CS170: Introduction to AI

Nathan Romanelli

Contents

1 What is AI

The ultimate goal of AI is to create an algorithm which can solve all problems (Artificial general intelligence). Since this isn't feasible, we instead aim to solve problems and their relevant sub-problems with specific algorithms. There's a direct trade-off from generality and problem solving ability. Training intuition is hard.

If artificial intelligence is possible, super intelligence is also theoretically possible. Super intelligence is AGI which surpasses human intelligence by orders of magnitude.

2 Search

2.1 Intro

Simple recursive searching can solve problems with high degrees of freedom; by exploring possible actions or different states of a system you can find an optimal path to a solution. Brute force search of a problem space isn't always feasible since with more degrees of freedom the size of the problem space exponentially increases.

2.2 Problem Solving Using Search

To solve problems using search we need to choose a problem space were going to search and an algorithm to search the space. The two choices are entirely independent of each other.

2.2.1 Problem Space

1 **Definition**

A **problem space** consists of the representation of the state, the initial state, a description of the actions we can take to transform the state (operators), and a description of the desired state (implicit or explicit). A solution consists of a path from the initial state to the goal state. If the path doesn't matter the problem is a constraint satisfaction problem.

An excess of operators, or a lack of them, can both be detrimental since an operator might be essential to a solution or useless for one, expanding our space without necessity. An implicit goal state can have multiple forms that satisfy some parameters, or it can have an explicit form.

2.2.2 Cost of operators

For some problems, not all operators will have the same cost. If operations are weighted, more operators might lead to a optimal solution. We can choose what parameter to optimize (Sum of cost of operations, number of operations, etc). There my be many options to optimize based on the attributes of the space. If the cost of moving from state to state is $G(n)$, the cost of getting to a state from the initial state S to some state n can be written as

$$\sum_{i \in (S,n)} G(i)$$

the optimal solution is one which minimizes this sum.

2.2.3 Algorithm

Preforming a random or brute force search of a large problem space isn't feasible; the time to do so increases exponentially with the dimensions of the space. Instead, we choose a systematic search method.

2 **Definition**
The **branching factor** of a problem space is the average number of children for each node in the problem space.

The branching factor of a problem determines how fast the problem space grows as we descend the tree. A small branching factor means with n levels of a tree there will be less nodes than a large branching factor.

3 **Definition**
The **diameter** of a problem is the depth of the longest solution, or the worst case, assuming only optimal moves. It is also referred to as God's number, since the path to the worst case is perfectly optimal.

The diameter can not always be strictly proven; but we can put upper/lower bounds on the worst case solution, by breaking rules or by using logic, until the bounds converge at the diameter. The worst case can be described as the hardest problem presentable (or the state furthest from the solution by some distance metric) if there is a unique solution or if all solutions are equally far from the initial state.

Creating a lower bound is easy, since you can just choose an arbitrarily low number (or the shortest distance metric even if its not possible) but proving increasing lower bounds and decreasing lower bounds until the two converge is extremely difficult.

4 **Definition**

1. An algorithm is **complete** if it can find a solution, given one exists, in finite time.
2. An algorithm is considered **optimal** if always finds the best solution to a problem, given multiple solutions.
3. **Time complexity** of an algorithm defines how long it takes to find a solution.
4. **Space complexity** is the memory required by an algorithm while searching for a solution.

As it turns out, memory often becomes a greater constraint than time with searching algorithms.

2.3 Searching Algorithms

2.3.1 generic search algorithm

5 **Algorithm**

```
function general-search(problem,QUEUEING FUNCTION)
nodes = MAKE-QUEUE(MAKE-NODE(problem,INITIAL-STATE))
loop do
if EMPTY(nodes) then return "failure"
node = REMOVE-FRONT(nodes)
if problem.GOAL-TEST(node.STATE) succeeds then return node
nodes = queueing-function(NODES,expand(NODE, problem.OPERATORS))
end
```

2.3.2 Breadth First Search

6 **Algorithm**

Intuitively, BFS expands all nodes at depth i before expanding nodes at depth $i + 1$. BFS is complete, and optimal, with a time complexity of $O(b^d)$ and a space complexity of $O(b^d)$.

Every node in the fringe is kept in the queue, which is poor for space complexity. On a problem with a high branching factor the queue can get too long to handle.

2.3.3 Uniform Cost Search

7 **Algorithm**

The algorithm expands the cheapest node where the cost is the path cost $G(n)$. It is complete, optimal, with a time complexity of $O(b^d)$ and space complexity of $O(b^d)$.

Hill climbing is basically a greedy search for some metric of cost.

2.3.4 Depth First Search

8 **Algorithm**

Expand node at the deepest level. It is not complete, not optimal, runs in $O(b^m)$ with space $O(bm)$ where m is the maximum depth.

DFS without checking for repeated states will run forever. If we check for repeated states, we may have to expand the full size of the problem space. DFS does well with space complexity, but can not find feasible all solutions. Occasionally, we know the full depth of the tree, and using this we can define a new search method.

2.3.5 Depth Limited Search

9 **Algorithm**

Expand node at the deepest level up to depth L. It is complete if the goal state lies above L, but its not optimal. It runs in $O(b^l)$ in space $O(bL)$.

2.3.6 Iterative Deepening Search

We don't always know the depth, so we can try to create a clever algorithm. DLS is not optimal, so we need a new method. We like the space complexity of Depth First search, but we need it to be optimal.

