General Formula

1 Agents

Have:
- Abilities
- Goals/Preferences
- Prior Knowledge
- Stimuli
- Past Experiences
- Actions

Belief State: Internal belief about the world
Knowledge: Information used to solve tasks
Representation: Data structure to encode knowledge
Knowledge Base: Representation of all knowledge possessed
Model: How KB relates to world

Dimensions of Complexity

1. Modularity
   - Flat: No modularity in computation
   - Modular: Each component is separateand siloed
   - Hierarchical: Modular components are broken into a hierarchical manner of subproblems
2. Planning Horizon:
   - Non-Planning: World doesn't change as a result (Ex: Protein Folding)
   - Finite: Reason ahead fixed number of steps
   - Indefinite: Reason ahead finite number (but undetermined) of steps
   - Infinite: Reason forever (focus on process)
3. Representation:
   - States: State describes how world exists
   - Features: An attribute of the world
   - Individuals and relations: How features relate to one another (Eg: child.failing() relates to child.grade)
4. Computational limits:
   - Perfect rationality: Agent always picks best action (Eg: Tic-Tac-Toe)
   - Bounded rationality: Agent picks best action given limited computation (Eg: Chess)
5. Learning:
   - Given knowledge (Eg Road laws)
   - Learned knowledge (Eg How car steers in rain)
6. Uncertainty:
   - Fully observable: Agent knows full state of world from observations (Eg: Chess)
   - Partially observable: Many states can lead to same representation (Eg: Battleship)
   - Deterministic: Action has predictable effect
   - Stochastic: Uncertainty exists over effect of action to state
7. Preference:
   - Achievement Goal: Goal to reach (binary)
   - Maintenance Goal: State to maintain
   - Complex Preferences: Complex tradeoffs between criteria and ordinality (can't please everyone)
8. Num Agents:
   - Single agent
   - Adversarial
   - Multiagent
9. Interactivity:
   - Offline: Compute its set of actions before agent has to act, so no computations required
   - Online: Computation is done between observing and acting

2 Search

b is branching factor (max num children of any node)
m is max depth of search-tree
d is depth of shallowest goal node

Search Problem:
- Set of states
- Initial state
- Goal function
- Successor function
- (optional) cost

Frontier: Ends of paths from start node that have been explored

Graph Search Algorithm

```
frontier is just start node
while frontier isn't empty
    select and remove path ⟨n_0, ..., n_k⟩ from frontier
    if goal(n_k) then
        return ⟨n_0, ..., n_k⟩
    for each neighbor n of n_k do
        add ⟨n_0, ..., n_k, n⟩
```

Uninformed Search

Depth-first-search

Use frontier as stack, always select last element
Cycle Check: Check if current node exists within path you are Checking
Space Complexity: $O(bm)$
Time Complexity: $O(b^m)$
Completeness: No
Optimal: No

Use:
- Restricted space
- Many solutions with long paths

Don't use:
- Infinite paths exist
- Optimal solutions are shallow
- Multiple paths to node

Breadth-first-search

Use frontier as queue, always select first element
Multiple-Path Pruning: Check if current node has been visited by any previous path by maintaining explored set.
Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness: Yes
Optimal: No (only finds shallowest goal node)

Use:
- Space isn't restricted
- Want shallowest arc

Don't use:
- All solutions are deep
- Problem is large and graph is dynamically generated

Iterative-Deepening

For every depth-limit, perform depth-first-search, this marries BFS and DFS by "doing BFS" but without space-concerns. However, we end up revisiting nodes.

Space Complexity: $O(bd)$
Time Complexity: $O(b^d)$, $b^d \sum_{n=1}^{d} n\left(\frac{1}{b}\right)^{n-1} = b^d\left(\frac{b}{1-b}\right)^2$ We visit level i $d-i$ times, and level i has b^i nodes, so that's the sum, then extending to infinity, and use geometric series.
Completeness: Yes
Optimal: No (only finds shallowest goal node)

Use:
- Space isn't restricted
- Want shallowest arc

Don't use:
- All solutions are deep
- Problem is large and graph is dynamically generated

Lowest-Cost-First-Search

Select a path on frontier with lowest cost. Frontier is priority queue ordered by path cost. *Technically uninformed/blind search because it's searching randomly*
Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness and optimality: Yes if branching factor is finite and cost of every edge is strictly positive.
Termination: Only terminate when the goal node is first on the frontier, not if it's in the frontier.

Dijkstra's

Similar to LCFS but keep track of lowest cost to reach each node, if we find lower cost path, update that value and resort the priority queue. Ex: Suppose we have path in frontier $\langle P, Q, R, \rangle$ and the found path to Q is 10 and overall cost is 12, then we find a new path to Q of cost 9, then we should recompute path to get cost 11.

Heuristic Search

$h(n)$ is estimate of cost of shortest path from n to a goal node. Should only use readily obtainable information and be **much** easier than solving the problem.

Greedy Best-First Search

Select path whose end is closest to a goal based on heuristic.
Frontier is a priority queue ordered by h.
Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness and optimality: Not guaranteed (could be stuck in a cycle or return sub-optimal path)

Heuristic Depth-First-Search

Do Depth-First-Search, but add paths to stack ordered according to h. Basically, do DFS, but sort the children by h to determine who to check. Same complexity and problems as DFS but used often.

A* search

Use both path cost and heuristic values. Frontier is sorted by $f(p) = \text{cost(p)} + h(p)$. Always selects node with lowest estimated distance.

Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness and optimality: Only with admissable heuristic, finite branching factor, and bounded arc-costs (there is a minimum positive arc-cost). A* always expands the fewest nodes for all optimal algorithms and use the same heuristic. No algorithm with same info can do better. This is because if an algorithm does not expand all nodes with $f(n) < \text{cost}(s, g)$ they might not find the optimal solution.

Admissable Heuristic

Never overestimates the shortest path from n to goal node.

Procedure for construction:
1. Define relaxed problem by simplifying or removing constraints
2. Solve relaxed problem without search
3. Cost of optimal solution to relaxed problem is admissable heuristic for original problem.

Dominating Heuristic

Given two heuristics, $h_2(n)$ dominates $h_1(n)$ if $\forall n h_2(n) \geq h_1(n)$ and $\exists n h_2(n) > h_1(n)$ We prefer dominating heuristics because it reduces the nodes we have to expand (they're bigger, so we don't care)

Monotone Restriction

A* guarantees finding optimal goal, but not necessarily shortest path. In order to do that, we would want our estimate path $f(p)$ to indeed allow us to remove longer paths, but what if one path has shorter cost, but heuristic sums make the shorter path have larger $f(p)$? We can avoid that, by inducing monotonic restriction. $h(n') - h(n) \leq \text{cost}(n', n)$. This guarantees heuristic estimate is always less than actual cost and if we ever find a shorter estimate, that estimate will actually be shorter, so we can prune it.

Further, monotonic restriction with multi-path pruning always finds shortest path to goal, not just optimal goal itself.

Note that admissability guarantees heuristic is never bigger than shortest path to goal, monotonicity ensures heuristic is never bigger than shortest path to any other node.

Summary

Strategy	Frontier Selection	Halt?	Space	Time
Depth-first	Last node added	No	Linear	Exp
Breadth-first	First node added	Yes	Exp	Exp
Heuristic Depth-first	Local min $h(n)$	No	Linear	Exp
Best-first	Global min $h(n)$	No	Exp	Exp
Lowest-cost-first	min cost(n)	Yes	Exp	Exp
A*	min $f(n)$	Yes	Exp	Exp

Adversarial Search (Minimax)

For one node look to maximize the heuristic, for the other node look to minimize it (to simulate the adversarial search)

- Alpha-beta pruning can ignore portions of search tree without losing optimality. Useful in application but doesn't change asymptotics
- Can stop early by evaluating non-leafs via heuristics (doesn't guarantee optimal play)

Higher-level strategies

Bidirectional Search: Search from backward and forward simultaneously taking $2b^{\frac{k}{2}}$ vs b^k and try to find where frontiers match
Island-driven Search: Find set of islands between s and g as mini problems. With m islands, you get $mb^{\frac{k}{m}}$ vs b^k but it's harder to guarantee optimality.

3 Constraints

Constraint Satisfaction Problems

- A set of variables
- Domain for each variable
- Two kinds of problems:
  - Satisfiability problems: Assignment satisfying hard constraints
  - Optimizatoin: Find assignment optimizing evaluation function (soft constraints)
- Solution is assignment to variables satisfying all constraints
- Solution is model of constraints

CSPs as graphs

Search spaces can be very large, path isn't important, only goal, and no set starting nodes make this bad idea
Complete Assignment: Nodes: Assignment of value to all variables
Neighbors: Change one variable value
Partial Assignment: Nodes: Assignment to first $k-1$ variables
Neighbors: Assignment to k^{th} variable

Constraints

- Can be **N-ary** (over sets of N variables) (Ex: A + B = C involves is 3-ary for 3 vars)

Generate and Test

Exhaust every possible assignment of vars and test validity

Backtracking

Order all variables and evaluate constraints in order as soon as they are fixed. (Ex: $A = 1 \wedge B = 1$ is inconsistent with $A \neq B$ so go to last assigned variable and change its value)

Consistency

Represent constraints as network to determine how all variables are related.
Domain Constraint: Unary constraint on values in domain written $\langle X, c(X) \rangle$ (Eg: $B, B \neq 3$)
Domain Consistent: A node is domain consistent if no domain value violates any domain constraint, and a network is domain consistent if all nodes are domain consistent.
Arc: Arc $\langle X, c(X, Y) \rangle$ is a constraint on X
Arc Consistent: Arc $\langle X, c(X, Y) \rangle$ is arc consistent if for every valid x there is a valid y such that constraint is satisfied.
Path Consistent: A set of variables is path consistent if all arcs and domains are consistent.

AC-3

Make Consistency network arc consistent
- To-Do Arcs Queue contains all inconsistent arcs
- Make all domains domain consistent
- Put all arcs in TDA
- Repeat until TDA is empty:
  - Select and remove an arc from TDA
  - Remove all values of domain of X that don't have value in domain of Y that satisfy constraint
  - If any were removed, add all arcs to TDA

Termination:
- If every domain is empty, no solution
- If every domain has a single value, solution
- If some domain has more than one value, split in two run AC-3 recursively on two halves
- Guaranteed to terminate
- Takes $O(cd^3)$ time, with n variables, c binary constraints, and max domain size is d because each arc $\langle X_k, X_i \rangle$ can be added to queue at most d times because we can delete at most d values from X_i. Checking consistency takes $O(d^2)$ time.

Variable Elimination

- Eliminate variables one-by-one passing constraints to neighbours.
- When single variable remains, if no values exist then network was inconsistent.
- Variables are eliminated according to elimination ordering.

