begin{cases} 1, & \text{if } \theta_0 + \theta_1 x_1 + \theta_2 x_2 > 0 \\ 0, & \text{otherwise} \end{cases}$$

Results in a graph that is discontinous and not differentiable which is not ideal - very confident prediction of 1 or 0 even for examples that are close to the boundary.



Logistic Function

'Softer' version of the threshold function that is differentiable and continous. Now, the hypothesis is

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

Even though the hypothesis is now differentiable, another problem arises in that the cost function is no longer convex (not a nice U shape) - **Why is this an issue?** Gradient descent can get stuck at local minima. Update the cost function to the following :

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x), y^(i))$$

Cost Function

$$Cost(h_\theta(x), y) = \begin{cases} -log h_\theta(x), & \text{if } y = 1 \\ -log(1 - h_\theta(x)), & \text{if } y = 0 \end{cases}$$

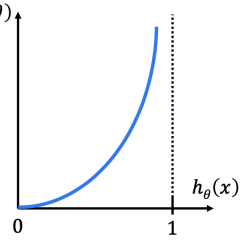
$log(1) = 0$, $log(0) = \infty$

If lets say y = 1, then the cost should approach 0 as h_θ approaches 1(which is the right answer). And vice versa for h_θ approaching 0 (further away from right answer, cost increase exponentially).

Consider $y = 0$:

$h_\theta(x) \to 0, \; J(\theta) \to 0$

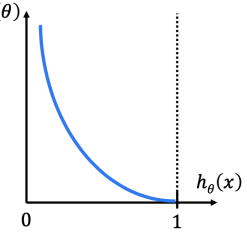
$h_\theta(x) \to 1, \; J(\theta) \to \infty$



Consider $y = 1$:

$h_\theta(x) \to 0, \; J(\theta) \to \infty$

$h_\theta(x) \to 1, \; J(\theta) \to 0$



can be combined to the following :

$$Cost(h_\theta(x), y) = -y log h_\theta(x) - (1 - y) log(1 - h_\theta(x))$$

Results in the same as gradient descent for linear regression!

Multi-class Classification

Perform binary classification with each class and take every other class as the other set. Then pick the class that maximises the heuristic function.

Overfitting

More features means better fit, but too many cooks can spoil the broth. Too few features - underfitting. Too many features - overfitting. Overfitting is not ideal because it can end up making poor predictions. Resolved by :

- Reduce number of features
- Regularise - keep all features but reduces magnitude of parameters θ_j

LR with Regularisation

Hypothesis:

$$h_\theta(x): \theta^T x$$

Cost Function:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda \sum_{i=1}^{n} \theta_i^2\right]$$

Second term known as **regularisation paramter** - aims to avoid over-fitting by scaling the values of θ using λ.

$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x_n^{(i)} - \frac{\lambda}{m}\theta_n$$

$$:= \left(1 - \frac{\alpha\lambda}{m}\right)\theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x_n^{(i)}$$

slightly less than 1 — "regular" gradient descent

Regularised Normal Equation

Mapping from linear regression, create identity matrix but the 0-th row and 0-th column value is 0. Because this is θ_0 which is the bias.

$$\theta = (X^T X + \lambda \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & 1 \end{bmatrix})^{-1} X^T Y$$

This works even if $X^T X$ is non-invertible if λ>0!

Regularised Logistic Regression

Hypothesis:

$$h_\theta(x): \frac{1}{1 + e^{-\theta^T x}}$$

Cost Function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right) + \frac{\lambda}{2m}\sum_{i=1}^{n} \theta_i^2$$

Gradient Descent:

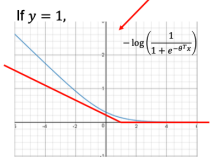
$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x_n^{(i)} - \frac{\lambda}{m}\theta_n$$