10 **Proposition**

We can try depth limited search with an iterative increasing limit L. This method has optimality and completeness of Breadth First Search with the nice space complexity of Depth First search.

11 **Algorithm**

Preform a Depth Limited Search with iteratively increasing L.

Since the nodes in a layer increases exponentially, at any given layer L the previous layers are constant time relative to L. This changes based on the branching factor, but trying to memoize these previous levels makes space complexity poor, defeating the purpose of Depth Limited Search.

2.3.7 Bi-directional search

12 **Algorithm**

Start searching from both the initial state and the goal state, meeting in the middle. This method is complete and optimal, if BOTH algorithms used to search are complete and optimal, and runs in $O(b^{\frac{d}{2}})$ in $O(b^{\frac{d}{2}})$ space.

A difficulty of this algorithm is knowing when the two searches have actually met. This can be done by keeping track of where the searches are in the space, but this very quickly consumes memory. Its possible to do this with hashing.

2.4 Hardness of your search

The time and space complexity depend on the branching factor b and the depth $d[a]$, so knowing this lets us figure out how hard the problem is. Getting the worst or average case branching factor is not usually hard, whereas knowing the depth can be very difficult. We can also bound the depth using upper and lower bounds.

13 **Fact**

Solutions always lie between 0 and infinity.

2.5 Discrete or continuous problems

Problems can be defined using discrete or continuous methods- sometimes it can be impossible to define operators discretely; such as when driving with no marked intersections. To handle this, we must project a discrete map onto the real valued world. Without this kind of mapping, problems can be unsolvable since almost all algorithms today are discrete. Since solutions can be implicit or explicit, many times we don't know what a solution would look like (Such as with 8 queens) so we must define the solution implicitly by some criteria a prospective solution must meet.

2.6 Heuristic Search

14 **Algorithm**

Moth Algorithm

The moth flies in a straight line and has a midpoint between his starting and end point; he then flies in a orthogonal line that brings him closer to the female moth. He repeats until he arrives at the female moth.

This is only possible because the moth has access to information (Smell), if you take it away them moth preforms a uniform grid search. The form of searching where the moth has access to information as it searches is heuristic search.

15 **Definition**

Heuristic Search is a form of informed search, as opposed to a uniform uninformed search covered above.

A form of information we can use for heuristic searching is finding distance to the goal state- this allows us to discount moves which bring us further from the goal state instead of exploring them as options equally valuable as random states.

16 **Definition**

A heuristic is a function that, when applied to a state, returns a number that is an estimate of the merit of the state, with respect to the goal. We write a heuristic as $h(n)$.

A heuristic tells us approximately how far the sate is from the goal state. Heuristics might underestimate or overs estimate the merit of a state. Heuristics that only underestimate are desirable and are called admissible. The Manhattan distance is a form of heuristic.

17 **Example**

We can use a heuristic with our general search algorithm; for queueing, we simply sort by cheapest $h(n)$. The algorithm is known as greedy search or hill climbing search or best first search.

A problem with heuristic searching is local minimums; heuristic searches can be caught in local minimums and since every surrounding state has a worse score by some metric, it will be stuck.

18 **Claim**

We have seen uniform cost which is optimal and complete, but slow. We also know hill climbing is neither optimal not complete, but very fast. We can potentially combine the two algorithms to create a new algorithm which is fast, optimal, and complete.

19 **Algorithm**

The A* Algorithm uses $g(n)$ to get the cost to get to the node, $h(n)$ which is the distance of the node to the goal state, and minimizes $f(n) = g(n) + h(n)$

If a heuristic is admissible, then A* is optimal and complete.

20 **Note**

The $h(n)$ of the goal state must be zero

$h(n) = 0$ doesn't imply we have arrived at the goal state

If we hardcode a heuristic to always return zero, A* simply becomes uniform cost search.

In general, we want a heuristic to be "tight." This means we want it to be as close to the true distance as possible without going over.

A* has the worst case $O(b^d)$ space complexity, but an iterative deepening version is possible. IDA* has $O(bd)$ space complexity.

21 **Note**

There is no way to beat A*, only a way to come up with better heuristics.

While greedy may sometimes produce optimal results comparable to A*, it will not always produce the optimal solution since it is not complete.

2.6.1 Island Driven Search

22 **Definition**

Island driven search lets you break the larger problem into smaller sub-problems which can be solved individually. This lets you reduce the space of a problem space when theres no feasible heuristics to drive A*.

An example of Island driven search is solving rubik's cubes- you can solve one side at a time and join your solutions to ultimately solve the cube and drastically reduce the problem space size. Generally, exploring an island will lead to an optimal solution if and only if that island is part of the optimal solution.

2.6.2 Search to failure

A solution to a certain problem doesn't always exist; solutions across parities of problems is impossible. For example, different parities of Rubik's cubes can never have identical states. In principle, we would exhaust our search and explore every nodes in a tree before deciding no solution exists, but this would take longer than the universe has existed. To make an unsolvable problem solvable, you can add 'bridges' between parities, such as switching cubes on a rubik's cube.

23 **Note**

Failure in search could be useful; suppose we prove something is true for n by searching a space until we find a solution, and we preform the same search for $n + 1$ until the search fails. This has proven something is only true for n.

Not knowing all operators can mean not finding a potential solution, or leading to failure.

24 **Definition**

Pre-computing a tree could mean instead of computing the entire tree we could preform a query in constant time using hashing.

Pre-computation is useful if some sections of a tree are being computed many times.

