

## PEAS & Problem Formulation

- Performance Measure: Parameters used to measure the performance/efficacy of an AI model (e.g. safety of a self-driving car)
- Environment: Surrounding of an agent

<b>Fully Observable</b> Agent has access to complete state of the environment at each point in time (e.g. chess board)	<b>Partially Observable</b> Agent does not have complete info about the current state of the environment (e.g. self-driving cars)
<b>Deterministic</b> Next state can be determined fully based on current state and action (e.g. chess)	<b>Stochastic</b> Next state has some randomness that cannot be completely determined by the agent (e.g. self driving cars)
<b>Episodic</b> The choice of an agent's action is independent of other actions (e.g. pick and place robot)	<b>Sequential</b> The choice of an agent's action is dependent on previous actions (e.g. chess and self-driving cars)
<b>Static</b> Environment is unchanged while an agent is still deciding on an action (e.g. chess with no clock)	<b>Dynamic</b> Environment constantly changes, even when agent is still deciding on an action (e.g. self-driving cars)
<b>Semi-Dynamic</b> Environment does not change, but agent's performance score does	
<b>Discrete</b> Actions and percepts are clearly defined and of finite in number (e.g. chess)	<b>Continuous</b> Actions and percepts cannot be clearly defined (e.g. self-driving cars)
<b>Single Agent</b> Single agent operating by itself in an environment (e.g. pick and place robot)	<b>Multi-Agent</b> Multiple agents operating in an environment (e.g. self-driving cars on a busy road)

- Actuator: The part of an agent that delivers the output of an action to the environment (e.g. steering wheel of a self-driving car)
- Sensor: Parts of an agent that receives the input (e.g. camera & speedometer of a self-driving car)

### Problem Formulation

- State/State Space:** State/Set of all states reachable from initial state (e.g. all cities reachable from starting city)
- Initial State:** e.g. starting city
- Goal State(s)/Test:** Desired outcome (e.g. destination city)
- Actions:** Things an agent can do (e.g. move to neighbouring city)
- Transition Model:** The effect of performing an action on a state (e.g. arrive at next city)
- Action Cost Function:** Cost of performing an action (e.g. distance)

## Uninformed Search Algorithms

### Breadth-First Search (BFS)

- Time:  
 $1 + b + b^2 + \dots + b^d = O(b^d)$
- Space:  $O(b^d)$  (expand last child)
- Complete: Yes, if  $b$  is finite. If  $d$  is infinite, then yes if  $\exists$  solution
- Optimal: Yes, if step cost is uniform
- In this module, next successor is at the start of the queue

```
create frontier: queue
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state is goal: return solution
        frontier.add(next state)
        visited.add(next state)
return failure
```

### Uniform-Cost Search

- Tier: Group of paths with the same cost.
- # Tiers ("depth") =  $C^*/\epsilon$ ,  $C^*$  = cost of optimal solution,  $\epsilon$  = min. edge cost
- Time:  $O(b^{C^*/\epsilon})$
- Space:  $O(b^{C^*/\epsilon})$
- Complete: Yes, if # tiers is finite i.e.  $\epsilon > 0$  and  $C^*$  is finite
- Optimal: Yes, if  $\epsilon > 0$
- Goal test is performed after POP, because the priority queue "reshuffles" priority after PUSH

```
create frontier: priority queue (PATH-COST)
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    if state is goal: return solution
    visited.add(state)
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state not in frontier:
            frontier.add(next state)
        else if next state has lower COST:
            update state in frontier
return failure
```

### Depth-First Search (DFS)

- Time:  $O(b^m)$ ,  $m$  = max depth
- Space:  $O(bm)$  (everytime we traverse down one level,  $b$  nodes are added)
- Complete: No, when depth is infinite
- Optimal: No
- In this module, next successor is at the top of the stack

```
create frontier: stack
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state is goal: return solution
        frontier.add(next state)
        visited.add(next state)
return failure
```

