Agents

Rational agent has goals or preferences and tries to perform a series of actions that yield the best/optimal expected outcome given these goals.
A **reflex** agent is one that doesn't think about the consequences of its actions, but rather selects an action based solely on the current state of the world.

- **Fully Observable** vs. **Partially Observable**, **Single Agent** vs. **Multi Agent**, **Deterministic** vs. **Stochastic**
- **Episodic** vs. **Sequential**: Current decision does not affect later decision.
- **Static** vs. **Dynamic**: If the environment can change while an agent is deliberating, we say the environment is dynamic for that agent; otherwise, it is static.
- **Discrete** vs. **Continuous**, **Known** vs. **Unknown**

Simple Reflex Agents, Model-based Reflex Agents (percept history), Goal-based Agents, Utility-based Agents

Search

Generally, environment is observable, discrete, known and deterministic.

Uninformed Search

- **Depth-First Search**: selects *deepest* fringe node, last-in, first-out (LIFO) stack, not complete, not optimal, $O(b^m)$ time, $O(bm)$ space.
- **Breadth-First Search**: selects the *shallowest* fringe node, first-in, first-out (FIFO) queue, complete, optimal is if all edge costs are equivalent, $O(b^s)$ time and $O(b^s)$ space with s is the depth of the goal.
- **Uniform Cost Search**: selects the *lowest cost* fringe node, heap-based priority queue, weight for a given enqueued node v is the path cost from the start node to v, or the *backward cost* of v, complete, optimal if we assume all edge costs are nonnegative, $O(b^{C^*/\varepsilon})$ time and $O(b^{C^*/\varepsilon})$ space with C^* is the optimal path cost and ε is the minimal cost between two nodes.

Informed Search

- **Greedy Search**: selects the fringe node with the *lowest heuristic value*, identical to UCS but use *estimated forward cost*, not complete, not optimal.
- **A* Search**: selects the fringe node with the *lowest estimated total cost*, combines the total backward cost used by UCS with the estimated forward cost used by greedy search, complete, optimal.

Admissibility

$$\forall n, 0 \le h(n) \le h^*(n)$$

Optimal for tree search

Consistency

$$\forall A, C, h(A) - h(C) \le cost(A, C)$$

Optimal for graph search

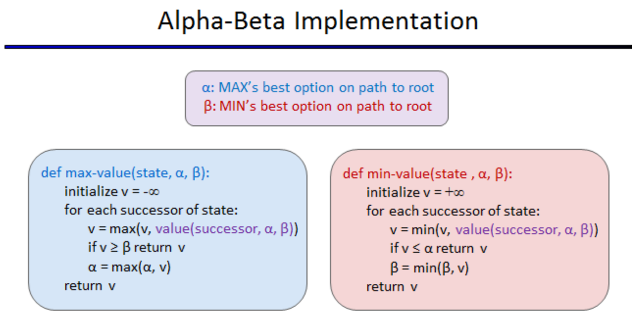
Games

Minimax

Execution is very similar to depth-first search and it's time complexity is identical, a dismal $O(b^m)$

Alpha-Beta Pruning

Can have runtime as good as $O(b^{m/2})$.


Alpha-Beta Implementation

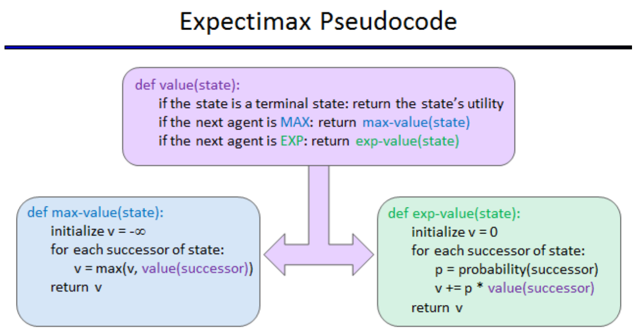
α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Expectimax

We can't prune in the same way that we could for minimax. However, pruning can be possible when we have known, finite bounds on possible node values.