2.7 Adversarial Search

There's games along two dimensions; Deterministic or Stochastic, and Perfect Information or Imperfect information. Deterministic means no amount of probability is involed, whereas in Stochastic there is an element of luck. In perfect information games we know all information at all times, whereas in imperfect information games some information is hidden. Games can be one player or two player, and can be cooperative or competitive. A game has a form of intelligence, often adversarial, whereas a puzzle simply has a starting and goal state (Like a Rubik's cube).

25 **Definition**
In two player games we have an initial state, a set of operators, a terminal test (to tell when the game is over), and a utility function (evaluation function) to determine who is winning.

The utility function is like a heuristic function and can be used in searches.

26 **Lemma**
We make a fundamental assumption that both players are rational and always make the optimal decision, given any information available. Because of this assumption, the payoff results are a lower bound since one player may play sub-optimally.

27 **Algorithm**
Minmax algorithm is optimal, but has a time complexity of $O(b^m)$ where $b4$ is he effective branching factor and m is the depth of the terminal states. The space complexity is only liner in b and m since we can do DFS.

A solution to this is depth-limited minmax search where we evaluate as deep as possible given some constraint, evaluate the fringe nodes using our utility function, upward propagate the values, and make our decision.

28 **Example**
Consider a game of chess where max is "White". Assign values to each piece coresponding to the utility of the pieces. Let w = sum of value of white pieces, and b = sum of black pieces on the board. The evaluation function is $e(n) = \frac{w-b}{w+b}$. If this is positive, it means white has an advantage, and if its negative black has an advantage.

This isn't a great heuristic, since it only accounts for material advantage and not position. A side could be one move from checkmate and have a poor utility score.

29 **Example**
Let X_i be the number of squares the ith piece attacks, $e(n) = \text{piece}_1\text{value} \cdot X_1 + \text{piece}_2\text{value} \cdot X_2 + \dots$

Modern evaluations are done using machine learning and old games to find winning positions.

30 **Fact**
We can not beat Minmax for exact game playing. Instead, our time is better spent finding better utility functions (similar to A* and heuristics).

While this is true, we can make Minmax more efficient using Alpha-Beta Pruning. Using depth as a limit to our search is an issue because of the "horizon effect" where we may be one move away from making a brilliant move.

31 **Proposition**

What if we could prune branches from Minmax so that we don't explore blatantly bad moves to the same depth as moves which may be heuristically better.

We can do this by considering a "best so far" branch while exploring the minmax tree. Keeping this in memory, as soon as we encounter a branch which is less optimal than our "best so far" branch, we can prune that branch from the tree and save computational power and time by not exploring that branch.

32 **Algorithm**

Alpha-Beta, which preforms Minmax with our pruning method, we are guaranteed to find the same value for the root node as Minmax. In the worst case we preform a minmax search in $O(b^d)$ time, where each node has b children and $d-$ply search is used, but in the best case it takes $O(2 \cdot b^{\frac{d}{2}})$ time.

This means using Alpha-Beta we can essentially search twice as deep as minmax given the same tree in the best case.

33 **Note**

In the chess program Deep Blue, the branching factor was reduced from about 35-40 to 6 using pruning.

Modern chess engines can be constarined by difficulty using the depth of the search; a beginner chess player can think about 3-4 moves ahead, versus a grandmaster can think about 9 moves ahead. Similarly, we can constrain the difficulty of the algorithms by limiting the depth of our search. Theoretically, if the tree could be completely searched the algorithm could solve the game.

2.7.1 Stochastic Games

To account for games where probability is a factor, we can include chance nodes in our tree. The operators must include the cross product of every outcome of chance and everything a human can do.

34 **Definition**

The expected value of minimax can be written as

$$\sum_{s \in \text{Successors}(n)} P(s) \cdot \text{Expectminimax}(s)$$

Where $P(s)$ is the probability of a random event occurring

This makes the time complexity of minimax $O(b^m n^m)$ where n is the number of distinct events. $\alpha - \beta$ pruning can be extended, but the time complexity is still enormous. However, humans are terrible at probability, so algorithms can still beat humans at even shallow searches.

2.7.2 Monte Carlo Tree Search

The search space of Go is massively larger than the space for chess. The game-tree complexity of go is about $2^7 87$. Chess, also, has a higher variance in the space; nodes in the same level will have very different values of evaluation, whereas in Go the nodes in a level will have very little variance. Chess is a subtractive game, meaning we start with all the pieces and remove them, whereas Go is an additive game, which contributes to its massive branching factor. All the utility functions we have for Go tend to only be useful deeper in the tree. Because of this, by the time the evaluation algorithms become useful in Go, the human is already in a winning position.

35 **Proposition**

Monte Carlo tree search is an algorithm for stochastic games/systems; its not exact like minimax or alpha beta, but its extremely fast and can approximate solutions.

If we consider evaluation functions the average of many terminal tests, Monte Carlo only needs to evaluate individual terminal tests. By randomly sampling the distribution, we can approximate the evaluation of a node without explicitly evaluating every outcome.

36 **Algorithm**

Until we run out of time, Monte Carlo tree search will repeat the following process:
1) Expand the current node
2) Play n random games from each leaf
3) Pass the statistics of each game up the tree
4) Expand the most promising node

37 **Fact**

Matrix multiplication was an untouched problem for about 50 years until the team behind alphago created alphatensor. The idea was to make a game out of matrix multiplication and create an algorithm which could find an efficient algorithm to find solutions.

2.7.3 Solved Games

38 **Definition**

A solved game is one in which you can predict the outcome from any position given rational play.

