CS2109S
AY24/25S2 Midterms
github.com/zeepheru

EXAM INFO
90 minutes, $\sim$ 30 questions
SEAT: MPSH2A, 82

INTELLIGENT AGENTS

PEAS

Performance measure, Environment, Actuators, Sensors
Performance Measure considerations:

- What is best for whom? What are we optimizing? What information is available? Any unintended effects? What are the costs?
- **Rational Agent** chooses actions that maximise perf measure.

Task Environment

- **Fully** or **Partially** Observable - can observe complete state of environment?
- **Deterministic** or **Stochastic** - randomness.
- **Strategic** or **Dumb** - prescence of other intelligent agents.
- **Episodic** or **Sequential** - episodic: choice of action in atomic episode depends **only** on the episode.
- **Static** or **Dynamic** - whether environment changes while agent is deliberating.
- **Discrete** or **Continuous** - distinct, clearly defined percepts and actions
- **Single** or **Multi**-Agent

Agents

Completely specified by the agent function.
Agent structures

- **Simple Reflex** Agents - `if ... then`
- **Goal-based Agents** - knows what happens if an action is taken, pick based on goal
- **Utility-based** Agents - predicts utility
- **Learning** Agents - performance element

`Exploration` vs. `Exploitation`

SEARCH

Problem Formulation

- States, Initial State, Goal State/Test: `is_goal(state)`
- Representation Invariant
- Actions: $|actions(state)| \leq b$ `(branching factor)`
- Transition Model: `new_state = transition(state, action)`
- Action cost function

Criteria

- Completeness
  - Complete $\Leftarrow \forall$ problems, solution exists.
  - Incomplete $\Leftarrow \exists$ problem, solution **does not** exists.
- Optimal
  - Optimal $\Leftarrow \forall$ solution-producing instance, solution is the best.
- Optimal and Incomplete: $\exists$ problem with no solution, but all found solutions are optimal.

Uninformed Search

```
frontier = Frontier()

frontier.append(Node(initial_state)
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal:
        return solution

    for action in actions(node.state):
        next_state = transition(
            node.state, action)
        frontier.add(Node(next_state))

return failure
```

- **BFS** - `Queue`, explore layer-by-layer
- **UCS** - `Pqueue` (by path cost), creates "tiers" based on costs to reach nodes
- **DFS** - `Stack`, go deep then backtrack
- **Depth-Limited Search (DLS)** - limit max depth
  - Yes DLS (DFS) is not complete and not optimal
  - "Depth-Limited BFS ensures that we can find the shortest path solution if it exists. The depth limit allows us to terminate in finite time if a solution does not exist."
- **Iterative Deepening Search (IDS)** - DLS with incrementing depth limits

Informed Search

A* Search

$$f(n) = g(n) + h(n)$$

- $g(n)$: cost to reach n, $h(n)$: heuristic
- Frontier: `PQueue( f(n) )`
- Complexities are `exp`
- Complete if edges costs are positive and b is finite
- Optimality depends on heuristic

Heuristics

- **Admissible**
  - **Never over-estimates cost**, it is an optimistic estimate.
  - Theorem: if $h(n)$ admissible, then A* with visited memory is optimal.
  - **Relaxed problem**: fewer restrictions. Optimal solution to a **relaxed problem** is an admissible heuristic.
- **Consistent**: $\forall$ node N and each successor P,
  - $h(N) \leq c(N, P) + h(P)$ and $h(G) = 0$
  - Theorem: if $h(n)$ consistent, then A* with visited memory is optimal.
- **Dominant**: $\forall n,\ h_1(n) \geq h_2(n) \implies h_1$ dominates h_2

Local Search

Typically `incomplete` and `suboptimal`

	Perturbative	*Constructive*
Search Space	complete solutions	*partial solutions*
Search step	modify solution	*extend solution*

Problem Formulation

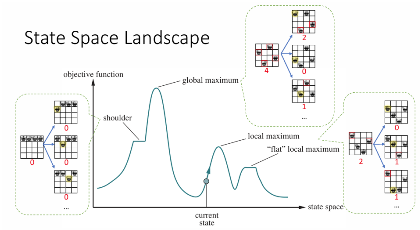
- States: may not map to actual problem state; represent **potential solutions**
- Initial State, *Goal Test (optional)*
- Successor Function: generate neighbouring states (candidate solutions)

Evaluation Function

- Want to `minimize` or `maximize`.