Expectimax Pseudocode

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Utilities

A lottery is a situation with different prizes resulting with different probabilities. To denote lottery where A is received with probability p and B is received with probability $(1-p)$, we write

$$L = [p, A;\ (1-p), B]$$

Utility function U

$$U(A) \ge U(B) \leftrightarrow A \succcurlyeq B$$

$$U([p_1, S_1; \dots;\ p_n, S_n]) = \sum_i p_i U(S_i)$$

Markov Decision Processes

A maximization of the following **additive utility** function:

$$U([s_0, a_0, s_1, a_1, s_2, \dots]) \\ = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) \\ + R(s_2, a_2, s_3) + \cdots$$

Or **discounted utility**:

$$U([s_0, a_0, s_1, a_1, s_2, \dots]) \\ = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) \\ + \gamma^2 R(s_2, a_2, s_3) + \cdots \\ = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \le \sum_{t=0}^{\infty} \gamma^t R_{max} \\ = \frac{R_{max}}{1-\gamma}$$

The Bellman Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

$$V^*(s) = \max_a Q^*(s, a)$$

Value Iteration

1. $\forall s \in S$, initialize $V_0(s) = 0$.
2. Repeat the following update until convergence: $\forall s \in S, V_{k+1}(s)$
$$\leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') \\ + \gamma V^*(s')]$$

Running time per iteration is $O(S^2 A)$.

Policy Extraction

Efficiency: $O(S^2)$ per iteration.

$$\forall s \in S, \pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Policy Iteration

1. Define an initial policy. This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until convergence:
   - Evaluate the current policy with **policy evaluation**. $V^\pi(s)$
   - Generate a better policy using policy improvement

Reinforcement Learning

Solving Markov decision processes is an example of **offline planning**, where agents have full knowledge of both the transition function and the reward function, all the information they need to precompute optimal actions in the world encoded by the MDP without ever actually taking any actions.

Model-Based Learning

Model-Free Learning

Direct evaluation and temporal difference learning fall under a class of algorithms known as **passive reinforcement learning** and **on-policy learning**. Q-learning falls under a second class of model-free learning algorithms known as **active reinforcement learning** and **off-policy learning**.

Direct Evaluation

Temporal Difference Learning

$$sample = R(s, \pi(s), s') + \gamma V^\pi(s')$$
$$V^\pi \leftarrow (1-\alpha)V^\pi(s) + \alpha \times sample$$

Q-Learning

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \times sample$$

Approximate Q-Learning

With **feature vectors**, we can treat values of states and q-states as **linear value functions**:

$$V(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \cdots + w_n \cdot f_n(s) = \vec{w} \cdot \vec{f}(s)$$
$$Q(s,a) = w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \cdots + w_n \cdot f_n(s,a) = \vec{w} \cdot \vec{f}(s,a)$$

Update rule

$$difference = \left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right] - Q(s,a)$$
$$w_i \leftarrow w_i + \alpha \times difference \times f_i(s,a)$$
$$Q(s,a) \leftarrow Q(s,a) + \alpha \times difference$$

Exploration and Exploitation

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \times \left[R(s,a,s') + \gamma \max_{a'} f(s',a')\right]$$
$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$$

Regret

Regret is a measure of your total mistake cost: the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards.

Constraint Satisfaction Problems

Filtering

Forward checking

Whenever a value is assigned to a variable X_i, prunes the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned.

Arc consistency

An arc $X \rightarrow Y$ is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint.

The AC-3 algorithm has a worst case time complexity of $O(ed^3)$, where e is the number of arcs (directed edges) and d is the size of the largest domain.

Ordering

- *Minimum Remaining Values (MRV)*: Chooses whichever unassigned variable has the fewest valid remaining values (the most constrained variable).
- *Least Constraining Value (LCV)* - Select the value that prunes the fewest values from the domains of the remaining unassigned values.