The most obvious way to solve a game is to run minmax until the tree is completely explored. This is easy to do for a game like tic-tac-toe, which is a solved game. For games with larger problem spaces, solving becomes extremely difficult or impossible.

39 **Fact**

For some games, you can prove that one player will always win given optimal play.

I can’t play the L game cause I only take dubs :pensive:
While chess hasn’t’ been solved, the end games have, up to 7 or fewer pieces remaining.

2.8 Optimizing Search

Optimizing search is different from path search; the search is used for problems where exhaustive and heuristic search are NP hard. The path is not important, so we don’t need a tree and the constraint is CPU time more than memory. Every state in this search is a "solution," but we want a globally, or at least locally, optimal solution. The search spaces are continuous in this search, and the task is usually to minimze or maximize a function.

40 **Example**

A salesman spends his time visiting n cities. In one tour he visits each city just once, and finishes up where he started. In what order should he visit them to minimize the distance traveled? There are $(n-1)!/2$ possible tours.

To preform a optimization search we need states. Assume we can represent a state, quickly evaluate the quality of a state, define operators to change from one state to another. Operators must be expressive; they should allow us to reach any part of the search space.

41 **Proposition**

Lets consider two algorithms; Hill climbing search (Fast but not optimal) and Random search (optimal, but extremely slow). If we could combine the best aspects of these algorithms, we could get a best of both worlds algorithm like A*.

Random search is theoretically optimal, since it would explore every point in a problem space, but it could take practically endless time to exhaust a sufficiently large search space. Hill climbing search is extremely fast, but it doesn't find every optimum or a global optimum every time. By combining the two, we can preform hill climbing searches on random points. Preforming a random hill climbing search is extremely cheap, so we can preform enough that our probability of not finding the global optimum falls below a threshold were satisfied with. Step size is also an issue; if we pick a small step size finding an optimum can take far too long, and if we pick a large one we might bounce ourselves out of a local optimum.

42 **Algorithm**

Simulated-Annealing is the combination of hill climbing and random search. The algorithm's sub searches are greedy and extremely fast, and repeated until we find a solution were satisfied with. Both algorithms are a special case of a probabilistic operator choice. Let $x \in [0,1]$ where x is chosen randomly. If $x > t$, choose best operator, else chose a random operator. If $t = 0$, this is pure greedy search. If $t = 1$, it is pure random search. If we set $t \in (0,1)$, it is partly random and partly greedy. t stands for temperature.

The idea is we start with a high value of t, then slowly lower the temperature to zero. In other words, we start by searching randomly and slowly make it greedy. When the algorithm is "young" it searches very randomly, and as it "matures" it begins to become more conservative and moves in the direction of improvement.

2.9 Genetic Algorithms

Three conditions are necessary for natural selection:

1. Variation- some of the members of the same species differ
2. Heritability- Some variability is inherited
3. Finite Resources- not every individual will live to reproduce

Given the above conditions the idea of natural selection is this:

43 **Definition**

Some of the characteristics that are variable will be advantageous to survival. Thus, the individuals with the desirable traits are more likely to reproduce and have offspring with similar traits, therefore species evolve over time.

The idea above powers our genetic algorithm.

44 **Algorithm**

1. Initialize a population of n states
2. **While** time allows
3. Measure the quality of the sates using some fitness function
4. "Kill off" some of the states (typically the worst)
5. Allow the surviving states to reproduce (sexually, asexually, else)
6. **End While**
7. Report the best state as the answer

We need to figure out how to represent states and evaluate them. Most commonly we can represent states as trees. The fitness function is fairly trivial; usually we can use a distance metric or heuristic we are trying to optimize.

45 **Proposition**
Sex between genetic algorithms can be useful for escaping local optima; it allows us to make jumps across large spaces to other optima.

The idea that we keep the "best" set of individuals is flawed, since this essentially turns our search into a greedy search! We are homogenizing our pool of genes which means we will be stuck in a local optima. To solve this, we can instead keep some "unfit" genes in the hope they may lead us out of local optima.

46 **Note**
Some discussion regarding genetic algorithm

1. Genetic algorithms require many parameters (population size, fraction of population generated by crossover, mutation rate, number of sexes, etc...) and with each parameter the possible permutations of our algorithms increases exponentially. This means its very difficult to tune all these parameters and ensure our algorithms functions how we want.
2. Because we are keeping some "bad" algorithms, instead of a greedy search we are basically preforming a hill climbing search with some "Randomness" which lets us escape local optima.
3. Genetic algorithms are very easily parralellized

2.10 Why are human's so smart?

There is really no answer to this question, but the most popular theory is runaway evolution.

47 **Proposition**
Consider sexual selection in nature; In a scenario where there is a constant preference in sexual selectors, the distribution in a sufficiently large population converges at that preference since any deviation is not selected. In contrast, if there is a preference for a trend, such as "I like mates with longer tails," then the distribution endlessly skews towards the preference.

The above scenario is an example of runaway evolution.

48 **Definition**

In runaway evolution, a competition within a species for sexual selection or survival is caught in a sort of feedback loop, resulting in extreme features in animals who win these competitions. Peacocks are an example of this, as their features they use for sexual competition are extremely exaggerated.

We theorize this applies to humans as well; humans prefer other humans who are smart, emotionally, socially, logically, or otherwise.

3 Machine Learning

3.1 Introduction

Instead of trying to code a very complex program to solve a problem, we can code a relatively simple one that can learn to solve the problem.