### Depth-Limited Search (DLS)

- Variant of DFS; backtracks whenever  $d$  = depth limit ( $l$ )
- Time:  $b^0 + b + b^2 + \dots + b^l = O(b^l)$
- Space:  $O(bm)$
- Complete: No, solution may lie in greater depths
- Optimal: No

### Iterative-Deepening Search (IDS)

- Perform DLS with depth limit  $l = 0 \dots N$  until solution is found
- Time:  $b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + \dots + (b^0 + b^1 + \dots + b^d) = (d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$ ,  $d$  = depth where solution is found
- Space:  $O(bd)$
- Complete: Yes
- Optimal: Yes, if step cost is uniform. IDS is like "BFS-ing" a DFS
- Generates overhead =  $\frac{N_{IDS} - N_{DLS}}{N_{DLS}}$

### Bidirectional Search

- Perform BFS from each goal state, and BFS from start until the searches both search trees contain some common node(s)
- Operators need to be reversible (i.e. need to know what is the parent of each state)
- Time:  $O(b^{d/2}) + O(b'^{d/2})$ ,  $b$  = branching factor for forward BFS,  $b'$  = branching factor for backward BFS
- Space:  $\max\{O(b^{d/2}), O(b'^{d/2})\}$
- Complete: Yes
- Optimal: Yes

## Informed Search Algorithms

### Greedy Best-First Search

- Evaluation function only considers the heuristic, i.e.  $f(n) = h(n)$
- Time:  $O(b^m)$  with good heuristic
- Space:  $O(bm)$  with good heuristic
- Complete: No
- Optimal: No

```
create frontier: priority queue
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    if state is goal: return solution
    for action in actions(state):
        next state = transition(state, action)
        frontier.add(next state)
return failure
```

### A\* Search

- Evaluation function considers the cost so far,  $g(n)$ , together with heuristic, i.e.  $f(n) = g(n) + h(n)$
- Time:  $O(b^m)$  with good heuristic
- Space:  $O(bm)$  with good heuristic
- Complete: Yes.  $g(n)$  monotonically increases along a path  $\rightarrow$  if loop back to visited node, it won't explore that node again
- Optimal: Yes, depending on heuristics

### Heuristics

- Admissibility:  $h(n) \leq h^*(n)$  for all  $n$ 
  - If  $h(n)$  is admissible, A\* using tree search is optimal
  - How?: relax the problem by removing certain restrictions
- Consistency:  $h(n) \leq \text{cost}(n, n') + h^*(n')$ 
  - If  $h(n)$  is consistent, A\* is optimal
- Dominance: If  $h_1(n) \geq h_2(n)$  for all  $n$ , then  $h_1(n)$  dominates  $h_2(n)$ 
  - A dominant heuristic is better; assuming it is admissible, it is closer to the true cost

### Variants of A\* Search

- Iterative Deepening A\*: Cutoff using  $f(n)$  cost (instead of depth)
- Simplified Memory Bound A\*: Drop nodes in PQ with worst  $f(n)$  if PQ is full

## Local Search

- Starting with initial solution, move from one solution to another in solution space by applying local changes until an optimal solution is found

### Trivial Algorithms

- Random Sampling:** Generate states randomly until solution is found
- Random Walk:** Randomly pick a neighbour from current state

### Hill Climbing Algorithm

- Can get stuck at local optimum
- Simulated Annealing**
  - With some probability  $P$ , pick a **random** neighbour instead of the best one
  - $P = e^{\frac{\text{value}(\text{next}) - \text{value}(\text{current})}{T}}$
  - If  $T$  decreases slowly enough, simulated annealing will find global optimum with high probability

```

current = initial state
loop:
    neighbour = successor with highest value
    if value(current) ≥ value(neighbour):
        return current
    current = neighbour

curr = initial state
for t = 1 ... ∞:
    T = schedule(t)
    if T = 0: return current
    nxt = random successor of current
    if value(nxt) > value(curr) or P(nxt, curr, T):
        curr = next
    
```

### Beam Search

- Perform  $k$  hill-climbing in parallel. i.e. there will be  $k$  successors selected at each **level**
- Local Beam Search:** Choose  $k$  successors deterministically (i.e. best  $k$  values)
- Stochastic Beam Search:** Choose  $k$  successors probabilistically