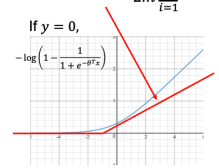
SVM

SVM splits 2 sets of data into 2 groups, finding the best way to split them in the process with the biggest margin.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right) + \frac{\lambda}{2m}\sum_{i=1}^{n} \theta_i^2$$

$$= \frac{1}{m}\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x) - (1 - y^{(i)}) cost_0(\theta^T x) + \frac{\lambda}{2m}\sum_{i=1}^{n} \theta_i^2$$





Approximate the curve to a straight line. $cost_1$ refers to when the cost is 1.

$$\min_\theta \left(\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x) + (1 - y^{(i)}) cost_0(\theta^T x)\right] + \frac{\lambda}{2m}\sum_{i=1}^{n} \theta_i^2\right)$$

convention:

$$\min_\theta C\left[\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x) + (1 - y^{(i)}) cost_0(\theta^T x)\right] + \frac{1}{2}\sum_{i=1}^{n} \theta_i^2$$

m is a constant, so can cancel out. Rename lambda as C. Difference between logistic regression cost function is the difference in $cost_i$. Solved using libraries. Can be seen as $CA + B$, where A is the summation of the costs, and B is the regularisation parameter. $C \approx \frac{1}{\lambda}$. If λ is large, then SVM ignores outliers. Because gradient is of line is about 0 so its just a straight line cutting y-axis. If λ is small, then its a vertical line cutting x-axis, prolly won't split nicely, take into account outliers.

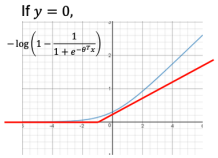
Math Intuition

$$\min_\theta C\left[\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x) + (1 - y^{(i)}) cost_0(\theta^T x)\right] + \frac{1}{2}\sum_{i=1}^{n} \theta_i^2$$

is equivalent to:

$$\min_\theta \frac{1}{2}\sum_{i=1}^{n} \theta_i^2$$

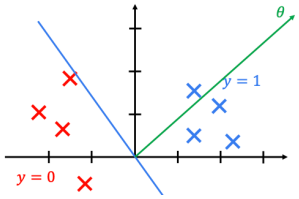
C very large —
s.t. $\theta^T x^{(i)} \geq 1$ if $y = 1$
s.t. $\theta^T x^{(i)} \leq -1$ if $y = 0$



Lookig at the graph for $y = 1$, cost is the lowest when x-axis values are ≥ 1. Likewise for $y = 0$, cost is the lowest when ≤ -1. Provided C is very large - cos we care about outliers.

$$\min_\theta \frac{1}{2}\sum_{i=1}^{n} \theta_i^2 = \theta_1^2 + \theta_2^2 = \left(\sqrt{\theta_1^2 + \theta_2^2}\right)^2 = |\theta|^2 \qquad \theta^T x^{(i)} = p|\theta|$$

s.t. $p|\theta| \geq 1$ if $y = 1$
s.t. $p|\theta| \leq -1$ if $y = 0$
Consider $n = 2, \theta_0 = 0$



Mathematically, its the product of the length of projection and the length of the vector its projected on.

Kernel Tricks

Non-linearly separable - solution using polynomial regression. Kernel is basically increasing the number of dimensions to find a way to separate the points.

Gaussian Kernel

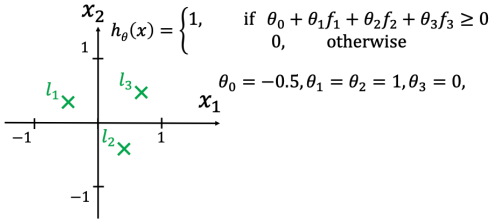
$$\min_\theta C\left[\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T x) + (1-y^{(i)}) cost_0(\theta^T x)\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

$$h_\theta(x) = \begin{cases} 1, & \text{if } \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$h_\theta(x) = \begin{cases} 1, & \text{if } \theta_0 + \theta_1 f_1 + \cdots + \theta_n f_n \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$f_i = similarity(x, l_i) = e^{-\frac{|x-li|^2}{2\sigma^2}}$$

Based on the distance of the point to a landmark, decide whether to include it in the boundary.



$$h_\theta(x) = \begin{cases} 1, & \text{if } \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\theta_0 = -0.5, \theta_1 = \theta_2 = 1, \theta_3 = 0,$$

If $h_\theta(x)$ is 1, then its in the boundary, otherwise its not. Compare distance from the landmarks using f and then determine the final result, whether its in or out.