3.2 Supervised Learning

3.2.1 Classification

In classification we hope to sort data points into given classes based on their features.

3.2.1.1 Linear Classifier A simple linear classifier hopes to "split" the data, and works best on data which is linearly separable. To minimize the error of a linear classifier, we take the squared distance of each misclassified point from the classifier and add it to a sum, divided by the total number of points. This is mean squared error.

For many real problems there is no fixed data; instead we have a stream of data. If we use sum of squares for linear classification, instead of calculating a new classifier all over again for new data, instead we can try to adjust the classifier in constant time on only the new data point(s).

49 **Note**

A benefit of linear classification is interpretability; being able to interpret the results of our classifications means we can gain insight into the problem.

3.2.1.2 Non Linear Classifiers Decision boundaries used to classify do not need to be linear; they can have more degrees of freedom which allows us to better fit data. Allowing classifiers to better fit the data makes it sensitive to outliers; a linear classifier is the least sensitive to outliers, so similarly one which has arbitrarily high degrees of freedom will be the most sensitive to outliers.

3.2.1.3 Nearest Neighbor Classification Nearest neighbor classification classifies new data points using the nearest data point. This can be extended to k nearest neighbor classification for odd $k \geq 0$. This lets us deal with outlier sensitivity, since we will ignore single points which may disrupt our algorithm.

3.2.1.4 Picking Hyper-parameters How do we know what value of k to pick? The best solution is k-fold cross validation for varying values of k until we find one which minimizes loss.

3.2.2 What is supervised Learning?

50 **Definition**

Training algorithms using data with examples and their correct classification is called supervised learning.

In supervised learning we process data points use the correct classification to adjust our algorithm until we minimize loss on the training data.

3.2.2.1 Irrelevant Features Unfortunately, k-nearest neighbors is very sensitive to irrelevant features. The more features we include, the higher the dimensionality of our space, which makes nearest neighbor classification very different in different spaces. If we introduce a new dimension, we could completely change the k nearest neighbors of any given point, meaning that feature could ruin our classification.

51 **Proposition**

To deal with k-nearest neighbor's sensitivity to irrelevant features, we can use search subsets of features to find which set minimizes loss.

The problem with searching is the exponential time complexity; for n features there are $2^n - 2$ subsets of features excluding the empty set and the entire set. Some features are only useful together; for example, weight and height for determining health. Separating them may make them individually useless which makes searching much harder.

Another issue with k-nearest neighbor is the sensitivity to measurements or units. There isn't really a way to know how to scale height in meters and weight in kilograms so that they have the same impact on classification. 200 kilograms and 200 meters mean very different things.

52 **Definition**

$Z-$ Normalization is a form of normalizing data to reduce sensitivity to scaling. For any data point x_i we normalize by

$$x_i := \frac{X - \mu}{\sigma}$$

53 **Definition**

The default rate is the size of the most common class over the size of the data set. It is the best accuracy if we use no features.

3.2.3 Feature Selection/generation

Given features of a data set, we may want to exclude useless features or generate new ones (Such as functions of multiple features). For example, instead of just using height, weight, and eye color to determine gender, we may instead want to use height, weight, and $\frac{\text{height}}{\text{weight}^2}$.

Selecting which features to use is hard, since given n features there are 2^n subsets of features. Because of this, we usually use greedy search.

3.2.4 Decision Tree Classifier

In general, if we have n Boolean features, the number of trees we can create is 2^{2^n}; this is good news since there may be many tree's which can accurately classify, but its bad news if we want to explore this space. In a decision tree, an internal node represents a test on an attribute, the branches are the outcomes of these tests, and leaf nodes represent classifications. We can construct trees beginning at the root and partition examples based on selected attributes, and prune the tree based on how well it preforms.

When building a tree, we want to pick features or ask questions which can give us good reductions of entropy in a data set. These features/questions will give us good splits of the data.

54 **Definition**

Entropy is a measure of disorderliness in a data set

Lets say we have a data set we want to classify; if we label all the data with the most common label, and then find all the outliers that are not properly labeled, the entropy of the data set is the amount of bits necessary to describe the 'purity' of the box.

55 **Definition**

Information Gain as a splitting criteria

Let P and N be some classes in the set S, with p examples of P and n examples of N. The amount of information necessary to determine if an example in S belongs to P or N can be quantified as

$$E(S) = -\frac{p}{p+n}\log_2(\frac{p}{p+n}) - \frac{n}{p+n}\log_2(\frac{n}{p+n})$$

This is useful to know since we want to select features/ask questions which give us the highest information gain. Information gain is the expected reduction in entropy. Intuitively, this makes sense since questions which apply to only one of many data points don't help us much, but ones which split the data very well are extremely useful.

56 **Definition**

Assume that using feature A, the data will be split into some subsets. The encoding information that would be gained by branching on A can be described as

$$\text{Gain}(A) = E(\text{current set}) - \sum E(\text{all child sets})$$

It's worth noting that

$$0 \leq \text{Gain}(A) \leq 1$$

Converting decision trees to rules, or **if else** statement is very easy, since each branch from a node is essentially a rule/if statement.

57 **Note**

As the data sets we attempt to split decreases in size, the number of questions which decrease entropy grows significantly. With only 2 data points, we can ask really any question specific to one of the data points and it will reduce entropy.

There's a few ways to deal with overfitting in trees; we can admit ignorance, we can use probability, or we can 'borrow' data from other points in the tree to test our question. One of the best ways is to get more good data.

