Introduction to AI

Agents and Environments

An **agent** is anything that can be viewed as perceiving its environment through **sensors** and acting upon that environment through **actuators**. The environment, in practice, is the part of the universe whose state we care about when designing the agent. We use the term **percept** to refer to the content an agent's sensors are perceiving. A **percept sequence** is the complete history of everything the agent has ever perceived.

Performance Measures

We can use a performance measure to evaluate any given sequence of environment states. As a general rule, it is better to design performance measures according to what one actually wants to be achieved in the environment, rather than according to how one thinks the agent should behave.

Rational Agent

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

Note that rationality is not omniscience nor perfection. Rationality maximizes *expected* performance, while perfection maximizes *actual* performance.

Nature of Environments

We can specify the task environment using the **PEAS** (Performance Measure, Environment, Actuators, Sensors) framework.

Properties of Task Environment

- **Fully observable vs Partially observable.** Fully observable if the sensors detect all aspects that are relevant to the choice of action. Relevance depends on performance measure.
- **Single-agent vs Multiagent.** Agent refers to any object with behaviour best described as maximising a performance measure whose value depends on your agent's behaviour.
- **Deterministic vs Non-deterministic vs Stochastic.** If the next state is completely determined by the current state and the action executed by the agent(s), it is deterministic. It may *appear* to be non-deterministic if the environment is partially observable. Stochastic is only if the model deals with probabilities, whereas non-deterministic doesn't need to be quantified.
- **Episodic vs Sequential.** If episodic, the agent's experience is divided into atomic episodes. The next episode does not depend on the actions taken in previous episodes. In sequential environments, current decision can affect all future decisions.
- **Static vs Dynamic.** Dynamic if the environment can change while an agent is deliberating.
- **Discrete vs Continuous.** Discrete means distinct and clearly defined. Applies to the *state* of the environment, to how *time* is handled, and to the *percepts* and *actions* of the agent.

Structure of Agents

- **Table driven agent.** Just index a lookup table with percept sequence.
- **Simple reflex agent.** Selects action based on current percept, ignoring percept history. Uses condition-action (if-then) rules
- **Model-based reflex agent.** Keep track of the part of the world it can't see now. This internal state is updated through the *transition model* of the world (i.e. knowledge on how the world changes) and the *sensor model*, i.e. information from percepts.
- **Goal-based agent.** In addition to tracking the state of the world, also track a set of goals, then pick the action that brings it closer to the goal.
- **Utility-based agent.** Uses an utility function to assign a score to any given percept sequence, i.e. an internalisation of the performance measure. If the utility function and the performance measure are aligned, then the agent will be rational.
- **Learning agent.** Uses a *performance element* to select external actions, a *critic* to give feedback on how the agent is doing and how the performance element can be improved, a *learning element* responsible for making improvements, and a *problem generator* that suggests actions that will lead to new and informative experiences.

Solving Problems by Searching

A search problem can be defined formally as:

- **State Space.** Set of possible states the environment can be in.
- **Initial State.** Where the agent starts.
- **Goal State(s).** Can be more than one.
- **Actions.** Given a state s, $\textsc{Actions}(s)$ returns a finite set of actions that can be executed in s.
- **Transition Model.** Describes what each action does. $\textsc{Result}(s,a)$ returns the state that results from doing action a in state s.
- **Action Cost Function.** $\textsc{Action-Cost}(s,a,s')$ or $c(s,a,s')$ which gives a numeric cost of applying action a in state s to reach s'.

Searching in General

Best-First Search

Pick a node n from the frontier with minimum value of an evaluation function $f(n)$. If goal state, return it, else expand to add its child nodes to the frontier. Child nodes are only added if they are unvisited or previously visited with a higher path cost.

Search Data Structures

A *node* generally has: *node*.$\textsc{State}$, *node*.$\textsc{Parent}$, *node*.$\textsc{Action}$ (action applied to the parent's state to get this node), *node*.$\textsc{Path-Cost}$ (total cost of path from initial state to this node, i.e. $g(node)$). The frontier can be a priority queue, a normal queue, or a stack.

Redundant Paths

For repeated nodes, we can either remember all visited nodes in a hashmap from state to node (graph search), don't care about them (for problems where it's rare for two paths to reach the same state) (tree-like search), or check for cycles by following up the chain of parents.