Tree-structured CSP

Reduce the runtime for finding a solution from $O(d^N)$ all the way to $O(nd^2)$. If no solution containing the current assignment exists on a tree structured CSP, then enforcing arc consistency will always result in an empty domain.

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Using cutset of size c, the running time is $O(d^c(n-c)d^2)$.

Local Search

Iterative improvement - start with some random assignment to values then iteratively select a random conflicted variable and reassign its value to the one that violates the fewest constraints until no more constraint violations exist.

Logic

Conjunctive normal form or **CNF** is a conjunction of clauses, each of which a disjunction of literals.

Propositional Logical Inference

1. $A \vDash B$ iff $A \rightarrow B$ is valid.
2. $A \vDash B$ iff $A \wedge \neg B$ is unsatisfiable.

First-Order Logic

Just as ⇒ appears to be the natural connective to use with ∀, ∧ is the natural connective to use with ∃.

DPLL

1. Early termination
2. Pure symbols
3. Unit Clauses

Theorem Proving

1. If our knowledge base contains A and $A \rightarrow B$ we can infer B (**Modus Ponens**)

2. If our knowledge base contains $A \wedge B$ we can infer A or B (**And-Elimination**).

3. If our knowledge base contains A and B we can infer $A \wedge B$ (**Resolution**).

Successor-state axiom

$$F^{t+1} \leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$$

Probability

Probability rules

$$P(a|b) = \frac{P(a,b)}{P(b)}$$

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

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)$$

Conditional Independence

If $x \perp\!\!\!\perp y|z$, then:

$$\forall x, y, z: P(x, y|z) = P(x|z)P(y|z)$$
$$\forall x, y, z: P(x|z, y) = P(x|z)$$

Variables

Given a joint PDF, we can compute any desired probability distribution $P(Q_1 \ldots Q_k|e_1 \ldots e_k)$, three types of variables:

1. Query variables Q_i
2. Evidence variables e_i
3. Hidden variables, which are only in joint distribution.

Bayes Net

Representation

Each node is conditionally independent of all its ancestor nodes in the graph, given all of its parents.

Each node is conditionally independent of all other variables given its Markov blanket. A variable's Markov blanket consists of parents, children, children's other parents.

Given all of the conditional probability tables (CPTs) for a graph, we can calculate the probability of a given assignment using the chain rule:

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|parents(X_i))$$

Polytrees

A polytree is a directed graph with no undirected cycles.

For poly-trees the complexity of variable elimination is **linear in the network size** if you eliminate from the leave towards the roots. Essentially the same theorem as for tree-structured CSPs.

Inference

Calculating some useful quantity from a probability model (joint probability distribution)

Exact Inference

NP-hard in general.

Inference by Enumeration

Compute the query from a joint distribution

Problem: sums of exponentially many products.

Variable elimination

To eliminate a variable X, we:

1. Join (multiply together) all factors involving X.
2. Sum out X.

A **factor** is defined simply as an *unnormalized probability*.

In general, with n leaves, factor of size 2^n

Polytrees

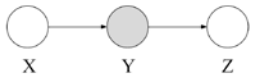
- A polytree is a directed graph with no undirected cycles.
- For poly-trees the complexity of variable elimination is **linear in the network size** if you eliminate from the leave towards the roots. Essentially the same theorem as for tree-structured CSPs.

Approximate Inference

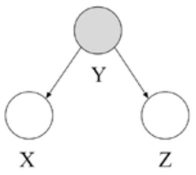
- **Prior sampling**: Simply generate samples based on distribution.
- **Rejection sampling**: Early reject any sample inconsistent with our evidence.
- **Likelihood weighting**: Ensures that we never generate a bad sample. We manually set all variables equal to the evidence in our query. Weight each sample by probability of evidence variables given parents.
- **Gibbs sampling**: Fix evidence. Initialize other variables. Repeat: Choose a non-evidence variable X, resample X from $P(X|markov\_blanket(X))$.

