# Mitchell-machine-learning

# Some notes and solutions to Tom Mitchell's *Machine Learning* (McGraw Hill, 1997)

Peter Danenberg

24 October 2011

## Contents



# 1 TODO An empty module that gathers the exercises' dependencies

such that running `chicken-install -s` installs them.

# 2 Exercises

## 2.1 DONE 1.1

CLOSED: *2011-10-12 Wed 04:21*

**Appropriate animal languages** could craft appropriate responses and prompts, perhaps, though ignorant of the semantics.

**fugues** train on bach data, or buxtehude. performance measure? perfect authentic cadence, of course. ;) no, not that simple.

**narratives** learn the structure of narratives? performance measure is tricky here.

**Not appropriate comedy** requires a bizarre ex nihilo and sponteneity (distinguishable from three above?) in fact, the second and third above are inappropriate, rather? define "inappropriate": difficult? vague performance measure?

**data representation and search** or have meta-learning-problems been solved?

**new science and mathematics** can "creativity" be modelled?

So we can't, indeed, escape the question of modelling; once the mechanics of learning have been mastered, there lies the ex nihilo.

## 2.2 DONE 1.2

CLOSED: *2011-10-12 Wed 04:21*

Learning task: produces melodic answers to query phrases. Given a phrase that ends on a dominant, say, within a key; gives an appropriate response that ends on the tonic. Must follow a constrained set of progressions (subdominant to dominant, dominant to tonic, flat-six to neopolitan, etc.), and be of an appropriate length.

**task T** constructing answering phrases to musical prompts (chords)

**performance measure P** percent of answers that return to the dominant once at the end (given appropriate length and progression constraints)

**training experience E** expert (bach, chopin, beethoven) prompts and answers.

**target function** $V$ : progression $\rightarrow \mathfrak{R}$; $V(b = \text{final tonic}) = 100$, $V(b = \text{final non-tonic}) = -100$.

**target function representation** $\hat{V}(b) = w_0 + w_1 x_1$, where $x_1 =$ length of prompt$-$ number of chords in answer

## 2.3 DONE 1.3

CLOSED: *2011-10-12 Wed 12:46*

Here's a solution for the trivial case in which $|\langle b, V_{\text{train}}(b)\rangle| = 1$ and the target function $\hat{V}$ consists of a single feature $x$ and a single weight $w$:

$$\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}(V_{\text{train}}(b) - \hat{V}(b))^2 \tag{1}$$

$$= 2(V_{\text{train}}(b) - \hat{V}(b))\frac{\partial}{\partial w}(V_{\text{train}}(b) - \hat{V}(b)) \tag{2}$$

$$= 2(V_{\text{train}}(b) - \hat{V}(b))(0 - x) \tag{3}$$

$$= -2(V_{\text{train}}(b) - \hat{V}(b))x \tag{4}$$

which gives:

$$w_{n+1} = w_n - \frac{\partial E}{\partial w} \tag{5}$$

$$\propto w_n + \eta(V_{\text{train}}(b) - \hat{V}(b))x \qquad \text{(by 4)} \tag{6}$$

It should be trivial to extend this to the case where $\mathbf{w}$ and $\mathbf{X}$ are vectors; the LMS training rule, furthermore, covers the summation.

## 2.4 DONE 1.4

CLOSED: *2011-10-12 Wed 18:21*

"Generating random legal board positions" is an interesting sampling strategy that might expose the performance system to improbable positions that form the "long tail" of possible board positions; the resultant hypothesis might not be optimized for common positions, though.

"Generating a position by picking a previous game and applying one of the moves not executed" is a good exhaustive search strategy which may or may not be feasible, depending on the state-space complexity of the game.

One mechanism might be to generate positions from expert endgames in reverse: you're starting with a pruned search space such that, by the time you're generating from openings, you might already have a reasonably good hypothesis. If e.g. world championship games share characteristics with expert games, this may be a reasonable heuristic for competitions. On the other hand, the hypothesis would be vulnerable to paradigmatic shifts in expert play.

If an exhaustive search of the state-space in infeasible and training examples were held constant, I'd bet on reverse-search of an expert-corpus over random sampling; especially if, *vide supra*, championship and expert play share certain characteristics.