Hill-Climbing

```
curr_state = initial_state
while True:
    best_successor =
        """a highest-valued successor
        state of curr_state"""
    if eval(best_successor) <= eval(curr_state):
        return curr_state
    curr_state = best_successor
```



Adversarial Search

Problem Formulation

- **Terminal States**: States of win/lose/draw
- **Utility Function**: Value of state from persp of our agent. Typically need to be computed.

Minimax

`Two-player zero-sum game`

- **Complete** if tree is finite
- **Optimal** against optimally-playing opponent. Otherwise there will be *faster* strategies.

Alpha-Beta Pruning

```
def minimax(state):
    # find max value  reachable from this value
    v = max_value(state, -MAX, MAX)

    # return a next action that has that state
    return action in expand(state) with value v

def max_value(state, a, b):
    if is_terminal(state):
        return utility(state)
    v = -MAX

    # iterate through next states
    for next_state in expand(state):
        v = max(v, min_value(next_state))
        a = max(a, v)
        if v >= b: return v
    return v # max value from current state

def min_value(state, a, b):
    if is_terminal(state):
        return utility(state)
    v = MAX
    for next_state in expand(state):
        v = min(v, max_value(next_state))
        a = min(b, v)
        if v <= a: return v
    return v # min value from current state
```

Good move ordering improves pruning effectiveness.

Cutoff

Imploying a `cutoff` strategy: halt search halfway and estimate value of midgame states using an **evaluation function**.
Better handles large/infinite game trees.

MACHINE LEARNING

- **Unsupervised** Learning: `Unlabeled` data, find patterns/structure
- **Supervised** Learning: `Labeled` data, learns mapping
  - **Classification**: Predict discrete label or category
  - **Regression**: Predict continuous numerical value

Dataset $D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$
True data generating function $f^*(x)$
Hypothesis class H Set of all possible models/functions
$h : X \to Y$

Performance Measure

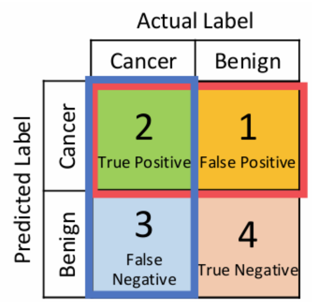
Regression

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)} \right)^2$$

$$MAE = \frac{1}{N} \sum_{i=1}^{N} \left| \hat{y}^{(i)} - y^{(i)} \right|$$

Classification

$$Accuracy = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{\hat{y}^{(i)} = y^{(i)}}$$



- **Accuracy**: $TP + TN / total$
- **Precision**: $TP/(TP + FP)$
  Number of correct out of predicted positives.
- **Recall**: $TP/(TP + FN)$
  Number predicted out of **actual** positives.
- **F1 Score**: $2 \cdot (1/P + 1/R)^{-1}$

Decision Trees

Given n boolean attributes, 2^{2^n} distinct trees.

Entropy:

$$I(P(v_1)\dots P(v_k)) = -\sum_{i=1}^{k} P(v_i)\log_2 P(v_i)$$

Information Gain:
I before dividing - weighted mean I after dividing

$$remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

$$IG(A) = I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right) - remainder(A)$$

Decision Tree Learning

Greedy, top-down, recursive.

```
def DTL(examples, attributes, default):
    if examples is empty: return default
    if examples have the same classification:
        return classification
    if attributes is empty:
        return mode(examples)
    best = choose_attribute(attributes, examples)
    tree = a new decision tree with root best
    for each value v_i of best:
        examples_i = {rows in examples with best = v_i}
        subtree = DTL(examples_i, attributes – best, mode(examples))
        add a branch to tree with label v_i and subtree subtree
```

Decision trees can have issues of **overfitting** when generalized to new test data.
Occam's Razor: prefer **short/simple** hypotheses over **long/complex**. Less likely to be concidences.
Pruning
- **Min-sample leaf**: Minimum threshold required to be in a leaf node.
  If splitting creates a leaf with nodes $<$ threshold, do not split.
- **Max-depth**: Note that depth is is of path from `root` to `leaf`

Data Preprocessing
- Partition continuous values (binning)
- Deal with missing values: assign common value, drop attribute, drop rows...