Pseudocode:
- If only one variable, return intersection of unary constraints referencing it
- Select variable X
  - Join constraints affecting X, forming constraint R
  - Project R onto its variables other than X, calling this R2
  - Place new constraint between all variables that were connected to X
  - Remove X
  - Recursively solve simplified problem
  - Return R joined with recursive solution

Local Search

- Maintain assignment of value to each variable
- At each step, select neighbor of current assignment
- Stop when satisfying assignment found or return best assignment found
- Heuristic function to be minimized: Number of conflicts

- Goal is an assignment with zero conflicts

Greedy Descent

Select some variable (through some method) and then select the value that minimizes the number of conflicts. THe problem is that we could be stuck in a local minimum, without reaching the proper global minimum.

Stochastic Local Search

Do Greedy descent, but allow some steps to be random, and the potential to restart randomly, to minimize potential for being stuck in local minimum.

Problem: in high dimensions often consist of long, nearly flat "canyons" so it's hard to optimize using local search.

Simulated Annealing

Pick variable at random, if it improves, adopt it. If it doesn't improve, then accept it with a probability through the temperature parameter, which can get slowly reduced.

Tabu Lists

Variant of Greedy Satisfiability, where to prevent cycling and getting stuck in local optimum, we maintain a "tabu list" of the k last assignments, and don't allow assignment that has already existed.

Parallel Search

- Total assignment is called individual
- Maintain population of k individuals
- At each stage, update each individual in population
- Whenever individual is a solution, it can be reported
- Similar to k restarts, but uses k times minimum number of steps

Beam Search

- Like parallel search, with k individuals, but choose the k best out of all the neighbors. The value of k can limit space and induce parallelism

Stochastic Beam Search

- Like beam search, but probabilistically choose k individualls at next generation. Probability of selecting neighbor is proportional to heuristic: $e^{-\frac{h(n)}{T}}$. This maintains diversity among the individuals, because it's similar to simulated annealing.

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create offspring.
- For each generation, randomly choose pairs where fittest individuals are more likely selected
- For each pair, do cross-over (form two offspring as mutants of parents)
- Mutate some values
- Stop when solution is found

Comparing Algorithms

Since some algorithms are super fast some of the time and super slow other times, and others are mediocre all of the time, how do you compare? You use runtime distribution plots to see the proportion of runs that are solved within a specific runtime.

4 Inference and planning

Problem Solving

Procedural solving: Devise algorithm, program, execute
Declarative solving: Identify required knowledge, encode knowledge in representation, use logical consequences to solve.

Logic

Syntax: What is an acceptable sentence
Semantics: What do the sentences and symbols mean?
Proof procedure: How to construct valid proofs?
Proof: Sequence of sentences derivable using an inference recursively

Statements/Premises: $\{X\}$ is a set of statements or premises, made up of propositions.
Interpretation: Set of truth assignments to propositions in $\{X\}$
Model: Interpretation that makes statements true
Inconsistent statements: No model exists
Logical Consequence: If for every model of $\{X\}$, A is true, then A is a logical consequence of $\{X\}$

Argument Validity is satisfied if any of the identical statements are true:
- Conclusions are a logical consequence of premises
- Conclusions are true in every model of premises
- No situation in which the premises are all true but the conclusions are false.
- Arguments → conclusions is a **tautology** (always true)

Proof

Knowledge Base: Set of axioms
Derivation: KB ⊢ g can be found from KB using proof procedure
Theorem: If KB ⊢ g, then g is a theorem
Sound: Proof procedure is sound if KB ⊢ g then KB ⊨ g (anything that can be proven must be true (sound reasoning))
Complete: Proof procedure is sound if KB ⊨ g then KB ⊢ g (anything that is true can be proven (complete proof system))

Complete Knowledge: Assume a closed world where **negation** implies failure since we can't prove it, if it's open there are true things we don't know, so if we can't prove something, we can't decide if it's true or false.

Bottom-up Proof (aka forward chaining)

Start from facts and use rules to generate all possible derivable propositions

To prove: $F \leftarrow A \wedge E$ $A \leftarrow B \wedge C$ $A \leftarrow D \wedge C$ $E \leftarrow C$ D C

Steps of proof: $\{D, C\} \rightarrow \{D, C, E\} \rightarrow \{D, C, E, A\} \rightarrow \{D, C, E, A, F\}$ Therefore, if g is an atom, KB ⊢ g, if g ∈ C at the end of the procedure, where C is the consequence set.

Top-Down

Start from query and work backwards yes ← F yes ← $A \wedge E$ yes ← $D \wedge C \wedge E$ yes ← $D \wedge C \wedge C$ yes ← $D \wedge C$ yes ← D yes ←

Individuals and Relations

KB can contain **relations**: part_of(C, A) is true if C is a "part of" A
KB can contain **quantification**: part_of(C, A) holds $\forall C, A$ Proofs are the same with extra bits for handling relations & quantification.

Planning

Decide sequence of actions to solve goal based on abilities, goal, state of the world Assumptions:
- Single agent
- Deterministic
- No exogenous events
- Fully-observable state
- Time progresses discretely from one state to another
- Goals are predicates of states to achieve or maintain (no complex goals)

Action: Partial function from state to state
Partial Function: Some actions are not possible in some states, preconditions specify when action is valid, and effect determines next state

State Representations

Feature-based representation of actions: For each action, there is a precondition (proposition) that specifies when action is valid and a set of consequences for features after action.
State-based representation: For each possible assignment of features, define a state. Then for each action define the starting and ending state for the state-based graph.