## 2.5 TODO 1.5

# 3 Notes

## 3.1 Chapters

### 3.1.1 1

- a computer program is said to learn from experienc E with respect to some class of tasks T and performance measure P, if its performanc at tasks in T, as measured by P, improves with experience E.
- neural network, hidden markov models, decision tree
- artificial intelligence :: symbolic representations of concepts
- bayesian :: estimating values of unobserved variables
- statistics :: characterization of errors, confidence intervals
- attributes of training experience:
  - type of training experience from which our system will learn
    - direct or indirect feedback

      **direct** individual checkers board states and the correct move for each

      **indirect** move sequences, final outcomes
      - credit assignment: game can be lost even when early moves are optimal
  - degree to which learner controls sequence of training examples
  - how well it represents the distribution of examples over which the final system performance P must be measured
    - mastery of one distribution of examples will not necessary (sic) lead to strong performance over some other distribution
- task T: playing checkers; performance measure P: percent of games won; training experience E: games played against itself.
- 1. the exactly type of knowledge to be learned; 2. a representation for this target knowledge; 3. a learning mechanism.
- program: generate legal moves: needs to learn how to choose the best move; some large search space
- class for which the legal moves that define some large search space are known a priori, but for which the best search strategy is not known
- target function :: choosemove : B -> M (some B from legal board states to some M from legal moves)

  - very difficult to learn given the kind of indirect training experience available
  - alternative target function: assigns a numerical score to any given board state

- alternative target function :: V : B -> R (V maps legal board state B to some real value)
  - higher scores to better board states
- V(b = finally won) = 100
- V(b = finally lost) = -100
- V(b = finally drawn) = 0
- else V(b) = V(b') where b' is the best final board state starting from b and playing optimally until the end of the game (assuming the oppont plays optimally, as well).
  - red black trees? greedy optimization?
- this definition is not efficiently computable; requires searching ahead to end of game.
- *nonoperational* definition
- goal: *operational* definition
- *function approximation*: $\hat{V}$ (distinguished from ideal target function V)
- the more expressive the representation, the more training data program will require to choose among alternative hypotheses
- $\hat{V}$ linear combination of following board features:

  $\mathbf{x}_1$ number of black pieces

  $\mathbf{x}_2$ number of red pieces

  $\mathbf{x}_3$ number of black kings

  $\mathbf{x}_4$ number of red kings

  $\mathbf{x}_5$ number of black pieces threatened by red

  $\mathbf{x}_6$ number of red pieces threatened by black
- $\hat{V} = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$
- $w_0 \ldots w_6$ are weights chosen by the learning algorithm
- partial design, learning program:

  **T** playing checkers

**P** percent games won

**E** games played against self

**target function** V : Board $\rightarrow \Re$

**target function representation** $\hat{V} = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$ two: design choices

- require set of training examples, describing board state b and training value $V_{\text{train}}$(b) for b: ordered pair $\langle b, V_{train}(b) \rangle$: $\langle \langle x_1 = 3, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0 \rangle, +100 \rangle$.
- less obvious how ta assign training values to the more numerous intermediate board states
- $V_{train}(b) \leftarrow \hat{V}(Successor(b))$
- $Successor(b)$ denotes the next board state following $b$ for which it is again the program's turn to move
  - train separately red and black
- $\hat{V}$ tends to be more accurate forboard states closer to game's end
- best fit: define the best hypothesis, or set of weights, as that which minimizes the squared error E between the training values and the values predicted by the hypothesis $\hat{V}$

  $E \equiv \Sigma_{\langle b, V_{train}(b) \rangle \in training\ examples} (V_train(b) - \hat{V}(b))^2$

  in statistics and signal processing, a minimum mean square error (MMSE) estimator describes the approach which minimizes the mean square error (MSE), which is a common measure of estimator quality.

  the term MMSE specifically refers to estimation in a bayesian setting, since in the alternative frequentist setting there does not exist a single estimator having minimal MSE.

  let $X$ be an unknown random variable, and let $Y$ be a known random variable (the measurement). an estimator $\hat{X}(y)$ is any function of the measurement $Y$, and its MSE is given by

  $MSE = E\left\{(\hat{X} - X)^2\right\}$

  where the expectation is taken over both $X$ and $Y$.

  $cov(X) = E[XX^T]$

http://en.wikipedia.org/wiki/Minimum_mean_square_error

in statistics, the mean square error or MSE of an estimator is one of many ways to quantify the difference between an estimator and the true value of the quantity being estimated. MSE

> is a risk function, corresponding to the expected value of the squared error loss or quadratic loss. . . the defference occurs because of randomness or because the estimator doesn't account for information that could produce a more accurate estimate.

http://en.wikipedia.org/wiki/Mean_squared_error

- thus we seek the weights, or equivalently the $\hat{V}$, that minimize $E$ for the observed training examples
  - damn, statistics would make this all intuitive and clear
- several algorithms are known for finding weights of a linear function that minimize E; we require an algorithm that will incrementally refine the weights as new training examples become available and that will be robust to errors in these estimated training values.
- one such algorithm is called the least mean squares, or LMS training rule.