Another good way of handling overfitting is simply pruning the tree through cross validation.

58 **Note**

Pros of DT

-Easy to understand

-Easy to create rules from them

Cons of DT

-Sensitive to overfitting (more-so than other algorithms)

-Classify by rectangular partitioning which doesn't handle correlated features well

-Can grow to be quite large.

3.2.5 Naive Bayes Classifier

Consider data points represented by a normal or Gaussian distribution; we can use the probability that a data point is in a class to help us classify.

59 **Proposition**

Using the data we have, we can calculate the probability a data point belongs to a certain class and use that to classify future data points.

$$P(\text{Class} \mid X_i) = \text{Prediction}$$

Intuitively, Naive Bayes means we find the probability an instance belongs to each class and pick the most probable class.

60 **Definition**

For a given class c_j and data d we have

$$p(c_j \mid d) = \frac{p(d \mid c_j)p(c_j)}{p(d)}$$

where

$p(c_j \mid d)$ is the probability of the class given the data

$p(d \mid c_j)$ is the probability of generating instance d given c_j

$p(c_j)$ probability of occurrence of class c_j

$p(d)$ probability of instance occurring

Naive Bayes is predicated on the assumption that the data we're processing is independent; obviously this isn't true, since height and weight of a person, for example, are clearly correlated, but the assumption lets us do useful things.

61 **Definition**

The following equation, which is essential to Naive Bayes, only works when the data is independent; otherwise the equation is not true

$$P(d \mid c_j) = p(d_1 \mid c_j) \cdot p(d_2 \mid c_j) \cdot \ldots \cdot p(d_i \mid c_j)$$

Despite its shortcomings, Naive Bayes is very space efficient and very fast. It is also insensitive to irrelevant features; this is because, in an ideal distribution, uncorrelated features will have equal probability across classes and not change our prediction. This isn't true for very small data sets, unfortunately.

We can handle the problem of assuming independence by considering the relationship between features; if features have correlations (Not necessarily causal) we can handle it by getting more data for the related features. Finding feature correlation can be done using search; we consider all arcs between features, evaluate them, and keep the ones that improve accuracy. The evaluation function is k-fold cross validated.

62 **Note**

Pro's/cons of Naive Bayes

Pros

- Fast to train, fast to classify
- Not sensitive to irrelevant features
- Handles real and discrete data
- Handles streaming data really well

Cons

-Assumes independence of features

3.2.6 Monte Carlo Search

Suppose we flip a coin 12 times in a row, what are the possible outcomes?

1. 12 heads, 0 tails
2. 11 heads, 1 tails ...
3. 5 heads and 7 tails
4. 0 heads, 12 tails

A way to estimate the probability of case 5 is by sampling the distribution. Say we flip a coin 12 times in a row thousands of times- we can count the number of times case 5 occurs and approximate the probability. If we preformed this experiment, we would find the results can be approximated by a normal distribution. This method of sampling can be used to estimate any number, even the value of pi such as in the image below.

Suppose we want to know some value with no analytical solution, or one which would be unfeasible to find analytically (Because the time or memory required is impossible). In such a case, we can use Monte Carlo simulation to approximate the distribution! We can build a model of he system we want to examine, "run" the model many times with random starting conditions from the correct distributions, and report the aggregate results.

3.2.6.1 Improving Monte Carlo If we receive a probability of a certain event occurring, we could code in uncertainty with a random number in a range of uncertainty. For example, if we find there is a 50$ chance of event A occurring, but we are only confident enough within a 2% margin of error. Instead of using 58% for our model, we can instead use a random number in the range (58-2,58+2).

Another improvement is considering dependence; probabilities of events may have some dependence, and accounting for this dependence can greatly improve the effectiveness of our simulation.

3.2.7 Neural Networks

3.2.7.1 Connecionist Neural Network

63 **Definition**

Connectionism refers to a computer modeling approach that is loosely based on the architecture of the brain

Modern neural networks consist of layers of connected neurons where a neuron is some function of inputs.

64 **Fact**

A neural network consisting of a single layer can represent any linear classifier. In fact, with multiple layers, a neural network can represent any possible function.

Neural networks consist of an input layer, hidden layers, and an output layer. The hidden layers can act as individual feature recognizers- they will activate when a specific aspect of the input is recognized (such as a horizontal line in a number, or the fur color of a puppy.)

Neural networks train slowly, need lots of data, and are hard to interpret. Network topology layouts are adhoc, and the code can be hard to debug. Additionally, it may converge to a local minimum of error, and we don't know how to model higher level cognitive mechanisms.

3.3 Unsupervised Learning

3.3.1 Clustering

Hierarchical clustering lets us efficiently use past examples to "cluster" new examples. For instance, we have a mental clustering of the world; at the root of the tree is the earth, then its children for instance are the continents/regions, and their children are respective countries or country groups. Hierarchical clusters can be represented as trees and efficiently traversed to classify new examples.

Clustering lets us reason about the world and make generalizations from incomplete information so that we can make predictions. Now that clustering is well motivated, lets define it.

65 **Definition**

Clustering is the organization of data into classes such that there is high intra-class similarity and low inter-class similarity.

Unlike clustering, which is a form of supervised learning, we find the classes/labels from the data which is unlabeled. The lack of training data makes clustering a form of unsupervised learning. Intuitively, we are searching for naturally occurring groups in the data.

We can judge the "goodness" of a clustering by measuring distance between data points- distance within a class should be minimized, and distance between classes should be maximized.