Causal Rule: When feature gets a new value
Frame Rule: When feature keeps its value *Features are capitalized, but values aren't If X is a feature, X' is feature after an action*

Forward Planning: Search in state-space graph, where nodes are states, arcs are actions, and a plan is a path representing initial state to goal state.
Regression Planning: Search backwards from goal, nodes correspond to subgoals and arcs to actions. Nodes are propositions (formula made of assignment of values to features), arcs are actions that can achieve one of the goals. Neighbors of node N associated with arc specify what must be true immediately before A so that N is true immediately after. Start node is goal to be achieved. Goal(N) is true if N is a proposition true of initial state.

5 Learning

Learning: Ability to improve behaviour based on experience. Either improve range (more abilities), accuracy, or speed.

Components of learning problem:
- Task: Behaviour to improve (Ex: Classification)
- Data: Experiences used to improve performance
- Improvement metric

Common Learning Tasks:
- Supervised classification: Given pre-classified training examples, classify new instance
- Unsupervised learning: Find natural classes for examples
- Reinforcement learning: Determine what to do based on rewards and punishments
- Transfer Learning: Learn from expert
- Active Learning: Learner actively seeks to learn
- Inductive logic programming: Build richer models in terms of logic programs

Learning by feedback:
- Supervised learning: What to be learned is specified for each example
- Unsupervised learning: No classifications given, learner has to discover categories from data
- Reinforcement learning: Feedback occurs after a sequence of actions

Metrics: Measure success by how well agent performs for new examples, not training.
P agent: Consider this agent that claims negative training data are the only negative instances, and gives positive otherwise (100% training data, bad test)
N agent: Same as N agent, but considers only positive training data. They both have 100% on training data, but disagree on everything else.

Bias: Tendency to prefer one hypothesis over another. Necessary to make predictions on unseen data. You can't evaluate hypotheses by the training data, it depends on what works best in practice (unseen data).

Learning as search: Given representation and bias, learning is basically a search through all possible representations looking for representation that best fits the data, given the bias. However, systematic search is infeasible, so we typically use a **search space**, **evaluation function**, and a, **search method**

Interpolation: Inferring results in between training examples (Ex: Train on 1,2,4,5. Test on 3)
Extrapolation: Inferring results beyond training examples (Ex: Train on 1, 2, 4, 5. Test on 1000)

Supervised Learning

Given input features, target features, training examples and test examples we want to predict the values for the target features for the test examples.
Classification: When target features are discerete.
Regression: When target features are continuous.

Noise: Sometimes features are assigned wrong value, or inadequate for predicting classification, or missing features.
Overfitting: Distinction appears in data that doesn't exist in unseen examples (through random correlations)

Evaluating Predictions

$Y(e)$ is value of feature Y for example e $\hat{Y}(e)$ is predicted value of feature Y for example e from the agent.
Error: Prediction of how close $\hat{Y}(e)$ is to $Y(e)$

Types of Features

Real-Valued Features: Values are totally-ordered. Ex: Height, Grades, Shirt-size. (The finitenss of the domain of values is irrelevant)

Categorical Features: Domain is fixed finite set. (Total-ordering is irrelevant). Point estimates are either definitive predictions or probabilistic predictions for each category.

Measures of Error

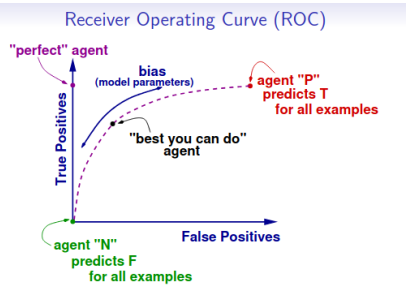
Absolute error: $\sum_{e \in E} \sum_{Y \in T} \left| Y(e) - \hat{Y}(e) \right|$
Sum of squares: $\sum_{e \in E} \sum_{Y \in T} \left(Y(e) - \hat{Y}(e) \right)^2$
Worst-case: $\max_{e \in E} \max_{Y \in T} \left| Y(e) - \hat{Y}(e) \right|$
- Cost-based error takes into account costs of various errors, so some are more costly than others.

Mean Log Loss: For probabilistic predictions, calculate the mean log loss ($\text{logloss}(p, a) = -\log(p[a])$) for all predictions (p = pred, a = acc). Mean log loss is a transformation of log-likelihood, so minimizing mean log loss finds highest likelihood.
Binary Log Loss: $\text{logloss}(p, a) = -a \log p - (1 - a) \log(1 - p)$ is a variant for boolean features.

Precision and Recall

Recall: Percentage of positive statements that are accurately predicted
Specificity: Percentage of negative statements that are accurately predicted
Precision: Percentage of predicted truths that are correct.