SVM with Kernels

From training data, choose landmark for each data point. Generate hypothesis.

From training data:
$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})$

We can generate:
$(f^{(1)}, y^{(1)}), (f^{(2)}, y^{(2)}), \cdots, (f^{(m)}, y^{(m)})$ $f \in \mathcal{R}^{m+1}$

Hypothesis: $h_\theta(x) = \begin{cases} 1, & \text{if } \theta^T f \geq 0 \\ 0, & \text{otherwise} \end{cases}$

$$\min_\theta C\left[\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f) + (1-y^{(i)}) cost_0(\theta^T f)\right] + \frac{1}{2}\sum_{i=1}^{m}\theta_i^2$$

Additional +1 for bias column.

- C : if large, more variance, less bias. else more bias, less variance.
- σ^2 - if large, features vary more smoothly (more spread out), higher bias, less variance. else lower bias, higher variance.

Training Logistic Regression

- minimise the cost function
- compute test set error
- misclassification error - some form of bias towards test set.

How to choose a model? - train every model, and pick the model with lowest test set error.

Measuring goodness of Model

Split test set into validation and test. Now, 3 ways to measure goodness :

1. Training Error

$$J_{train}(\theta) = \frac{1}{2m_{train}} \sum_{i=1}^{m_{train}} (h_\theta(x_{train}^{(i)}) - y_{train}^{(i)})^2$$

2. Cross Validation Error

$$J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

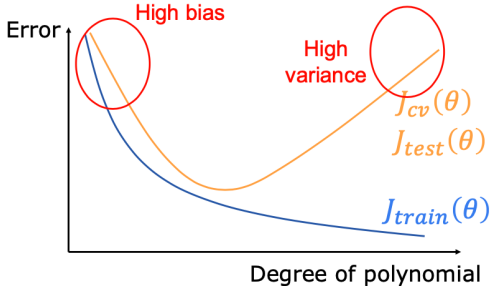
3. Test Error

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

Compute $J_{cv}(\theta)$, and pick the lowest. Then use test error to estimate performance on unseen samples. Validation set makes resulting hypothesis bias towards it.

Bias and Variance

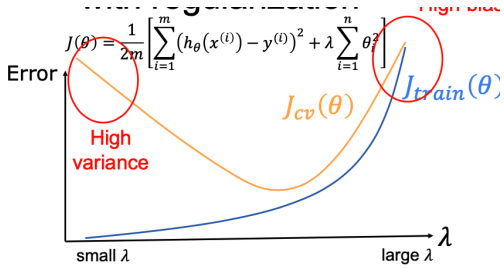
Underfitting - high bias. Overfitting - high variance.



As the degree of polynomial increase, error of training set decreases because we can better fit the data. With low degree, fit is bad, more error. With high degree, fit is good. But for a sample that is not seen before, the error will be high.

Choosing λ

With small lambda, the magnitude of parameters are still relatively big, hence there is still overfitting. But with a big lambda, it will cause underfitting since magnitude of parameters are so small.



For training :
When lambda is small, error will be small, because essentially theres like no term. But when lambda is large, regularisation parameter is more significant and hence will have higher weightage in the fitting, leading to a poorer fit of the actual data and hence greater error. For cv : when lambda is small, underfit, more error. when lambda is big, overfit, more error. When lambda is small, become overfitting, when lambda is big, become underfit. Notice the difference in the tags for this graph and that with the error-polynomial graph.