D-Separation

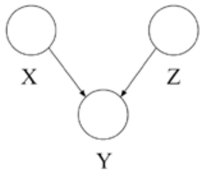
- Causal Chains $\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Z}|\boldsymbol{Y}$:



- Common Cause $\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Z}|\boldsymbol{Y}$:



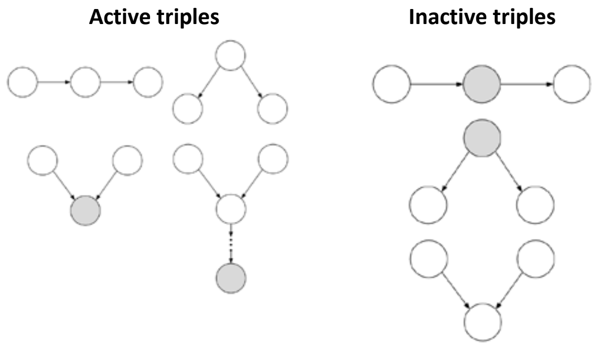
- Common Effect $\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Z}$:



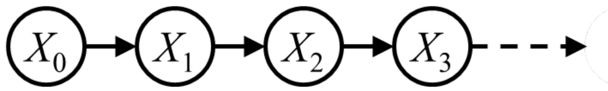
D-Separation algorithm

Given a Bayes Net G, two nodes X and Y, and a (possibly empty) set of nodes $\{Z_1, \ldots, Z_k\}$ that represent observed variables, must the following statement be true: $X \perp\!\!\!\perp Y|\{Z_1, \ldots, Z_k\}$?

1. Shade all observed nodes $\{Z_1, \ldots, Z_k\}$ in the graph.
2. Enumerate all undirected paths from X to Y.
3. For each path:
   a. Decompose the path into triples (segments of 3 nodes).
   b. If all triples are active, this path is active and *d-connects* X to Y.
4. If no path d-connects X and Y, then X and Y are d-separated, so they are conditionally independent given $\{Z_1, \ldots, Z_k\}$



Markov Models



- The **transition model** $P(X_t|X_{t-1})$ specifies how the state evolves over time.
- **Stationarity** assumption: transition probabilities are the same at all times.
- **Markov** assumption: "future is independent of the past given the present".
- Joint distribution: $P(X_0, \ldots, X_T) = P(X_0) \prod_{t=1:T} P(X_t|X_{t-1})$

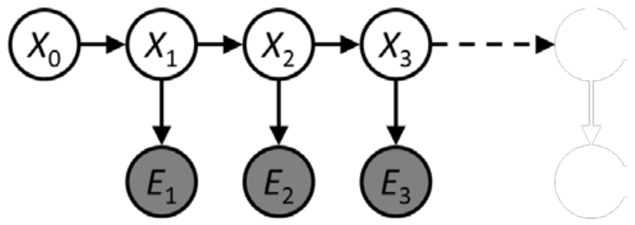
The Mini-Forward Algorithm

$$P(X_{t+1}) = \sum_{x_t} P(X_t = x_t)P(X_{t+1}|X_t = x_t)$$

Stationary Distributions

$$P(X_{t+1}) = P(X_t) = \sum_{x_t} P(X_{t+1}|x_t)P(x_t)$$

Hidden Markov Models



X_i is a **state variable** and E_i is an **evidence variable**.

Joint distribution:
$P(X_0, X_1, \dots, X_T, E_T) = P(X_0) \prod_{t=1:T} P(X_t|X_{t-1}) P(E_t|X_t)$

Belief distribution notations

$$B(X_i) = P(X_i|e_1 \dots e_i)$$
$$B'(X_i) = P(X_i|e_i, \dots, e_{i-1})$$
$$e_{1:t} = e_1, \dots, e_t$$

The Forward Algorithm

Exact inference

1. Time elapse update/Prediction
$$B'(X_{i+1}) = \sum_{x_i} P(X_{i+1}|x_i) B(x_i)$$
2. Observation update/Filtering
$$B(X_{i+1}) \propto P(e_{i+1}|X_{i+1}) B'(X_{i+1})$$

Combine to get:

$$B(X_{i+1}) \propto P(e_{i+1}|X_{i+1}) \sum_{x_i} P(X_{i+1}|x_i) B(x_i)$$

Runtime per time step: $O(|X|^2)$ where $|X|$ is the number of states.