Receiver Operating Curve: A graph between True positives and False Positives



Basic Models: Decision Trees, Linear classifiers (Generalize to Neural Networks), Bayesian Classifiers

6 Decision Trees

- Representation: Decision tree
- Bias: Tendency towards a simple decision tree

Nodes: Input attributes/features
Branches: Labeled with input feature values (can have multiple feature values)
Leaves: Predictions for target features (point estimates) Search through the space of decision trees from simple to complex decision trees

Learning:
- Incrementally split training data
- Recursively solve sub-problems
- Try to get small decision tree with good classification (low training error) and generalization (low test error)

Learn Pseudocode:

```
X is input features, Y output features, E is training examples
Output is decision tree (Either Point estimate of Y or form ⟨X_i, T_1, ..., T_N⟩ where X_i is input feature and T_1, ..., T_N are decision trees.
procedure DecisionTreeLearner(X, Y, E):
    if stopping criteria is met then
        return pointEstimate(Y, E)
    select feature X_i ∈ X
    for each value x_i ∈ X_i do
        E_i = all examples in E where X_i = x_i
        T_i = DecisionTreeLearner({X_i}, Y, E_i)
    return ⟨X_i, T_1, ..., T_N⟩
```

Classify Pseudocode:

```
X: input features, Y: output features, e: test example, DT: Decision Tree
procedure ClassifyExample(e, X, Y, DT)
    S ← DT
    while S is internal node of form ⟨X_i, T_1, ..., T_N⟩
        j ← X_i(e)
        S ← T_j
    return S
```

Stopping Criteria:
- Stopping criteria is related to final return value
- Depends on what we will need to do
- Possible: Stop when no more features, or when performance is good enough

Feature Selection: We want to choose sequence of features resulting in smallest tree. Actually we should myopically split, as if only allowed one split, which feature would give best performance.

Heuristics:
- Most even split
- Maximum information gain
- GINI index

Information Theory

Bit is binary digit, n bits classify 2^n items, straight up. But if we use probabilities we can make the average bit usage lower. Eg: P(a) = 0.5, P(b) = 0.25, P(c) = P(d) = 0.125 If we encode a:00, b: 01, c: 10, d: 11, it uses on average 2 bits. If we encode a:0, b: 10, c: 110, d: 111, it uses on average 1.75 bits.

In general we need $-\log_2 P(x)$ bits to encode x, each symbol needs on average $-P(x)\log_2 P(x)$ bits and to transmit an entire sequence distributes according to $P(x)$ we need on average $\sum_x -P(x)\log_2 P(x)$ bits (entropy/information content)

Information Gain

Given a set E of N training examples, if number of examples with output feature $Y = y_i$ is N_i then $P(Y = y_i) = P(y_i) = \frac{N_i}{N}$ (Point estimate)

Total information content for set E is $I(E) = -\sum_{y_i \in Y} P(y_i)\log_2 P(y_i)$

So after splitting E into E_1 and E_2 with size N_1, N_2 based on input attribute X_i, the information content is $I(E_{\text{split}}) = \frac{N_1}{N}I(E_1) + \frac{N_2}{N}I(E_2)$ and we want the X_i that maximizes information gain $I(E) - I(E_{\text{split}})$

Note:

$$-\frac{N_1}{N}\left(\sum_{y\in Y} P(y \mid x=1)\log P(y \mid x=1)\right) - \frac{N_2}{N}\left(\sum_{y\in Y} P(y \mid x=2)\log P(y \mid x=2)\right)$$

$$-\sum_{y\in Y} P(x=1)P(y \mid x=1)\log P(y \mid x=1) - \sum_{y\in Y} P(x=2)P(y \mid x=2)\log P(y \mid x=2)$$

$$-\sum_{x\in X, y\in Y} P(x=x)P(y \mid x=x)\log P(y \mid x=x)$$

$$-\sum_{x\in X, y\in Y} P(x,y)\log\frac{P(x,y)}{P(x)}$$

Final Return Value: **Point estimate** of Y (output features) over all examples. This is just a prediction of target features. (Eg: Mean, Median, most likely classification, $P(Y = y_i) = \frac{N_i}{N}$)

Priority Queue

Basic version grows all branches for a node in decision tree, but it's more efficient to sort leaves via priority queue ranked by how much information can be gained with best feature at that leaf and always expand leaf at top of queue.

Overfitting: When decision tree is too good at classifying training it doesn't generalize well, occurs with insufficient data. Methods to fix:
- Regularization: Prefer small decision trees over big ones (complexity penalty)
- Pseudocounts: Add some data based on prior knowledge
- Cross validation: Partition into training and validation set. Use validation set as test set and optimize decision maker to perform well on validation set, not training set.
- Errors can be caused by:
  - Bias:
    - Represntation bias: Model is too simple
    - Search bias: Not enough search
  - Variance: Error due to lack of data
  - Noise: Error from data depending on features not modeled, or because process generating data is stochastic.
  - Bias-variance trade-off:
    - Complicated model, not enough data (low bias, high variance)
    - Simple model, lots of data (high bias, low variance)
  - Capacity: Ability to fit wide variety of functions (inverse of bias)

Linear Regression

The goal is to fit a linear function a set of real-valued training data. The linear function would be a combination of weights, which we can optimize over by taking the partial derivative of the loss for each weight, setting to 0, and solving the linear system (ex. a la RANSAC)

Squashed Linear Functions

Linear functions don't make sense for **binary classification** because we shouldn't extrapolate beyond the domain $[0, 1]$ which linear functions allow us to do. The solution is to use a function with domain $[-\infty, \infty]$ and range $[0, 1]$ to transform a linear function onto sensical binary domain.

Activation Function: Funciotn that transforms from real $(-\infty, \infty)$ to subset $[0, 1]$

To determine weights for a squashed linear function, you need to minimize the **log loss**.

Log Loss: $\text{LL}(E, w) = -\frac{1}{|E|} \times \sigma_{e\in E}\left(Y(e) \times \log\hat{Y}(e) + (1 - Y(e)) \times \log\left(1 - \hat{Y}(e)\right)\right)$ $\left(\hat{Y}(e)\right)$ is the predicted value after transformation. To optimize for this function, take partial derivatives for each weight and find their minimal.