Summary

On underfit (bias)

- J_{train} high
- $J_{train} \approx J_{cv}(\theta)$

On overfit (variance)

- J_{train} low
- $J_{train} \ll J_{cv}(\theta)$

Feature Scaling

In similarity function, chance that certain values of x will dominate others, need to perform feature scaling so that the similarity function would be fair.

Choosing the methods

n is number of features including bias column. m is number of training samples. Intermediate value is about 10,000. **SVM can be faster than neural networks sometimes.**

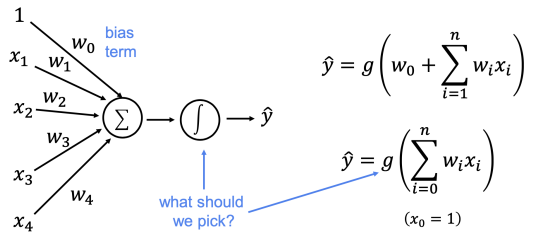
$n \gg m$	n small, m ok	$m \gg m$
logistic regression, SVM no kernel (don't transform)	SVM with gaussian	logistic regression, SVM no kernel (don't transform)

Neural Networks

Aim to linearly separate 2 groups of points. Independent on number of features, can be used on data with lots of features.

Perceptron

Based on the idea of perceptrons in the brain, inputs with their respective weights, sum them up, apply a function to get an output.



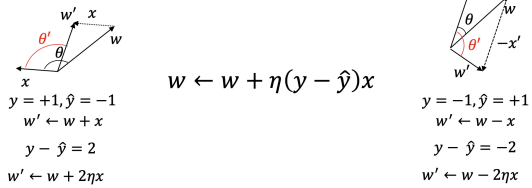
g refers to the non-linear activation function.

Perceptron Update Algo

Similar idea to gradient descent. Select initial weights, apply onto features, classify, then update. Repeat till converge. Using the sigmoid function as the evaluative function.

$$w = w + \eta(y - \hat{y})$$

Since perceptron aims to find best fit line, aims to reduce the angle size between wrongly classified points and their correct classfications. y is the correct classifier of the point and $\hat{y}$ is the curren classifier of the point, based on the function.



In the case of wrongly classifying a positive $\hat{y}$, where the angle should be greater than π, we have to shift the weight vector right. Conversely for the negative $\hat{y}$ on misclassification.

Perceptron	Linear SVM
• not robust: Can select any model to linear, not deterministic • Cannot converge on non-linearly separable data	• Perceptron of "optima stability" • Maximizes margin • Soft margin: can learn from non-linearity separable data

Gradient Descent

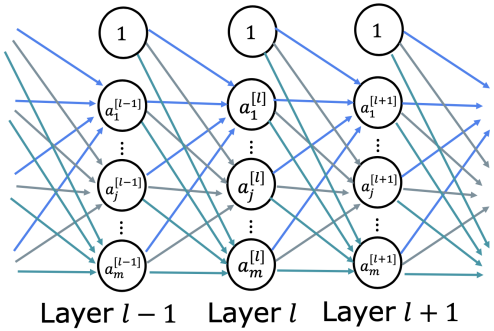
To find the appropriate weights, perform gradient descent, ie find the lowest MSE for $y - \hat{y}$. As a result of some differentiation using sigmoids and chain rule, we get

$$w = w + \eta(\hat{y} - y)\hat{y}(1 - \hat{y})x$$

where x is the feature value.

Multilayer Perceptron

Outputs can be mapped as inputs to another activation function. These intermediate functions between the input and the output are known as **hidden layers**.



Back Propagation

Used to determine the weights of the hidden layers. Have to go forward first, using the weights of the previous layer, then use back propogation to get the accurate weights of the entire network.

Matrix Form

$\hat{y}'(W^{[l]}) = a^{[l-1]} \quad \delta^{[l]}$

$\delta^{[l]} = g'^{[l]}(f^{[l]}) \quad W^{[l+1]} \quad \delta^{[l+1]}$

Basic idea : differentiate from left to right. g is the activation function used. f is the product of weights and the inputs.