Problem Solving Performance

- **Completeness.** Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- **Cost optimality.** Does it find a solution with the lowest path cost of all solutions?
- **Time complexity.** How long does it take?
- **Space complexity.** How much memory is needed?

Uninformed Search Strategies

An uninformed search algorithm is given no clue about how close a state is to the goal(s).

Breadth-First Search

Equivalent to Best-First Search with $f(n) =$ depth of the node. Optimisations include: using a normal queue, using a set to track visited nodes, and **early goal test**, i.e. check whether a node is a solution as soon as it is generated.

Assume all nodes have b successors. BFS is complete if b is finite, has time and space complexities of $1+b+b^2+\cdots+b^d = O(b^{d+1})$, and is optimal if cost is 1 per step.

Uniform-Cost Search or Dijkstra's

Expand the least-cost unexpanded node using a priority queue. Equivalent to Best-First Search with $f(n) =$ path cost (from initial state), and equivalent to Breadth-First Search if all action costs are equal.

Let C^* be the cost of the optimal solution, and $\epsilon > 0$ be a lower bound on the cost of each action. UCS is complete if all action costs are $> \epsilon > 0$, and has worst-case time and space complexities of $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d. It is cost-optimal.

Depth-First Search

Can be incomplete if there is an infinite path. Time complexity of $O(b^m)$ where m is the maximum depth of the tree. But space complexity is only $O(bm)$. We can optimise it further with backtracking search, where only one successor is generated at a time, i.e. $O(m)$ space. But DFS in general is not optimal.

Depth-Limited Search

Instead of searching infinitely, we set a depth limit of l. Time complexity is $O(b^l)$ and space complexity is $O(bl)$. If l is poorly selected, then the algorithm is incomplete. A good depth limit is the diameter of the graph, but it's not always known. The number of nodes generated is $N(DLS) = b^0 + b^1 + \cdots + b^d$.

Iterative Deepening Search

Try all possible depth limits from 0 until solution is found OR the failure value is returned from the depth-limited search. If a solution exists, memory requirements is $O(bd)$ and time complexity is $O(b^d)$. Else, memory is $O(bm)$ and time complexity is $O(b^m)$. It is complete, and is cost-optimal if step cost is equal throughout. The number of nodes generated is $N(IDS) = (d+1)b^0 + (d)b^1 + (d-1)b^2 + \cdots + (1)b^d$.

Bidirectional Search

We can search simultaneously from the initial state and backwards from the goal state(s). Motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d. We thus need to track multiple frontiers and expand the node with the minimum value of the evaluation function, and must be able to "traverse backwards" from the goal state(s).

Informed (Heuristic) Search

We can apply domain-specific hints about the location of goals using a heuristic function $h(n) =$ estimated cost of the cheapest path from the state at node n to a goal state. Example in route-finding: the straight-line distance.

Greedy Best-First Search

Form of Best-First Search that expands first the node with the lowest $h(n)$ value, which is the node that appears to be the closest to the goal, i.e. $f(n) = h(n)$.

It is not complete as it is possible to go into loops. Worst-case time and space complexities are $O(b^m)$ or $O(|V|)$, but with a good heuristic function, it may be possible to have $O(bm)$. It is not cost-optimal.

A* Search

This is a Best-First Search that uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost from the initial node to node n.

Assuming all action costs are $> \epsilon > 0$ and that state space is finite, A* search is complete. If there are infinitely many nodes with $f(n) \leq f(goal)$, then it is not complete. Time and space complexities are both exponential $O(b^d)$ for a poor heuristic. Whether it is cost-optimal depends on certain properties of the heuristic:

- **Admissibility.** Never overestimates the cost to reach a goal, i.e. optimistic. If $h(n)$ is admissible, then A* using **tree-like search** is cost-optimal.
- **Consistency.** Stronger than admissibility. $h(n)$ is consistent if, for every node n and every successor n' of n generated by an action a, we have $h(n) \leq c(n,a,n') + h(n')$, i.e. triangle inequality. If $h(n)$ is consistent, then A* using **graph search** is cost-optimal.