LINEAR Regression

Features $x^{(i)} \in \mathbb{R}^d$; target $y^{(i)} \in \mathbb{R}$
Linear model:

$$h_w(x) = w_0(x_0:1) + w_1x_1 + \dots + w_dx_d = w^Tx$$

Feature Transformations
- **Feature Engineering**
  - Add a new feature: $z = x^k, z = \log(x), z = e^x$
- **Feature Scaling**
  - Min-max scaling: $z_i = \frac{x_i - \min(x_i)}{\max(x_i)-\min(x_i)}$
    This one is to $[0, 1]$
  - Standardization: $z_i = \frac{x_i - \mu_i}{\sigma_i}$

Loss: MSE

$$J_{MSE}(w) = \frac{1}{2N}\sum_{i=1}^{N}\left(h_w(x^{(i)}) - y^{(i)}\right)^2$$

Normal Equation

$$w = (X^TX)^{-1}X^TY$$

- Slow, inverting requires $O(d^3)$
- X^TX needs to be `invertible`.

Gradient Descent

$$w_j \xleftarrow{update} w_j - \gamma\frac{\partial J(w_0, w_1\dots)}{\partial w_j}$$

Learning rate $\gamma ? 0$, a hyperparameter.
Theorem: A `convex` function has a single global minimum.
Theorem: MSE loss function is `convex` for linear and polynomial regression.
To deal with differently-scaled features:
- Normalize/Standardize *(or other scaling)*
- This can actually make GD `faster`.
- Different learning rate γ_i for each weight.

Variants
- Introduce `randomness`, may escape local minima
- **Mini-batch**: Consider subset of training data at a time.
- **Stochastic (SGD)**: Select **one** random data point at a time.

LOGISTIC Regression

target $y \in \{0, 1\}$

Sigmoid/Logistic Function

$$\sigma(x) = \frac{1}{1+e^{-x}} \;\Big|\; \sigma'(x) = \sigma(x)(1-\sigma(x))$$

$$h_w(x) = \sigma(w_0x_0 + w_1x_1 + \dots + w_dx_d) = \sigma(w^Tx)$$

Then compare against `decision threshold`.
Decision boundary: separates classes in feature space.

Non-linearly separable: use non-linear **feature transformations**, e.g. scale to $[-1, 1]$, then x^2.

Loss: BCE

True value y, prediction $\hat{y}$:

$$BCE(y, \hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

$$J_{BCE}(w) = \frac{1}{N}\sum_{i=1}^{N} BCE\left(y^{(i)}, h_w(x^{(i)})\right)$$

Theorem: BCE loss function is `convex` for logistic regression.

Multi-class

target $y \in \{0, 1, 2, \dots, C\}$
One-vs-one
- Separate classifier for every pair.
- Class with most votes selected.

One-vs-rest
- Separate classifier for each class against **all other** classes.
- Class with highest probability (confidence score) selected.

Supervised Learning ++

Dataset

Quality
- relevance, noise, balance (classification)

Quantity: more usually $\rightarrow$ better, but containing **all** is simply `memorization`.

Model Complexity

Size and expressiveness. e.g. `Polynomial` higher complexity than `linear`.
- `Simple model` good enough for simple truth, few data points needed
  high bias, low variance
- `Complex model` overfits if **few** data points given
  low bias, high variance
- **Bias**: Dependency to fit what it's capable of, `complex model` can fit data well $\rightarrow$ **low bias**.
- **Variance**: How much model changes as **number of data points** changes.

Hyperparameters

Predefined and adjusted manually. By comparison **parameters**, e.g. weights, are learned during training.
- learning rate, feature transformations, batch size/iterations in mini-batch

Hyperparmater Tuning
- Grid Search (exhaustive search): trying all possible combinations
- Random Search: randomly sampling rates and Hyperparameters
- Local Search: e.g. Hill climb

MISC

$$\log_b(a) = \frac{\log_x(a)}{\log_x(b)}$$

Search	Time	Space	Complete	Optimal
BFS	exp	exp	✓	✓
UCS	exp	exp	✓	✓
DFS	exp	m^k	×	×
DLS (DFS)	exp	exp	×	×
IDS	exp	m^k	✓	✓
A*	exp	exp	✓*	*depends*
Minimax	exp, $O(b^m)$	m^k	✓*	✓*

Time complexity: nodes generated; Space complexity: size of frontier.

More text.