Stochastic Gradient Descent

The goal is to find a local minimum of weights according to some error function, based on some initial value, learning rate, and partial derivative of weights to error.

The definition of gradient descent requires all training data to be interpreted. Stochastic gradient descent uses a random sample (batch) before updating its weights.

Batch: Set of b examples used in each update
Epoch: $\left\lceil\frac{|Es|}{b}\right\rceil$ batches, which is one pass through all data (on average)
By definition of P(y | x)
Linear Learner:

Inputs:
  Xs: set of input features
  Y: Target Feature
  Es: Set of training examples
  η: learning rate
  b: batch size
Output: Funciton to make prediction on examples
Algorithm:
  Initialize weights randomly
  Define squashed linear regression prediction function
  initialize derivative values to 0
  Select batch B of size b from Es
  for each example e in B
    Predict target and find error
    for each weight, add the partial derivative of error to derivative values
  for each weight take gradient descent step
Return: Prediction algorithm based on weights

Smaller batch sizes learn faster, but may not converge to local optimum, because you're more influenced by the randomness of your selection process.

Incremental Gradient Descent: Select batches of size 1 to update. Used for streaming data where each example is used once and then discarded.
Catastrophic forgetting: Discarding older data (and not using it again) means you fit later data better but forget earlier examples.

Linear Separability

Each input feature can be viewed as a dimension, and a hyperplane can separate an m-dimensional space into two spaces.
Linearly separable: A dataset is linearly separable if there exists a hyperplane where the classification is true on one side and false on other side.

Categorical Target Features

Indicator Variables: Technique to transform a categorical feature into a set of binary variables (If you have a categorical feature or not)

One issue with indicator variables is that, if we can only use one value, then the predicted probabilities should add to 1. One way to ensure this is to only learn for all but one variables and make the remaining variable's value the difference to 1, but this doesn't work because errors for other values accumulate but the non-trained value's errors don't.

One solution is to find a linear function for each value, exponentiate, and normalize. For instance, take the softmax function:

$$\text{softmax}((\alpha_1, ..., \alpha_k))_i = \left(\frac{\exp(\alpha_i)}{\sum_{j=1}^{k}\exp(\alpha_j)}\right)$$

This guarantees all values are positive and sum to 1, so are proper analogs for probability distribution.

Multinomial Logistic Regression: Given the categorical functions with k values, we can generalize our linear function into $u_{j(e)} = w_{0,j} + X_1(e) \times w_{1,j} + ... + X_{m(e)} \times w_{m,j}$ Note we have one linear function for each output value, so our weights are w_{ij} for input i, output j.

Categorical Log Loss: Take the log loss of the softmax function.

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} \quad & logloss(softmax((u_1(e), ..., u_k(e))), v_q) \\ &= \frac{\partial}{\partial w_{ij}} - \log\left(\frac{exp(u_q(e))}{\sum_j exp(u_j(e))}\right) \\ &= \frac{\partial}{\partial w_{ij}}(\log(\sum_j exp(u_j(e))) - u_q(e)) \\ &= ((\hat{Y}(e))_j - \mathbf{1}(j = q)) * X_i\end{aligned}$$

Here we only compare the predicted target value against the test target feature.

One-hot encoding: Given complete set of values where only one value is 1 and rest are 0

Overfitting

A complex model (with many degrees of freedom) has a better chance of fitting to the training data.

Bias: The error due to the algorithm finding an imperfect model. This is the deviation between the model found and the ground truth model.

Representation Bias: Representation does not contain a model close to the ground truth
Search Bias: Algorithm hasn't searched enough of the search space to find a better model.

Variance: Error from lack of data.

Bias-Variance Trade-Off: Complicated models can be accurate, but without sufficient data it can't estimate it properly (low bias, high variance). Simple models cannot be accurate, but can estimate model well given data (high bias, low variance)

Noise: Inherent error due to data depending on features not modeled or because data collection is inherently stochastic.

Overconfidence: Learner is more confident in prediction than data warrants.

Pseudocounts

Optimal prediction is usually the mean, however, this is not a good estimate for new cases, so we minimize the weight of the mean through pseudo-examples:

$\hat{v} = \frac{c \times a_0 + \sum_i v_i}{c+n}$, this value is a "weighted" mean, which minimizes the relative influence of the actual mean. a_0 is some defined set value.

Regularization

In addition to trying to fit a model to data, also include a term that rewards simplicity and penalizes complexity.

Ridge Regression: A linear regression function with an L2 regularizer
Lasso (Least Absolute Shrinkage and Selection Operator): Loss function + L1 regularizer

Feature Selection: The use of specific features for calculation. L1 regularization does this by making many weights zero.

Cross Validation

Instead of separate training and test sets, we go one step further. We have one training set, one validation set, and one test set (each completely separate).

Validation Set: This is an emulation of the test set, used to optimize the hyperparameters of the model.

Note, we want to train on as many examples as possible to get better models, so partitioning our data into three is not ideal. We use **k-fold cross validation**.

k-fold cross validation:
- Partition non-test examples randomly into k sets of approximately equal size, called folds.
- For each parameter, train k times for that parameter setting, using one of the folds as the validation set and remaining folds for training.
- Optimize parameter settings based on validation errors
- Return model with selected parameters trained on all data.

Composite Models

Although linear functions and decision trees can't be used to represent many functions, combining linear functions with non-linear inputs is a way to use simple linear models but make them more accurate.