66 **Definition**

Distance is a metric used to judge the similarity of two data points.
There are some desirable properties of distance metrics:

1. Symmetry: $D(A, B) = D(B, A)$
2. Constancy of self-similarity: $D(A, A) = 0$
3. Positivity: $D(A, B) = 0$ if and only if $A = B$
4. Triangular inequality: $D(A, B) \leq D(A, C) + D(B, C)$

There are two primary forms of clustering: Hierarchical and Partitional. In the former, classes are children of each other in a tree like fashion.

3.3.2 Hierarchical Clustering

Dendograms are useful for representing hierarchical clusterings. In dendrograms, the height of the cluster is indicative of how well defined the cluster is, and can be used to find outliers/the best number of clusters.

67 **Definition**

Dendograms are...

For n leaf nodes, the number of dendrograms we can create is

$$\frac{(2n-3)!}{2^{n-2}(n-2)!}$$

We can't possibly search this entire space, but we can use heuristic search. There's two primary methods of building a tree- bottom up (Agglomerative), where we start each item in its own cluster and merge the best pair into a new cluster, and repeat until all clusters are merged, or top down (Divisive), in which we start with a single cluster and split based on the best way to divide the data. How to determine the distance between a data point and a cluster isn't always clear, but there are three primary methods.

68 **Definition**

Single Linkage: The distance between two clusters is determined by the closest objects in those clusters. Also known as nearest neighbor.

Complete Linkage: Single linkage, but we use the furthest data point, or nearest neighbor.

Group Average Linkage: The average distance between all pairs in the clusters.

Wards Linkage: We minimize the variance of the merged clusters.

3.3.3 Partitional Clustering

The most common method of clustering next to hierarchical. This method of clustering lets us group data points into individual groups. For example, for data sets with small amounts of clusters, dendrograms could reach unreasonable sizes, whereas partitional clustering can cluster the data into a small, usable, number of clusters. Ideally, the algorithm will tell us the most "natural grouping" of that data set.

3.3.3.1 k-means

Suppose we wanted to find a point in the center of each cluster such that we minimize the mean squared error.

69 **Definition**

The objective function for k-means is se_K such that

$$se_{K_i} = \sum_{j=1}^{m} ||t_{ij} - C_k||^2$$

$$se_K = \sum_{j=1}^{m} se_{K_i}$$

70 **Algorithm**

***K*-means**

1. Decide a value for k
2. Initialize the k clusters randomly
3. Decide the class memberships of the N object by assigning them to the nearest cluster
4. Re-approximate the position of the clusters based on the previous assignment
5. Repeat steps 3 and 4 until there is no changes in the membership of the N objects

If K is is too large then K-means can fail to find a meaningful grouping of the data. Additionally, bad starts can cause bad groupings. K-means is relatively efficient, but often rests at a local optimum. The algorithm only applies when means are well defined for the data, k needs to be defined as a hyper parameter, and the algorithm is sensitive to noisy data. The algorithm is also unsuited for clusters with non-convex shapes.

4 Knowledge Representation

4.1 Knowledge Bases

The human mind is originally a blank slate, it can ingest facts about the world. Some are taught explicitly, some are learned. The mind can create new facts by combining old facts. This is part of intelligence, and something we want to do with algorithms. Suppose we replaced a human mind with an algorithm, we want to build a system such that we can tell it facts and the system can generate new facts (Typically in response to a question).

We want to simulate fact create in our algorithms. For example, suppose we're given an email with the name "Asami Keogh." We know that Asami is a Japanese name, and Keogh is an Irish name. From this we can determine that Asami is most likely female, married to someone Irish. This is fact generation. Another example, At a high level, Isaac Newton was able to read Euclid's elements and derived new facts such as the laws of motion, law of universal gravitation, and calculus.

71 **Definition**

Knowledge representation is a subarea of Artificial Intelligence concerned with understanding, designing, and implementing ways of representing information (knowledge) in computers so that programs (agents) or humans can use this information

72 **Definition**

A knowledge base is a system which we can tell a set of facts/rules, ask questions, and receive answers to those questions.

73 **Note**

"A program has common sense if it automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows - John Mcarthy

Knowledge bases must be able to be told information, they must also be capable of make deductions based on that information.

It is important to note, the query's could be made by robots, not always by humans. For instance, a driver-less car may want to inquire about the legality of certain maneuvers.

74 **Proposition**

Suppose we have a knowledge base KB, and we want to know if a fact P is true. We will try to show that given the facts of our database, P must be true. We denote this as KB entails P.

75 **Definition**

A language used to represent knowledge should be Expressive, unambiguous, context independent, and compositional.

76 **Definition**

An algorithm should be Sound, Complete, and Efficient.

An algorithm is **Sound** if only true facts can be derived.

An algorithm is **Complete** if all true facts can be derived.

An algorithm is **Efficient** if we can call a question and receive an answer in a reasonable time/space.

Natural languages are too ambiguous to us. Programming language is not very expressive. The best language to use is logic. We can consider propositional logic, Predicate logic, and first-order logic. A logical language consists of Semantics and syntax. Semantics define what a sentence means, and syntax defines how sentences can legally be assembled.

77 **Definition**

Propositional symbols, logical connectives, parenthesis for grouping, and logical constants. Each symbol, P,Q, or R, is a atomic sentence. True and False are atomic sentences. A sentence wrapped in parenthesis is a sentence.

With out new language , we can store facts in our knowledge base.

78 **Fact**

For any sentence, now matter how complex, we can always prove or disprove it with truth tables. In other words, truth table construction is complete.