Properties

Efficiently computes gradient by :

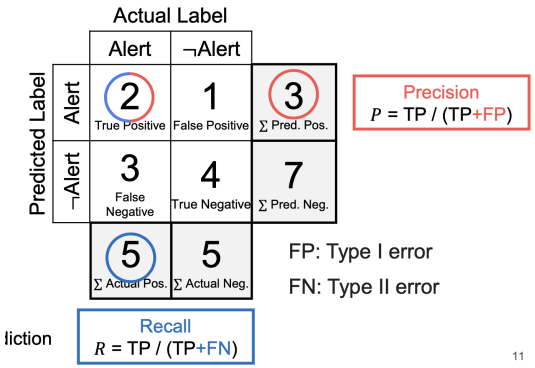
- avoiding duplicate calculations
- not computing unnecessary intermediate values
- compute gradient of each layer

Basically uses something like dynamic programming. Base case : start at last layer, compute the loss in weight. Subsequent layers uses calculated value of the next layer multiplied by the loss in weight of the current layer.

Precision vs Recall

Measure of how good a model is.

- Precision : correctly convict a guilty person (minimise if false positive is costly)
- Recall : percentage of correct convictions (minimise if false negative is costly)



F1 score

$$\left(\frac{P^{-1} + R^{-1}}{2}\right)^{-1}$$

- More robust, less sensitive to extreme values
- Consider that numerators of P and R are the same, compares denominators only.

Receiver Operator Characteristic Curve

Values below threshold lines are supposed negative. Percentage of true positive and false positive shown. Area under graph give a measure. As long as area greater than half, its better than chance. And if it approaches 1, then its very accurate. Greater than half because half is the straight line graph obtained when left to chance.

Spatial Exploitation

Convolution - consider a group of pixels instead of individual when processing images. Without convolution, huge number of inputs and weights. Convolution sends groups of pixels, reducing number of inputs. With kernel, mutliply and sum together to get result. Element-wise multiplication and then sum everything up. Different types of kernels results in different effects on the image.

Padding & Stride

Pixels are lost along edges, use padding. Instead of going sequentially, use something like python 'step' works for both horizontal and vertical, this is stride. Since padding increases the size of image matrix, the resulting feature map will also be bigger.

Linear Layer: $a^{[l]} = g^{[l]}\left(\left(W^{[l]}\right)^{\top} a^{[l-1]}\right)$

could be same

Activation is a 3D Matrix! Concatenation of Feature Maps

Convolution Layer: $A^{[l]} = g^{[l]}\left(W^{[l]} * A^{[l-1]}\right)$

Weights = **Kernels** (3D matrix)

Pooling Layer

Feature maps can get very big, downsample feature maps. Helps to train later kernels to detect higher-level features. Reduces dimensionality. Groups pixels in groups, essentially reducing the size of the image by size of group. This means more blurry and pixelated.

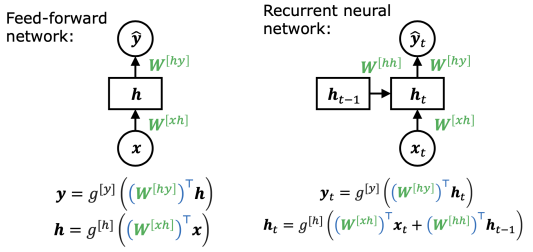
Softmax

Detects maximum not as an absolute value. Bigger values are closer to 1.