Particle Filtering

Approximate inference

The Hidden Markov Model analog to Bayes' net sampling.

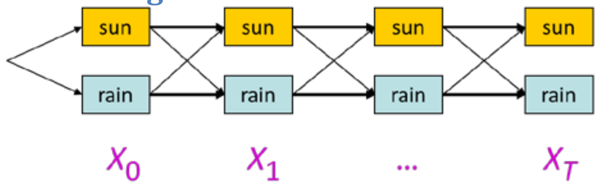
Particle Filtering Simulation

Time Elapse Update - For a particle in state t_i, sample the updated value from the probability distribution given by $P(T_{i+1}|t_i)$.

Observation Update - For a particle in state t_i with sensor reading f_i, assign a weight of $P(f_i|t_i)$. The algorithm for the observation update is as follows:

1. Calculate the weights of all particles as described above.
2. Calculate the total weight for each state.
3. If the sum of all weights across all states is 0, reinitialize all particles.
4. Else, normalize the distribution of total weights over states and resample your list of particles from this distribution.

Viterbi Algorithm



Forward algorithm computes *sums of paths*, Viterbi algorithm computes *best paths*.

Use dynamic programming to solve for the best path.

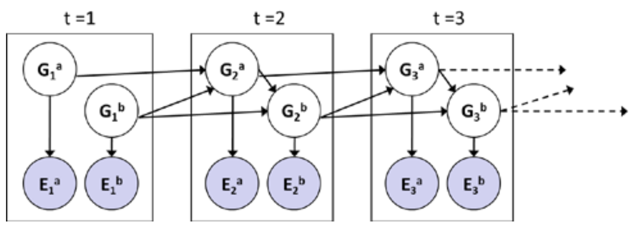
Define $m_t[x_t] = \max_{x_{1:t-1}} P(x_{1:t}, e_{1:t})$, the maximum probability of a path starting at any x_0 and the evidence seen so far to a given x_t at time t. Recurrence relation:

$$m_t[x_t] = P(e_t|x_t) \max_{x_{t-1}} P(x_t|x_{t-1}) m_{t-1}[x_{t-1}]$$

The algorithm consists of two passes: **the first runs forward in time** and computes the probability of the best path to that (state, time) tuple given the evidence observed so far. **The second pass runs backwards in time**: first it finds the terminal state that lies on the path with the highest probability, and then traverses backward through time along the path that leads into this state (which must be the best path).

- Time complexity: $O(|X|^2 T)$
- Space complexity: $O(|X|T)$
- Number of paths: $O(|X|^T)$

Dynamic Bayes Nets (DBNs)



Dynamic Bayes nets are a generalization of HMMs.

Particle Filters

- A particle is a complete sample for a time step
- Initialize: Generate prior samples for the $t = 1$ Bayes net
  - Example particle: $G_1^a = (3,3), G_1^b = (5,3)$
- Elapse time: Sample a successor for each particle
  - Example successor: $G_2^a = (2,3), G_2^b = (6,3)$
- Observe: Weight each entire sample by the likelihood of the evidence conditioned on the sample
  - Likelihood: $P(E_1^a|G_1^a) \cdot P(E_1^b|G_1^b)$
- Resample: Select prior samples (tuples of values) in proportion to their likelihood