Kernel Function: Function applied to input features to create new features. (Ex. x^2, x^3, xy)

Regression Tree: Decision Tree with constant function at each leaf (piecewise constant function)

Piecewise Linear Function: Decision Tree with linear function at leaves.

Ensemble Training: Using multiple learners, combine outputs via some function to create an ensemble prediction.

Base-level algorithms: Algorithms used as inputs for ensemble learners.

Random Forest: Using multiple decision trees make predictions through each tree and combine outputs. This works best if trees make diverse predictions: Each tree uses different subset of examples to train on (bagging) or subset of conditions for splitting are used.

Boosting

Boosting: Sequence of learners where each learns from errors of previous ones.

Boosting Algorithm:
- Base learners exist in sequence.
- Each learner is trained to fit examples previous learners didn't fit well.
- Final prediction uses a composite of predictions of each learner.
- Base learners don't have to be good, but have to be better than random.

Functional Gradient Boosting: Given hyperparameter K with K base learners, final prediction as function of inputs is $p_0 + d_1(X) + ... + d_{k(X)}$ where p_0 is initial prediction and d_i is difference from previous prediction (i.e. $p_{i(X)} = p_{i-1}(X) + d_{i(X)}$) Given p_{i-1} is fixed, the learners meant to optimize for $\hat{d}_i$ optimize for: $\sum_e \text{loss}\left(p_{i-1}(e) + \hat{d}_i(e), Y(e)\right) = \sum_e \text{loss}\left(\hat{d}_i(e), Y(e) - p_{i-1}(e)\right)$

Gradient-Boosted Trees

Gradient-Boosted Trees: Linear models where features are decision trees with binary splits, learned using boosting.

The prediction for example (x_e, y_e) is $\hat{y_e} = \sum_{k=1}^{K} f_{k(x_e)}$

Each f_k is a decision tree, each leaf has a corresponding output w_j (we call this the weight because of how it weighs to the larger sum). We also have a function q which corresponds to the if-then-else structure of the tree. For simplicity, we represent w as a single vector of weights, so instead of the tree hierarchy, we use a vector to store all the weights for quick compute, and q maps onto this vector (just so simplify tree traversal).

The loss function is: $\mathcal{L} = \left(\sum_e (\hat{y_e} - y_e)^2\right) + \sum_{k=1}^{K}\Omega(f_k)$

$\Omega(f) = \gamma \cdot |W| + \frac{1}{2}\lambda \cdot \sum_j w_j^2$, this is the regularization term to minimize the weights. $|W|$ minimizes the number of leaves. The lambda product minimizes the values of the parameters. γ and λ are non-negative learnable parameters.

Choosing Leaf Values

The model is learnt using boosting, so each tree is learned sequentially. Consider building the tth tree, where previous are fixed. Assume (for simplicity) that the tree structure (q) is fixed. Lets optimize for the weight of a single leaf w_j. Let $I_j = \{e \mid q(x_e) = j\}$ be the set of training examples that map to the jth leaf.

Since the tth tree is learnt, for regression, the loss for the tth tree is: $\mathcal{L}^{(t)} = \frac{1}{2}\lambda * \sum_j w_j^2 + \sum_e \left(y_e - \sum_{k=1}^{t} f_{k(x_e)}\right)^2 + \text{constant} = \frac{1}{2}\lambda * \sum_j w_j^2 + \sum_e \left(y_e - \sum_{k=1}^{t-1} f_{k(x_e)} - w_j\right)^2 + \text{constant}$ where constant is regularization values for previous trees and size of tree.

Therefore, the minimum value for w_j is $w_j = \frac{\sum_{e\in I_j}(y_e - \hat{y}_e^{(t-1)})}{|I_j| + \lambda}$

For classification, when you take the derivative, it's difficult to solve analytically, so instead an approximation is used in the opposite step of the gradient.

Choosing Splits

Proceed greedily. Start with single leaf, find best leaf to expand to minimize loss. For small datasets look through every split, for larger splits, take subsamples or pre-computed percentiles.

No-Free-Lunch Theorem

No matter the training set, for any two definitive predictors A and B, there are as many functions from the input feature sto target

features consistent with evidence that A is better than B on off-training set (examples not in training) as when B is better than A on off-training set.

Consider m-Boolean input features, then there are 2^m assignments of input features, and 2^{2^m} functions from assignments onto {0, 1}. If we assume uniform distribution over functions, we can use 2^m bits to represent a function (one bit for each assignment, basically a lookup), and memorize the training data. Then for a training set of size n, we'll set n of these bits, but the remaining bits are free to be assigned in any way, as in there is a wide class of functions that it could be.

7 Reasoning Under Uncertainty

Probability

Prior Probability: Belief of agent before it observes anything
Posterior Probability: Updated belief (after discovering information - observation)

Bayesian Probability: Probability as a measure of belief (Different agents can have different information, so different beliefs)

Epistemology: A value based on one's belief
Ontology: A veritable fact of the world

The belief in a proposition α is measured in between 0 and 1, where 0 means it's **believed** to be definitely false (no evidence will shift that) and 1 means it's believed to be definitely true. Note the bounds are arbitrary. A value in between 0 and 1 doesn't reflect the variability of the truth, but rather strength of agent's belief.

Worlds: Semantics are defined in terms of "possible worlds", each of which is one way the world could be. There is only one true world, but real agents aren't omniscient, and can't tell.

Random Variable: Function on worlds—given world, it returns a value. Set of values it returns is the domain of the variable.