While filling out a truth table is complete, it grows exponentially. For n variables, we can have 2^n rows in the table. We can mitigate this expensive computation by using Inference rules.

79 **Definition**

Inference Rules

Given a set of facts, we can derive infinitely useless facts. Instead, we should search the space of the facts we have using the inference rules as operators. Even though we have 6 operators, the branching factor is not 6, since we will have more than one fact. This means our branching factor can be massive for a space. To resolve this we want to use a heuristic. For example, if were given a long fact and we want to resolve P, we should expand facts which involve P early on.

Representing generalizations using propositional logic is hard, so we can instead consider First order Logic. First order logic consists of

1. Terms- People, numbers, cars, books, ...
2. Functions- MotherOf, DivisorOf, CountryOfOirginOf,...
3. Predicates- isMale, isPrime, isAmerican,...
4. Connectives-

5. Quantifiers-

Predicates return True or False whereas Functions return Terms. Predicates can be ambiguous depending on order, so isBrother(Harpo, Groucho) can be read is "It is the case that Harpo is the brother of Groucho. The predicate is not automatically implied, so isBrother(x,y) $\implies$ isBrother(y,x) must be both defined explicitly.

Suppose we want to represent the fact that all Irishmen are European. We could represent this fact by explictily enconding isIrish(Eamonn) $\implies$ isEuropean(Eamonn) for all 2.5 million Irishmen, or instead we could use a universal qualifier.

$$\forall x \text{ isIrish}(x) \implies \text{isEuropean}(x)$$

Suppose now that we want to amke statements about the existence of objects. We could do this using the existencial qualifier.

$$\exists x \text{ isEuropean}(x) \implies \neg\text{isIrish}(x)$$

Propositional logic is very powerful, but bad at dealing with rare exceptions, uncertatiny, and poor at dealing with time.

80 **Definition**

A model is a "world" in which a sentence is True.

Models allow for entailments. A knowledge base entails α if and only if all models of that knowledge base also entail α.

There are three main tools used to prove things in propositional logic:

1. Universal Elimination
2. Existential Elimination
3. Existential Introduction

Substitution allows us to replace variables with specific terms, for example replacing isBrother(x,groucho) with isBrother(Harpo,Groucho).

5 Anytime Algorithms

Anytime algorithms are a general tool beyond AI, but are useful for AI agents that operate in the real world (driver-less cars, robots, etc). Often, when dealing with agents, we want to make decisions quickly since time is a relevant factor in real world decision making. While an agent may be able to make the perfect decision by computing all possible actions, it may come at a great cost of time (like hitting a pedestrian for a driver-less car). We may want an approximate answer that is more efficient to compute, so we use anytime algorithms.

81 **Definition**

An anytime algorithm is one which returns an answer even if ended before convergence. An answer returned before convergence is an approximate answer, and the longer the algorithm runs the higher the accuracy of the answer.

Certain algorithms are much better for this form of approximation than others- for example, nearest neighbor can be used to very quickly make a classification.

82 **Note**

Desireable properties of such an algorithm

1. Trade execution time for quality of results
2. Always has the best-so-far answer available
3. quality of the answer improves with execution time
4. Allows users to suspend the process during execution or continue computing as needed

The quality of the solution returned scales logarithmically with execution time- at the start of the execution, another millisecond of computation increases the accuracy far more than a millisecond of computation after an hour of computation.

The answer returned by an anytime algorithm has measurable quality, and the quality of an approximate result can be determined. Using k-fold cross validation we can approximate the quality of solution vs execution time curve which is very useful for knowing when we should stop execution (Or what time is necessary to pass a minimum quality threshold).

83 **Note**

1. Interruptability- After some setup time, the algorithm can be stopped at any time and provide an answer
2. Monotonicity- The quality of the result is a non-decreasing function of runtime
3. Diminishing returns- The improvement in quality of result is largest at the beginning of computation and decreases with time
4. Measurable Quality- The quality of the solution can be determined
5. Preemptability- The algorithm can be started/stopped quickly with minimal overhead

Some of the algorithms already covered are anytime- Greedy search, random search, simulated annealing, genetic algorithms, k-means clustering, decision tree construction, Monte Carlo game search, and more. These are already anytime without modification.

We can convert some algorithms to anytime with a bit of computation- A* or minimax, for example, can be adjusted to approximate.

84 **Proposition**

1. A*- Suppose we just run greedy search, and record the answer we get, then start classic A* and update the best so far answer. We will have an initial approximation that improves as A* runs.
2. Minimax- Suppose we preform this in an iterative deepening fashion, where we explore as deep as we can in a given time, evaluate the fringe nodes, and pass the values upward through the tree.

Some can not be converted to anytime, dendrogram construction or theorem proving (Since we can not now which way an algorithm leans, to true or false, before it converges for resolution refutation.)

Nearest neighbor can easily be turned anytime, by just checking points until we're interrupted. We must alternate between the classes or we may search only instances of one class and mis-classify a new data point. Additionally, we an compress points that we consider redundant into single instances and reduce computation time/improve accuracy. To do this, we can search over all possible orderings to find the best ordering that maximizes classification accuracy. This is an NP hard problem, but we can approximate it through an algorithm called SimpleRank. By finding a better ordering of the data, nearest neighbor converges far faster and with greater accuracy.

85 **Note**

Normal batch algorithms are often brittle for real world application. Where possible, we should use anytime algorithms, which are more robust and generally safer.