## Adversarial Search

### Minimax

- Alternate min-max at each level
- Max nodes pick max successor;
- Min nodes pick min successor
- Time:  $O(b^m)$
- Space:  $O(bm)$
- Optimal: Yes, if players are playing optimally

```

minimax(S, α, β, maximizer):
    if terminal(S): return utility(S)
    if maximizer:
        v = -∞
        for each action in actions(S):
            if α ≥ β: break
            S' = transition(action, S)
            v = max(v, minimax(S', α, β, False))
            α = max(α, v)
    else:
        v = ∞
        for each action in actions(S):
            if α ≥ β: break
            S' = transition(action, S)
            v = min(v, minimax(S', α, β, True))
            β = min(β, v)
    return v
    
```

### Minimax with alpha-beta pruning

- $\alpha$ : min score the max player can guarantee
- $\beta$ : max player the min player can guarantee
- At max level: update  $\alpha$  only.
- At min level: update  $\beta$  only.
- Initially,  $(\alpha, \beta) = (-\infty, \infty)$
- Branches are pruned whenever  $\alpha \geq \beta$
- Ordering at the leaves can affect pruning efficiency
  - With perfect ordering time is reduced to:  $O(b^{\frac{m}{2}})$

## Supervised Learning

- Learns from **labelled** data
- Formal definition: Find a hypothesis  $h: x \rightarrow \hat{y}$  from a hypothesis class  $H$  s.t.  $h \approx f$ , where  $f: x \rightarrow y$  is a true mapping function
- Classification:** predict **discrete** output
- Regression:** predict **continuous** output

### Performance Measure

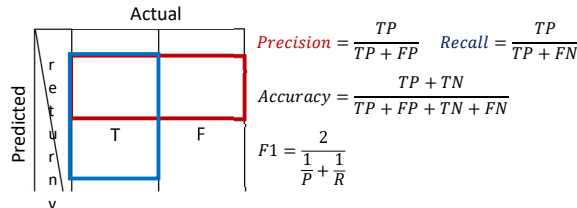
- Split data into training set and test set
  - Training set: Train learning algorithm to generate hypothesis
  - Test set: Apply hypothesis on test set to analyse performance

### Regression

- Absolute Error =  $|\hat{y} - y|$
- Squared Error =  $(\hat{y} - y)^2$
- Mean Squared Error =  $\frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2$

### Classification

- Accuracy =  $\frac{1}{N} \sum_{i=1}^N 1_{\hat{y}_i = y_i}$
- Confusion Matrix



### Decision Tree

- A tree-like structure with each internal node as a decision, each branch is an outcome of the decision and each leaf is the final prediction/decision
- Recursive pick an attribute to split remaining data until no more attributes to split OR all rows accounted for
- "Best" attributes are greedily chosen first

### Choosing "best" attribute

- Entropy  $I$** 
  - Indicates the "purity" of a decision – lower entropy indicates less noise and increased certainty
  - $I(P(v_1), \dots, P(v_n)) = -\sum_{i=1}^n P(v_i) \log_2 P(v_i)$
  - If binary decisions, then  $I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n} \log_2 \left(\frac{p}{p+n}\right) - \frac{n}{p+n} \log_2 \left(\frac{n}{p+n}\right)$
- Information Gain  $IG$** 
  - Difference in  $I$  between current and remainder
  - Remainder = sum of  $I$  of children nodes
  - This is used to decide the best attribute to use for splitting
- Gain Ratio & Split Information**
  - Attributes with very specific values (e.g. HP number) can result in very high  $IG \rightarrow$  overfitting
  - $GainRatio(A) = \frac{IG(A)}{SplitInformation(A)}$
  - $SplitInformation(A) = -\sum_{i=1}^d \frac{|E_i|}{|E|} \log_2 \left(\frac{|E_i|}{|E|}\right)$

### Cost-Normalized-Gain

- Factor in the cost of attributes and choosing low-costs ones
- Cost-Normalized-Gain** =  $\frac{IG^2(A)}{Cost(A)}$  or  $\frac{2^{IG(A)} - 1}{(Cost(A) + 1)^w}$ ,  $0 \leq w \leq 1$ 
  - $w$  is adjusted on how important the cost is (larger  $w \rightarrow$  more imp)
- Continues-Values Attributes**
  - Can be discretised into a set of intervals
- Attributes with missing values**
  - Assign most common value, drop the attribute, drop the rows etc.