Primitive Proposition: Assignment of a value to a variable, or inequality between variable and value, or between variables.

Proposition: A connection between primitive propositions using logical connectives, that is either true or false in a world.

Probability Measure: Function μ from sets of world, into nonnegative real numbers that satisfies two constraints: If Ω_1 and Ω_2 are disjoint sets of worlds, then $\mu(\Omega_1 \cup \Omega_2) = \mu(\Omega_1) + \mu(\Omega_2)$ and $\mu(\Omega) = 1$ where Ω is the set of all possible worlds.

Probability of Proposition: Probability of proposition α, written $P(\alpha)$ is measure of set of possible worlds where α is true, so $P(\alpha) = \mu(\{\omega : \alpha \text{ is true in } \omega\})$

Probability Distribution: If X is random variable, $P(X)$ is function from domain of X onto real numbers such that given $x \in \text{domain}(X)$, $P(X)$ is probability of proposition $X = x$

Joint Probability Distribution: If $X_1, ..., X_n$ are all of the random variables, then $P(X_1, ..., X_n)$ is the distribution over all worlds, and an assignment to the random variables corresponds to a world.

Infinite Worlds

There can be infinitely many worlds if domain of variable is infinite, infinitely many variables (ex. variable for location of robot for every second from now into the future)

The probability of $X = v$ can be zero for a variable with continuous domain, but between a range of values, $v_0 < X < v_i$ it can have real-values (think of probability function), therefore a **probability density function** is used: $P(a \leq X \leq) = \int_a^b p(X)dX$

Parametric Distribution: Density function is defined by formula with free parameters.

Nonparametric Distribution: Probability function where number of parameters is not fixed, such as in decision tree.

Discretization: Convert continuous variables into discrete values (like heights that are converted into separate regions and then capped)

Conditional probability

Evidence: Proposition e representing conjunction of all of agent's observations

Posterior Probability: $P(h \mid e)$, given evidence e, belief of h
Prior Probability: $P(h)$, without any evidence, what is initial assumption of h

Formal Definition of prior probability: $\mu_{e(S)} = \begin{cases} c \times \mu(S) \text{ if e is true in } \omega \text{ for all } \omega \in S \\ 0 \text{ if e is false in } \omega \text{ for all } \omega \in S \end{cases}$

Then, for μ_e to be probability distribution:

$$\begin{aligned} 1 &= \mu_{e(\Omega)} \\ &= \mu_{e(\{w: e \text{ true in } w\})} + \mu_{e(\{w: e \text{ false in } w\})} \\ &= c \times \mu(\{w : e \text{ true in } w\}) + 0 \\ &= c \times P(e) \end{aligned}$$

So $c = \frac{1}{P(e)}$

Formal definition of posterior probability:

$$\begin{aligned} P(h \mid e) &= \mu_{e(\{\omega: h \text{ true in } \omega\})} \\ &= \mu_e(\{\omega : h \wedge e \text{ true in } \omega\}) + \mu_{e(\{\omega: h \wedge \neg e \text{ true in } \omega\})} \\ &= \frac{1}{P(e)} \mu(\{\omega : h \wedge e \text{ true in } \omega\}) + 0 \\ &= \frac{P(h \wedge e)}{P(e)} \end{aligned}$$

Chain Rule:

$$\begin{aligned} P(a_1 \wedge ... \wedge a_n) &= P(a_n \mid a_1 \wedge ... \wedge a_{n-1}) \times P(a_1 \wedge ... \wedge a_{n-1}) \\ &= P(a_n \mid a_1 \wedge ... \wedge a_{n-1}) \times ... \times P(a_2 \mid a_1) \times P(a_1) \\ &= \prod_{i=1}^{n} P(a_i \mid a_{i-1} \wedge ... \wedge a_1) \end{aligned}$$

Bayes Rule

Given current belief in proposition h based on evidence k, given new evidence e we update the belief as follows:

$P(h \mid e \wedge k) = (P(e \mid h \wedge k)) \times P(h \mid k) \frac{)}{P(e \mid k)}$
(assuming $P(e \mid k)) \neq 0$

Simplifying by keeping k implicit, we get:

$P(h \mid e) = \frac{P(e \mid h) \times P(h)}{P(e)}$

Expected Value ($\xi_{P(X)} = \sum_{v \in \text{ domain}(X)} v \times P(X = v)$) if finite/ countable, integral if continuous.

Independence

Conditional independence: If $P(X \mid Y, Z) = P(X \mid Z)$, then X is conditionally independent of Y

Unconditional Independence: If $P(X, Y) = P(X)P(Y)$, so they are conditionally independent given no observations. Note this doesn't imply they are conditionally independent.

Context-specific independence: Variables X and Y are independent with respect to context $Z = v$ if $P(X \mid Y, Z = v) = P(X \mid Z = z)$ that is it is conditionally independent for one specific value of Z

Belief Networks

Markov Blanket: Set of locally affecting variables that directly affect X's value.

Belief Network: Directed Acyclic Graph representing conditional dependence among a set of random variables. Nodes are random variables. Edges are direct dependence. Conditional independence is determined by an ordering of the variables; each variable is independent of its predecessors in total ordering given a subset of the predecessors called the parents. Independence is indicated by missing edges.

8 Probabilistic Learning

Bayes Rule: $P(m \mid E) = \frac{P(E \mid m) \times P(m)}{P(E)}$

9 Neural Networks

10 Planning with Uncertainty

11 Reinforcement Learning