Memory-Bounded Search

Memory is split between the frontier and the reached states. Some basic ways to save space is to store a node either in the frontier or in *reached*, to remove states from *reached* when we can prove they are no longer needed e.g. by prohibiting U-turns, or by reference counting, e.g. a node with four neighbours visited four times can be removed. Other new algorithms include:

- **Beam search.** Limit frontier size to top k best f-scores, or limit to nodes with f-scores within δ of the best f-score. May be incomplete and suboptimal, but fast and saves space.
- **Iterative-deepening A* search (IDA*).** Similar to IDS, but instead of using depth as the limit, we use the smallest f-cost of any node that exceeded the cutoff on the previous iteration. If all nodes have distinct f-costs, then each iteration might only cover one new node.
- **Recursive best-first search (RBFS).** Like recursive DFS, but also tracks the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this value, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with the backed-up value, i.e. the best f-value of its children. RBFS is optimal if $h(n)$ is admissible. Space complexity is linear in the depth of the deepest optimal solution. Time complexity is harder to characterise.

IDA* and RBFS use too little memory, even if more memory is available. They may end up reexploring the same states many times over. We can thus employ:

- **Memory-bounded A* (MA*).** Not covered.
- **Simplified MA* (SMA*).** Expand newest best leaf until memory is full. Then drop the oldest node with the worst f-value and backup the forgotten node's value to its parent. SMA* will thus only regenerate that subtree only when all other paths have been shown to look worse than the path it has forgotten.

Heuristics

Some ways of characterising the quality of a heuristic:

- **Effective branching factor.** We can compute b^* from N nodes generated and solution depth b using $N+1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$. The closer to 1 b^* is, the better.
- **Effective depth.** Reduce depth by constant k_h, i.e. $O(b^{d-k_h})$ vs $O(b^d)$ for uninformed search.

If h_2 dominates h_1, i.e. for any node n, $h_2(n) \geq h_1(n)$, then A* using h_2 will never expand more nodes than using h_1. To generate heuristics, we can **relax** the problem, i.e. remove certain restrictions on the actions.

Local Search

Sometimes, we only care about finding the final state. Local search algorithms search without tracking paths nor reached states, and can be used to solve optimisation problems, where the aim is to find the best state according to an **objective function**.

Hill-Climbing Search / Greedy Local Search

Find local maxima by travelling to neighbouring states with the highest value, and terminating when no neighbour has a higher value. Easy objective function is to negate the heuristic function.

This algorithm suffers from local maxima (i.e. non-global maximum), ridges (sequence of local maxima), plateaus (sequence of same values, but is local maxima) and shoulders (same values, but progress is possible). Solutions are:

- **Sideways Move.** We do a limited number of consecutive sideways move, in hopes that the plateau is really a shoulder.
- **Stochastic Hill Climbing.** Chooses a random uphill move, with probabilities based on the steepness of the move.
- **First-choice Hill Climbing.** Randomly generate successors until one is better than the current state. Useful when a state has e.g. thousands of successors.
- **Random-Restart Hill Climbing.** Perform a fixed number of steps from some randomly generated initial steps, then restart if no maximum found.

Simulated Annealing

Basically, randomly pick a next move. If it's a better state, go for it, else accept it with a probability less than 1, and this probability decreases exponentially with the "badness" of the move. Idea is to escape local maxima by allowing some random moves once in a while.

Local Beam Search

by Hanming Zhu

Pick k random initial states, then generate their successors. If goal is found, terminate, else pick the k best successors and repeat. Similar to k random restarts but information is shared. It may suffer from a lack of diversity if the k states start to cluster. **Stochastic beam search** alleviates this problem by choosing successors with probability proportional to their value.

Genetic Algorithms

- Each individual is a string over a finite alphabet (often a Boolean string).
- We have a mixing number, which is the number of parents that combine to form offspring, commonly two.
- Selection process determines who gets to be parents, one way is to assign probability proportional to fitness score.
- A recombination procedure. Common approach is to randomly select a crossover point to split the parent strings, then mix the four parts to form two children.
- Mutation rate. Once an offspring is generated, every bit in its composition is flipped with probability equal to the mutation rate.
- The next generation can be purely offspring, or may also include parents (elitism).