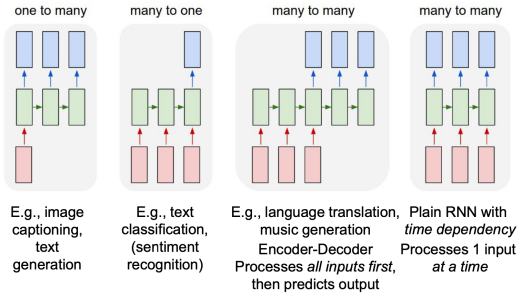
$$P\left(y = j|\theta^{(i)}\right) = \frac{e^{\theta^{(i)}}}{\sum_{j=0}^{k} e^{\theta^{(j)}}}$$

Recurrent Neural Network

Multi layer perceptron will be too slow. Too many features to process. Instead, exploit some form of temporal locality. Predice output based on input and the previous function. Similar to deciding RTT. This is known as **Exponentially-Weighted Moving Average (EWMA)**. λ between 0 and 1.



Types :



Issues

Overfitting

Common issue for all ML models. Use regularisation to resolve.

- Drop-out : some activations become 0 randomly. More training will actually improve and not plateau
- Early stopping : Since loss from test will increase after certain point, we can stop when we reach minimum loss on **test set**.

Vanishing/Exploding Gradients

Sigmoid functions are less than 1. As more values are multiplied together, approach 0 and 'vanishes'. If gradients keep getting larger, will diverge and 'explode'.

Solutions to problems :

- Proper weights initialisation - random weight initialisation
- Don't use functions that have limits - choose something like Relu
- Batch normalisation - something like feature scaling (normalise inputs of layers)
- Gradient Clipping - cap the loss so it doesn't get too big

Unsupervised Learning

- Little to no idea what results should look like
- No feedback on predicted results

K-Means

- Pick k random points for K clusters (centroids)
- Group points closest to centroids as the same cluster
- Find the mean squared distance between the point and its own centroid - loss
- Repeat till converge

- $x^{(i)}$: ith training data point, $1 \le i \le m$
- μ_k: cluster centroid k, $1 \le k \le K$
- $c^{(i)}$: Index of cluster centroid closest to $x^{(i)}$, $1 \le c^{(i)} \le K$

$$J\left(c^{(1)}, c^{(2)}, \cdots, c^{(m)}, \mu_1, \mu_2, \cdots, \mu_K\right) = \frac{1}{m}\sum_{i=1}^{m} \left\|x^{(i)} - \mu_{c^{(i)}}\right\|^2$$

$$\min_{\substack{c^{(1)}, c^{(2)}, \cdots, c^{(m)} \\ \mu_1, \mu_2, \cdots, \mu_K}} J\left(c^{(1)}, c^{(2)}, \cdots, c^{(m)}, \mu_1, \mu_2, \cdots, \mu_K\right)$$

Local Optimum

Convergence can lead to local optimum when we want global optimum. If we somehow picked a centroid that lead to lowest mean squared distance, get stuck (local optimum). Solution - Try again, then pick the lowest. **K-medoids** - pick points that are furthest from all other points.

Choosing Number of Clusters

As there are more clusters, loss decreases. If all the points are each their own clusters, the loss will be 0. Mean squared distance between point and centroid decreases as number of cluster increases.

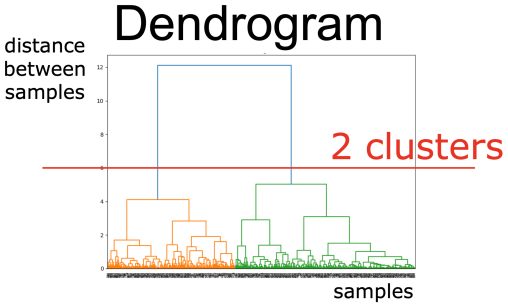
- Elbow method - find point of inflexion on loss-k graph. Use that value as k
- Business need - depending on context, the value of k is fixed.

Hierachical Clustering

Instead of picking k, let some algo pick it for us instead. Idea :

- Every data point is a cluster
- Find clusters that are closest to each other and group them into one new cluster
- To find closest - use feature scaling or mean normalisation
- Eventually end up with 1 big cluster

Everytime we reduce the number of clusters by 1, eventually converge to 1. Then to choose, look at dendrogram. Draw a threshold horizontal line. Count the number of big clusters below the line.