### Overfitting

- DTs performance is perfect on training data
- Occam's Razor**
  - Prefer short/simple hypotheses
  - Prefer shorter decision trees
- Min-sample Pruning**
  - All leafs must have sample size greater than the min-sample
  - If sample size is smaller, then prune the parent decision node by making it a leaf with the majority decision as the value
- Max-depth Pruning**
  - Depth = # decisions in one traversal
  - Prune all decision nodes exceeding the max-depth, picking majority

## Linear Regression

- $H$  is a set of line formulas
- Regression:** Predicts a value with discrete inputs

### Loss Function

- Notations:
  - Given a set of  $m$   $(x, y)$  examples  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
  - $x^{(i)} = (x_1, \dots, x_n)$ , each  $x_i$  is a feature. Sometimes, we can prepend with the **bias**,  $x_0$
  - $h_w$  is a hypothesis  $\in H$ ,  $h_w(x^{(i)}) = w_0 x_0 + w_1 x_1^{(i)} + \dots + w_n x_n^{(i)} = \hat{y}^{(i)}$
  - $w_0 x_0$  forms the y-intercept, so that the line does not always need to intersect at  $y = 0$

### Mean Squared Error (MSE)

$$J_{MSE} = \frac{1}{2m} \sum_{i=1}^m (h_w(x^{(i)}) - y^{(i)})^2$$

### Mean Absolute Error (MAE)

$$J_{MAE} = \frac{1}{2m} \sum_{i=1}^m |h_w(x^{(i)}) - y^{(i)}|$$

### Gradient Descent

- Minimizes  $J_{MSE}$
- Main Idea: Update each weight  $w_i$  in  $w = (w_1, w_2, \dots, w_k)$  iteratively until  $J_{MSE}(w)$  is minimum (local/global)
  - How do update? Check the slope of  $J(w)$  against each  $w_j$  (using derivative), then update each  $w_j$

$$w_j \leftarrow w_j - \gamma \frac{\delta J(w_0, w_1, \dots)}{\delta w_j} = w_j - \gamma \frac{1}{m} \sum_{i=1}^m (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

	Batch GD	Mini-batch GD	Stochastic GD
# samples	All	Random subset	One random
Speed	Slow	Faster	Fastest
May escape local minima (for non-LR)?	Less likely	Yes	Yes

- MSE is **convex** for linear regression: it only has 1 minima (the global)

#### Features of different scales

- Features with large scales can dominate learning process → GD may not converge

- Mean Normalization:** Normalize each feature  $i$  of input  $x$  to

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

- Different learning rate** for each weight (smaller for features with larger scales)

#### Non-Linear Relationships

- Polynomial Regression:  $h_w(x^{(i)}) = w_0 + w_1 f_1(x_1^{(i)}) + \dots + w_n f_n(x_n^{(i)})$
- Max degree:  $n - 1$ ; any larger can lead to overfitting
- Requires appropriate scaling otherwise it can lead to the same issues encountered by features of different scales

#### Normal Equation

$$X = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

$$h_w(X) = Xw = w^T X^T$$

$$w = (X^T X)^{-1} X^T Y, \text{ assuming } X^T X \text{ is invertible}$$

- We can compute the weights using Normal Equation without going through gradient descent if  $X^T X$  is invertible
- Advantage(s)
  - Directly finds  $w$  that minimises  $J_{MSE}$
  - No feature scaling and playing around with  $\gamma$  is required
- Disadvantage(s)
  - Very slow with large number of features ( $O(n^3)$ )
  - Requires  $X^T X$  to be invertible