Adversarial Search and Games

The most commonly studied games are deterministic, two-player, turn-taking, perfect information, zero-sum games. Perfect information just means fully observable, and zero-sum means that there is no "win-win" situation.

- **Initial State.** S_0.
- **Player.** To-Move(s) tells us whose turn it is in s.
- **Actions.** Actions(s) gives us a set of legal moves in s.
- **Transition Model.** Result(s, a) returns the state that results from doing action a in state s.
- **Terminal State(s).** Is-Terminal(s) is true when the game is over.
- **Utility Function.** Utility(s, p) tells us the final numeric value to player p when the game ends in terminal state s.

Minimax Search Algorithm

Given a game tree, we can work out the **minimax value** of each state in the tree, which is the utility of being in that state, *assuming that both players play optimally* from there to the end of the game.

$$\text{MM}(s) = \begin{cases} \text{Utility}(s, \text{max}), & \text{if Is-Terminal}(s) \\ \max_{a \in \text{A}(s)} \text{MM}(\text{R}(s, a)), & \text{if T-M}(s) = \text{max} \\ \min_{a \in \text{A}(s)} \text{MM}(\text{R}(s, a)), & \text{if T-M}(s) = \text{min} \end{cases}$$

In terms of algorithm, it is a recursive one that backs up the minimax values through the tree as the recursion unwinds. As such, like DFS, it has time complexity of $O(b^m)$, space complexity of $O(bm)$, and is complete if tree is finite.

Alpha-Beta Pruning

We can prune the tree in a way that doesn't affect the outcome. α is the best choice found so far along search path we have for max, while β is the best choice for min. If we are at a min node, we can stop checking our successors once we find a node that is smaller than or equal to α, since that upper bounds all other values found subsequently. If at a max node, then we can stop if we find a node that is larger than or equal to β.

Note that the values of α and β are updated recursively, i.e. at a node, I will first start with $-\infty, +\infty$, and I will update my α in my max nodes, and my β in my min nodes.

Move ordering matters as well. If the best successor for a node is generated first, then we can prune the remaining successors. If done perfectly, alpha-beta would only need to examine $O(b^{m/2})$ nodes instead of $O(b^m)$, i.e. effective branching factor becomes $\sqrt{b}$, or that we can examine a tree twice as deep in the same time. With random move ordering, we have roughly $O(b^{3m/4})$ for moderate b.

One way to optimise this is via **transpositions**, i.e. cache previously seen states in a transposition table, since different permutations of the move sequence can end up in the same position.

Heuristic Alpha-Beta Tree Search

This is a **Type A** strategy, where we go wide but shallow. We replace our utility function with an **evaluation function** that estimates the expected utility of a state. For terminal states, Eval(s, p)=Utility(s, p), and for nonterminal states, Utility($loss, p$) $\leq$ Eval(s, p) $\leq$ Utility(win, p). We also replace Is-Terminal(s) with Is-Cutoff(s, d), where d is search depth of s, and returns true if a state is a terminal state, and either true or false for any other states.

Evaluation Function

Use the features of the state to compute an expected utility value. For chess, it's typically the linear weighted sum: Eval(s)$= w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$. Features are e.g. number of rooks, etc. Assumption is that the contribution of each feature is independent of the values of other features, which is not always true. For this reason, many programs use nonlinear combinations, e.g. a bishop may be worth more in endgame, so two features (move number, number of remaining bishops) are combined.

Cutting Off Search

Naive way is to cut off past a certain depth. We can then do iterative deepening. If we keep entries in the transposition table, subsequent rounds will get faster and we can use the evaluations to improve move ordering.

Machine Learning

Machine learning refers to when a computer observes some data, builds a model based on that data, and uses that model as both a **hypothesis** about the world and a piece of software that can solve problems.

Types of Problems

- **Classification.** Output is one of a finite set of values.
- **Regression.** Output is a number.

Types of Feedback

- **Supervised Learning.** Agent observes input-output pairs and learns a function that maps from input to output.
- **Unsupervised Learning.** Agent learns patterns in the input without any explicit feedback. Most common task is clustering.
- **Reinforcement Learning.** Agent learns from a series of reinforcements: rewards and punishments.