Decision Networks

- Chance nodes: ◯
- Action nodes: ▭
- Utility nodes: ◇

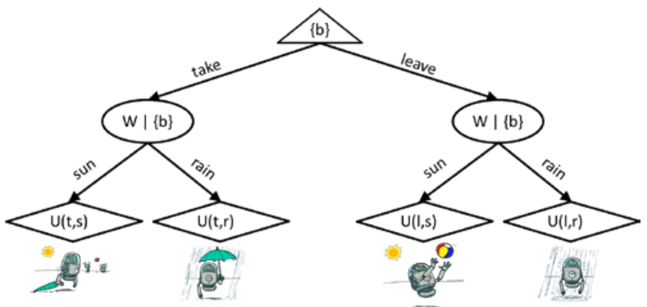
Expected utility:

$$EU(a|e) = \sum_{x_1,\dots,x_n} P(x_1, \dots, x_n|e) U(a, x_1, \dots, x_n)$$

Maximum expected utility:

$$MEU(e) = \max_a EU(a|e)$$

Outcome Trees



The Value of Perfect Information

Maximum expected utility with new evidence e':

$$MEU(e, e') = \max_a \sum_s P(s|e, e') U(s, a)$$

E' is a random variable whose value is unknown, so we don't know what e' will be.

$$MEU(e, E') = \sum_{e'} P(e'|e) MEU(e, e')$$

Value of Perfect Information:

$$VPI(E'|e) = MEU(e, E') - MEU(e)$$

Properties of VPI

- **Nonnegativity.** $\forall E', e: VPI(E'|e) \geq 0$
  Observing new information always allows you to make a *more informed* decision
- **Nonadditivity.** $VPI(E_j, E_k|e) \neq VPI(E_j|e) + VPI(E_k|e)$ in general
  The VPI of observing two new evidence variables is equivalent to observing one, incorporating it into our current evidence, then observing the other.
- **Order-independence.** $VPI(E_j, E_k|e) = VPI(E_j|e) + VPI(E_k|e, E_j) = VPI(E_k|e) + VPI(E_j|e, E_k)$
  Observing multiple new evidences yields the same gain in maximum expected utility regardless of the order of observation.

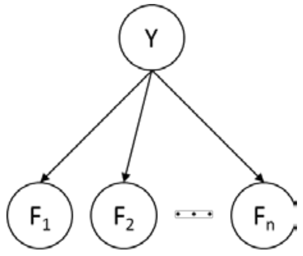
Generally: If $\text{Parents}(U) \perp\!\!\!\perp Z|\text{CurrentEvidence}$, then $VPI(Z|\text{CurrentEvidence}) = 0$.

Machine Learning

The first, **training data**, is used to actually generate a model mapping inputs to outputs. Then, **validation data** (also known as **hold-out** or **development data**) is used to measure your model's performance by making predictions on inputs and generating an accuracy score. If your model doesn't perform as well as you'd like it to, it's always okay to go back and train again, either by adjusting special model-specific values called **hyperparameters** or by using a different learning algorithm altogether until you're satisfied with your results. Finally, use your model to make predictions on the third and final subset of your data, the **test set**. The test set is the portion of your data that's never seen by your agent until the very end of development, and is the equivalent of a "final exam" to gauge performance on real-world data.

- Learn parameters (e.g. model probabilities) on training set
- (Tune hyperparameters on held-out set)
- Compute accuracy of test set
- Very important: never "peek" at the test set!

Naive Bayes Classifier



We have observed values for $F_1, \dots, F_n$, and want to choose the value of Y that has the highest probability conditioned on these features:

$$prediction(f_1, \dots, f_n) = \underset{y}{\operatorname{argmax}}\, P(Y = y|F_1 = f_1, \dots, F_N = f_n)$$
$$= \underset{y}{\operatorname{argmax}}\, P(Y = y) \prod_{(i=1)}^{n} P(F_i = f_i|Y = y)$$

Parameter Estimation

Maximum likelihood estimation (MLE) assumes each sample is **independent, identically distributed (i.i.d.)**.

$$\mathcal{L}(\theta) = P_\theta(x_1, \dots, x_N) = \prod_{i=1}^{N} P_\theta(x_i)$$

The maximum likelihood estimate for q is a value that satisfies the following equation:

$$\frac{\partial}{\partial\theta}\mathcal{L}(\theta) = 0$$

Maximum Likelihood for Naive Bayes

$$\theta = \frac{1}{N_h}\sum_{j=1}^{N_h} f_i^{(j)}$$

Smoothing

Laplace's estimate:

$$P_{LAP,k}(x) = \frac{count(x) + k}{N + k|X|}$$

k is the strength of the prior.