### Logistic Regression

- Binary Classification** with continuous, based on some probability

#### Logistic Function

- $h_w(x)$  outputs the probability of  $x$  being part of the positive class
- $h_w(x) = \sigma(w_0 + w_1 x_1 + \dots + w_n x_n)$ , where  $\sigma$  is the logistic function:

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

- A decision threshold (of usually 0.5) is used as a cut-off for classification
- The **decision boundary** separates the set of points in the negative and positive class on the plane. It intersects the decision threshold
  - Decision boundary is perpendicular to the vector  $w$

#### Loss Function

- Not ideal to use  $J_{MSE}$  – it will be non-convex
- Cross-entropy (CE)** for  $C$  classes,  $\hat{y}$  = predicted,  $y$  = actual

Cross-entropy (CE)

Binary Cross-entropy (BCE)

$$CE(y, \hat{y}) = \sum_{i=1}^C -y_i \log(\hat{y}_i) \quad BCE(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

- BCE is CE but with only 2 classes (1 or 0)
- Use BCE  $J_{BCE}$  Loss as the loss function: sum BCE across all samples

$$J_{BCE} = \frac{1}{m} \sum_{i=1}^m BCE(y^{(i)}, h_w(x^{(i)}))$$

- $J_{BCE}$  is **convex** for logistic regression

#### Gradient Descent

- Minimizes  $J_{BCE}$
- Weight update: (exactly the same as that in Linear Regression)

$$w_j \leftarrow w_j - \gamma \frac{\delta J(w_0, w_1, \dots)}{\delta w_j} = w_j - \gamma \frac{1}{m} \sum_{i=1}^m (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

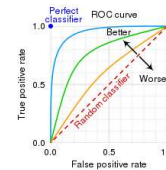
#### Non-Linear Decision Boundaries

- Transform features:  $h_w(x^{(i)}) = \sigma(w_0 + w_1 f_1(x_1^{(i)}) + \dots + w_n f_n(x_n^{(i)}))$

#### Performance Measure

- Receiver Operator Characteristic (ROC) Curve**

- TPR against FPR graph
  - TPR = TP / (TP + FN)
  - FPR = FP / (FP + TN)
- TPR against FPR graph



- Area Under Curve (AUC) of ROC Curve**

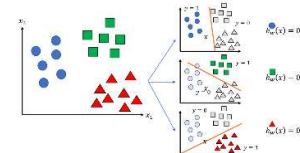
- $0 \leq \text{AUC} \leq 1$
- $\text{AUC} = 0.5 \rightarrow$  random chance; The closer to 1, the better

### Multi-class Classification

- Run logistic regression multiple times, each on different classes

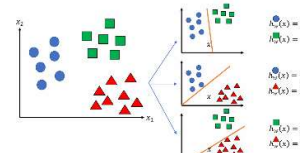
#### One-vs-All

- For each class  $C_i$ , run logistic regression on  $C_i$  and  $C - C_i$
- Pick class with **highest probability**



#### One-vs-One

- For each pair of classes  $(C_i, C_j)$ , run logistic regression on  $C_i$  and  $C_j$
- Pick class with **most wins**



### Model Evaluation

- Given dataset  $D$  and an error function (e.g. MSE, CE),  $J_D(h)$  is the expected error of a model/hypothesis  $h$

$$J_D(h) = \frac{1}{N} \sum_{i=1}^N \text{error}(h(x^{(i)}), y^{(i)})$$

- The lower the  $J_D$ , the better

#### Train-Validation-Test sets

- Training set**  $D_{train}$ : Used to train the models  $h_1, h_2, \dots, h_k$
- Validation set**  $D_{val}$ : Used to run on each of the models  $h_1$  to  $h_k$  after training, then pick the best model  $h_i$ , that is, the model with least  $J_{D_{val}}$
- Test set**  $D_{test}$ : Used to run on chosen  $h_i$  to report  $h_i$ 's performance =

#### Underfitting & Overfitting

- If  $J_{D_{val}} \approx J_{D_{train}}$ : High bias → Underfit and high training error
- If  $J_{D_{val}} \gg J_{D_{train}}$ : High variance → Overfit and low training error