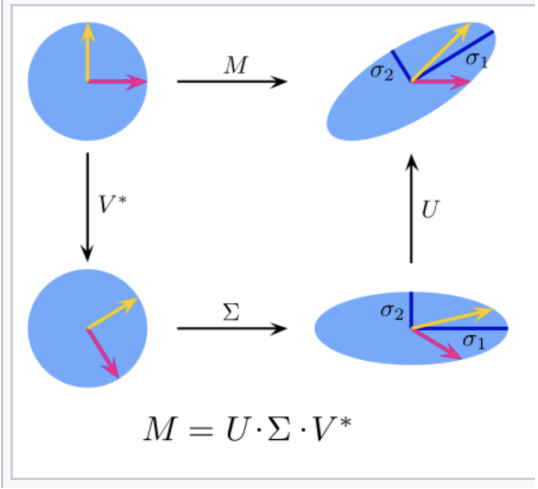
Result may still show overlap because points are clustered in higher dimensions. Problems :

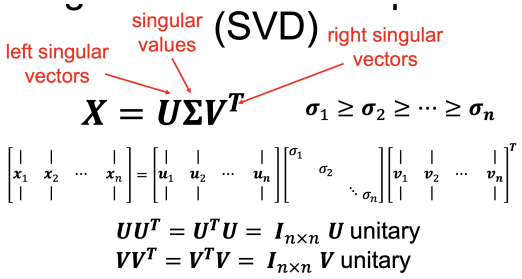
- computational complexity
- long training times
- higher risk of inaccuracy
- human intervention on output
- lack of transparency on how data were clustered

Dimensionality Reduction

Idea : reduce number of features to improve performance. Then convert it back. Kernel tricks makes things separable in higher dimensions. Modify kernel trick to output something in fewer dimensions instead.

Singular Value Decomposition

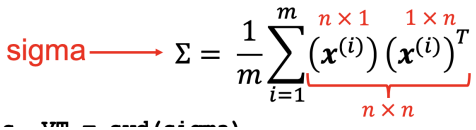




Unitary means that it matrix multiplication of itself with its transpose gives the identity matrix. Singular values in this case refers to the weights.

Principal Component Analysis

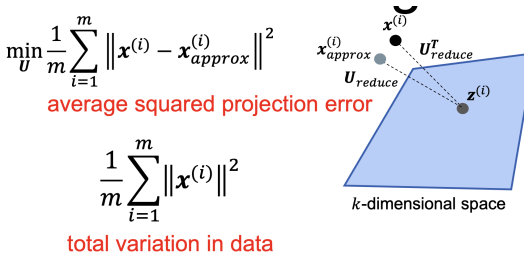
How to project all points onto lower dimension and preserve most of the information. Compute the covariance matrix and pass it throuugh SVD. Then we select the first k vectors (ie the first k columns). To obtain the lower dimension features, multiply the result with the original matrix.



Exploit the fact that the result of selecting the vectors produces a matrix of shape n x k. Transposing it and multiplying it with a data point (n x 1) will result in k x 1. Which is in a lower dimension than the data point input.

Intuition

Project the data point onto lower dimensional space. Then project the point back into higher dimensions to get the original back (hopefully). There may be some distance between the original and approximate, referred to as the error. Aim to minimise the sum of this distance for all points.



Choose **minimum** k such that average squared error divided by total variation is smaller than 1%

Applying

- Apply to training set
- Once we get the reduced matrix, apply to test or cross validation sets
- **PCA is not meant to resolve overfitting!**

Reconstruction

To get the approximate original back, multiply the reduced matrix with the lower dimension matrix. Result is approximately the same since we did not preserve all the vectors in the lower dimension. It is possible to identify useless features based on the size of σ. The bigger the values of σ chosen, the easier it will be to recover the **original** matrix.