Supervised Learning

Given a training set of N example input-output pairs $(x_1, y_1), (x_2, y_2), \cdots (x_N, y_N)$, where each pair was generated by an unknown function $y = f(x)$, discover a function h (hypothesis) that approximates the true function f. The true measure of a hypothesis is how well it handles inputs it has not yet seen, i.e. the **test set**.

Learning Decision Trees

A decision tree is a representation of a function that maps a vector of attribute values to a single output value — a "decision". It reaches this by performing a sequence of tests, starting at the root and following the appropriate branch until a leaf is reached. Each internal node corresponds to a test, and each leaf specifies the value to be returned.

Expressiveness

A Boolean decision tree is equivalent to a logical statement of the form $Output \Leftrightarrow (Path_1 \vee Path_2 \vee \cdots)$ where each $Path_i$ is a conjunction of the form $(A_m = v_x \wedge A_n = v_y \wedge \cdots)$ of attribute-value tests corresponding to a path from the root to a true leaf. Any function in propositional logic can be expressed as a decision tree.

For Boolean functions with n Boolean attributes, the truth table will have 2^n rows, and each row can output either true or false, so there are 2^{2^n} possible functions / distinct decision trees.

Choosing attribute tests

We can learn a decision tree from examples, one that is simpler than the true decision tree. The learning algorithm is a recursive divide-and-conquer algorithm:

1. If remaining examples are all true or false, then we return that value.
2. If there is a mix, then we choose the best attribute (highest importance) to split them and recurse.
3. If there are no examples left, then we return the most common value from the node's parent's examples.
4. If there are no attributes left for splitting, then we return the most common value of the current examples.

To choose the attribute with the highest importance, we need to use the notion of information gain, which is defined in terms of **entropy**, which, for a random variable V with values v_k having probability $P(v_k)$ is $H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$. The result is in number of bits.

For boolean variables, $B(q) = -(q \log_2 q + (1-q) \log_2 (1-q))$ where q is the probability of it being true. If the training set contains p positive examples and n negative examples, then the entropy of the output variable on the entire set is $B(\frac{p}{p+n})$.

An attribute A with d distinct values divides the training set E into subsets $E_1, \cdots, E_d$. Each subset E_k has p_k positive examples and n_k negative examples. We define $Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p_n} B(\frac{p_k}{p_k + n_k})$.

The information gain from the attribute test on A is the expected reduction in entropy: $Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$. The most important attribute is the one with the largest information gain.

Generalisation and Overfitting

Overfitting becomes more likely as the number of attribute grows, and less likely as we increase the number of training examples. We can thus do **decision tree pruning**, by eliminating nodes that are clearly not relevant. We start by looking at a test node that only has leaf nodes as descendants. If the test seems irrelevant, we replace it with a leaf node, and we repeat this process.

We can detect that a node is irrelevant by using a statistical significance test. We assume there is no underlying pattern (null hypothesis), then we calculate the extent to which they deviate from a perfect absence of pattern. If statistically unlikely (5% or less), then it means there is a pattern. This can be done via χ^2 tests, also known as χ^2 pruning.

Other issues

- **Missing data.** If node n tests A, then assign most common value of A at node n to the example with missing data for A and is at node n. Or we can further filter by most common value amongst examples with same output. Or we can assign each possible value of A some probability, then we split the examples with missing data according to this.
- **Continuous and multivalued input attributes.** We can use inequality tests on the value of an attribute. For attributes with a large number of possible values but are not continuous nor have a meaningful ordering, we can replace gain with information gain ratio instead, where $SplitInformation(A) = -\sum_{i=1}^{d} \frac{|E_i|}{|E|} \log_2 \frac{|E_i|}{|E|}$ and $GainRatio(A) = \frac{Gain(A)}{SplitInformation(A)}$.
- **Attributes with differing costs.** Assume that there are costs incurred to obtain some attributes, e.g. costs of biopsies, etc. We can learn a consistent decision tree with low expected cost by replacing gain with $\frac{Gain^2(A)}{Cost(A)}$ or $\frac{2^{Gain(A)} - 1}{(Cost(A)+1}^{\omega}$ where $\omega \in [0, 1]$ determines relative importance of cost vs information gain.
- **Continuous-valued output attribute.** We will need a regression tree.

Linear Regression and Classification

In all the below equations, α is the learning rate, which is often scheduled to decrease in magnitude with time (proven to still converge).