$|X|$ is the number of different values x can take on

Laplace for conditionals

$$P_{LAP,k}(x|y) = \frac{count(x,y) + k}{count(y) + k|X|}$$

Perceptron

Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation
  - o activation$_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$
- If the activation is:
  - o Positive, output +1
  - o Negative, output -1

- A hyperplane is a linear surface that is one dimesion lower than the latent space, thus dividing the surface in two.

Binary Perceptron

1. Initialize all weights to 0: $w = 0$
2. For each training sample, with features $f(x)$ and true class label $y^* \in \{-1, +1\}$, do:
   a. $y = \text{classify}(x)$
   b. If $y \neq y^*$, update weights: $w \leftarrow w + y^* f(x)$
3. If no update, terminate.

Bias

Boundary may not go through the origin.

Multiclass Perceptron

- A weight vector for each class:
  - o w_y
- Score (activation) of a class y:
  - o $w_y \cdot f(x)$
- Prediction highest score wins
  - o $y = \text{argmax}_y\, w_y \cdot f(x)$

Weight update:

If wrong, lower score of wrong answer, raise score of right answer:

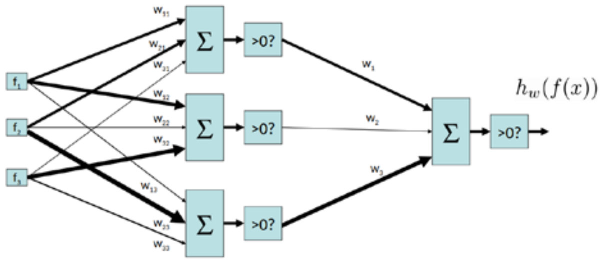
$$w_y = w_y - f(x)$$
$$w_{y^*} = w_{y^*} + f(x)$$

Properties

- **Separability**: true if some parameters get the training set perfectly correct
- **Convergence**: if the training is separable, perceptron will eventually converge (binary case)

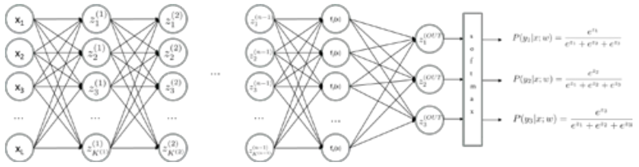
Neural Networks

Multi-layer Perceptron



Multi-layer Feedforward Neural Networks

This is much like the multi-layer perceptron, however, we choose a different non-linearity to apply after the individual perceptron nodes.

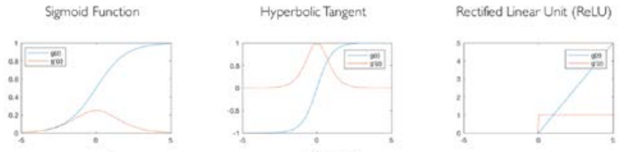


$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g is a nonlinear activation function.

Theorem. (Universal Function Approximators) A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

Common Activation Functions



$$g(z) = \frac{1}{1+e^{-z}} \qquad g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad g(z) = \max(0, z)$$

$$g'(z) = g(z)(1-g(z)) \qquad g'(z) = 1 - g(z)^2 \qquad g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Each application of ReLU to one element can only introduce a change of slope for a single value of x.

The Chain Rule

Mathematically, it states that for a variable z which is a function of n variables $x_1, \dots, x_n$ and each x_i is a function of m variables $t_1, \dots, t_m$, then we can compute the derivative of z with respect to any t_i as follows:

$$\frac{\partial f}{\partial t_i} = \frac{\partial f}{\partial x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{\partial x_2} \cdot \frac{\partial x_2}{\partial t_i} + \dots + \frac{\partial f}{\partial x_n} \cdot \frac{\partial x_n}{\partial t_i}$$

Loss Functions and Multivariate Optimization

The **softmax function** σ defines the probability of classifying $x^{(i)}$ to class j as:

$$\sigma\left(x^{(i)}\right)_j = \frac{e^{f\left(x^{(i)}\right)^T w_j}}{\sum_{(k=1)}^N e^{f\left(x^{(i)}\right)^T w_k}} = P\left(y^{(i)} = j | x^{(i)}\right)$$

Given a vector that is output by our function f, softmax performs normalization to output a probability distribution.

Likelihood function:

$$\ell(\boldsymbol{w}) = \prod_{i=1}^{m} P(y^{(i)} | x^{(i)}; \boldsymbol{w})$$

This expression denotes the likelihood of a particular set of weights explaining the observed labels and datapoints. We would like to find the set of weights that maximizes this quantity.

$$\ell\ell(\boldsymbol{w}) = \log \ell(\boldsymbol{w}) = \sum_{i=1}^{m} \log P(y^{(i)} | x^{(i)}; \boldsymbol{w})$$

To maximize our log-likelihood function, we differentiate it to obtain a **gradient vector** consisting of its partial derivatives for each parameter:

$$\nabla_w \ell\ell(\boldsymbol{w}) = \left[\frac{\partial \ell\ell(\boldsymbol{w})}{\partial \boldsymbol{w_1}}, \dots, \frac{\partial \ell\ell(\boldsymbol{w})}{\partial \boldsymbol{w_n}}\right]$$

Gradient ascent

Gradient ascent is a greedy algorithm that calculates this gradient for the current values of the weight parameters, then updates the parameters along the direction of the gradient, scaled by a step size, α.

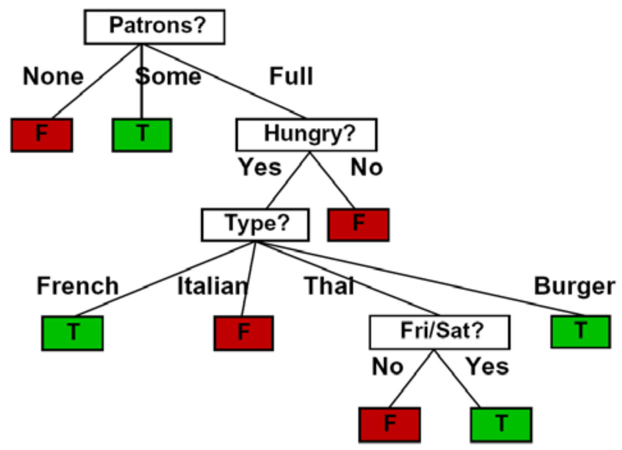
```
Initialize weights $w$
For  $i = 0, 1, 2, \dots$
          $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \nabla_w \ell\ell(\boldsymbol{w})$
```

If rather than minimizing we instead wanted to minimize a function f, the update should subtract the scaled gradient ($\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_w \ell\ell(\boldsymbol{w})$) – this gives the **gradient descent** algorithm.

Techniques

- **Batch gradient descent**: at each iteration of gradient descent, to use all the data points.
- **Mini-batching**: rotates through randomly sampled batches of k data points at a time.
- **Stochastic gradient descent (SGD)**: limit where the batch size $k = 1$.

Decision Trees (DTs)



Can express any function of the features.

DTs automatically conjoin features / attributes. Features can have different effects in different branches of the tree!

Choosing an Attribute

A good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative".

Entropy

$$H(\langle p_1, \dots, p_n \rangle) = E_p \log_2 \frac{1}{p_i} = \sum_{i=1}^{n} -p_i \log_2 p_i$$

More uniform = higher entropy. More peaked = lower entropy.

Information Gain

- Difference between entropy before and after.
- Use **expected entropy**, weighted by the number of examples.