Univariate Linear Regression

Form of $y = \theta_1 x + \theta_0$, where x is the input and y is the output, and θ_0 and θ_1 are real-valued coefficients to be learnt. w may also be used instead of θ.

We also use the squared-error loss function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{j=1}^{m} (h_\theta(x_j) - y_j)^2$ where m is the number of examples. This sum is minimized when its partial derivatives with respect to w_0 and w_1 are zero: $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = 0$ and $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = 0$. There is a unique solution if these partial derivatives are zero: $\theta_1 = \frac{m(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{m(\sum x_j^2) - (\sum x_j)^2}$ and $w_0 = (\sum y_j - w_1(\sum x_j))/m$.

Gradient Descent

We can't always find partial derivatives that are zero. Gradient descent (i.e. reverse hill climbing) allows us to compute an estimate of the gradient at each point and move a small amount in the steepest downhill direction, until we converge on a point with minimum loss. Since the loss surface is convex, we will always arrive at the global minimum.

$$\theta_0 \leftarrow \theta_0 - \alpha \frac{1}{m} \sum_{j=1}^{m} (h_\theta(x_j) - y_j)$$

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{1}{m} \sum_{j=1}^{m} (h_\theta(x_j) - y_j) \times x_j$$

This is batch gradient descent. Stochastic gradient descent will randomly select a small number of training examples at each step, so it's faster, but may need more steps.

Multivariable Linear Regression

We can extend the above to n-element vectors: $h_\theta(x_j) = \theta_0 + \sum_i \theta_i x_{j,i}$ $J(\theta_0, \cdots, \theta_n) = \frac{1}{2m} \sum_{j=1}^{m} (h_\theta(x_j) - y_j)^2$.

$$\theta_i \leftarrow \theta_i - \alpha \frac{1}{m} \sum_{j=1}^{m} (h_\theta(x_j) - y_j) \times x_{j,i}$$

It is possible to solve this analytically. Let $\mathbf{y}$ be the vector of outputs, and $\mathbf{X}$ be the data matrix. Then the vector of predicted outputs is $\hat{\mathbf{y}} = \mathbf{X}\theta$ and squared-error loss is $L(\theta) = ||\hat{\mathbf{y}} - \mathbf{y}||^2 = ||\mathbf{X}\theta - \mathbf{y}||^2$. We set the gradient to 0, and rearrange to get the minimum-loss weight vector $\theta = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}$, also known as the normal equation.

We can also do feature scaling via mean normalisation, i.e. replace $x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i}$. We can also do polynomial regression, i.e. let $x_2 \leftarrow x_1^2$, and so on. Or even $\sqrt{x}$.

Linear Classifiers with Hard Threshold

Linear functions can also be used to do classification. A decision boundary is a line or surface that separates two classes, and a linear decision boundary is called a linear separator. We can thus have the classification hypothesis $h_\theta(x) = 1$ if $\theta_0 + \theta_1 x_1 + \theta_2 x_2 > 0$ and 0 otherwise. Since now h_θ spits out 1s and 0s, we cannot use our previous methods anymore. Our weight update rule is now $\theta_i \leftarrow \theta_i + \alpha(y - h_\theta(x)) \times x_i$. This is the perceptron learning rule and works if the data is linearly separable.

Linear Classification with Logistic Regression

The threshold function does have some problems, since it's not differentiable and is discontinuous. It also cannot classify some examples as unclear borderline cases. We can soften the threshold function by approximating it with a continuous, differentiable function — the logistic function $Logistic(z) = \frac{1}{1+e^{-z}}$. We thus have $h_\theta(x) = \frac{1}{1+e^{-\theta^\intercal \cdot x}}$. Then, we can have $J(\theta) = \frac{1}{m} \sum_{j=1}^{m} Cost(h_\theta(x), y_i)$ where

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

$= -y \log h_\theta(x) - (1-y) \log(1 - h_\theta(x))$. The gradient descent equation remains the same. We interpret $h_\theta(x)$ as the estimated probability that $y = 1$ on input x.

For multi-class classification, we can train n different logistic classifiers for all n classes, then for each input, we pick the class i with the maximum output from their hypothesis.

by Hanming Zhu