> least mean squares (LMS) algorithms is a type of adaptive filter used to mimic a desired filter by finding the filter coefficients that relate to producing the least mean squares of the error signal (difference between the desired and the actual signal). it is a stochastic gradient descent method in that the filter is only adapted based on the error at the current time.
> the diea behind LMS filters is to use steepest descent to find filter weight h(n) which minimize a cost function:
> $C(N) = E\left\{|e(n)|^2\right\}$
> where e(n) is the error at the current sample 'n' and E{.} denotes the expected value.
> this cost function is the mean square error, and is minimized by the LMS.
> applying steepest descent means to take the partial derivatives with respect to the individual entries of the filter coefficient (weight) vector, where $\bigtriangledown$ is the gradient operator:
> $\hat{h}(n+') = \hat{h}(n) - \frac{\mu}{2} \bigtriangledown C(n) = \hat{h}(n) + \mu E\{x(n)e^*(n)\}$
> where $\frac{mu}{2}$ is the step size. that means we have found a sequential update algorithm which minimizes the cost function. unfortunately, this algorithm is not realizable until we know $E\{x(n)e^*(n)\}$.
> for most systems, the expectation function must be approximated. this can be done with the following unbiased estimator:
> $\hat{E}\{x(n)e^*(n)\} = \frac{1}{N}\sum_{i=0}^{N-1} x(n-i)e^*(n-i)$
> where N indicates the number of samples we use for that estimate.
> the simplest case is $N = 1$:
> $\hat{h}(n+1) = \hat{h}(n) + \mu x(n)e^*(n)$

http://en.wikipedia.org/wiki/Least_mean_squares_filter

> in probability theory and statistics, the expected value (or expectation value, or mathematical expectation, or mean, or first moment) of a random variable is the integral of the random variable with respect to its probability measure.
>
> for discrete random variables this is equivalent to the probability-weighted sum of the possible values.
>
> for continuous random variables with a density function it is the probability density-weighted integral of the possible values.
>
> it os often helpful to interpret the expected value of a random variable as the long-run average value of the variable over many independent repetitions of an experiment.
>
> the expected value, when it exists, is almost surel the limit of the sample mean as sample size grows to infitiny.

http://en.wikipedia.org/wiki/Expected_value

  - damn, everytime we encroach something interesting; find out why differential equations, linear algebra, probability and statistics are so important. that's like two years of fucking work, isn't it? or at least one? maybe it's worth it, if we can pull it

- LMS weight update rule: for each training example $\langle b, V_{train}(b)\rangle$:
  - use the current weights to calculate $\hat{V}(b)$
  - for each weight $w_i$, update it as: $w_i \leftarrow w_i + \eta(V_{train}(b) - \hat{V}(b))x_i$
- here $\eta$ is a small constant (e.g., 0.1) that moderates the size of the weight update.
- notice that when the error $V_{train}(b) - \hat{V}(b)$ is zero, no weights are changed. when $V_{train}(b) - \hat{V}(b)$ is positive (i.e., when $\hat{V}(b)$ is too low), then each weight is increased in proportion to the value of its correpsonding feature. this will raise the value of $\hat{V}(b)$, reducing the error. notice that if the value of some feature $x_i$ is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board.
  - mastering these things takes practice; the practice, indeed, of mastering things; long haul, if crossfit, for instance, is to be believed; and raising kids
  - don't forget: $V_{train}(b)$ (for intermediate values) is $\hat{V}(Successor(b))$, where $\hat{V}$ is the learner's current approimation to V and where $Successor(b)$ denotes the next board state following b for which it is again the program's turn to move

- performance system :: solve the given performance task (e.g. playing checkers) by using the learned target function(s). it taks an instance of a new problem (game) as input and produces a trace of its solution (game history) as output (e.g. select its next move at each step by the learned $\hat{V}$ evaluation function). we expect its performance to improve as this evaluation function becomes increasingly accurate.

- critic :: takes history or trace of the game produces as output set of training examples of the target function: $\{\langle b_1, V_{train}(b_1)\rangle, \ldots, \langle b_n, V_{train}(b_2)\rangle\}$.

- generalizer :: takes as input training examples, produces an output hypothesis that is its estimate of the target function. it generalizes from the specific training examples, hypothesizing a general function that covers these examples and other cases beyond the training examples. generalize correpsonds to the LMS algorithm, and the output hypothesis is the function $\hat{V}$ described by the learned weight $w_0, \ldots, w_6$.

- experiment generator :: takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e. initial board state) for the performance system to explore. more sophisticated strategioes could involve creating board positions designed to explore particular regions of the state space.

- many machine learning systems can be usefully characterized in terms of these four generic modules.

```
digraph design {
  generator [label="Experiment Generator"]
  performer [label="Performance System"]
  critic [label=Critic]
  generalizer [label=Generalizer]
  performer -> critic [label="Solution trace"]
  critic -> generalizer [label="Training examples"]
  generalizer -> generator [label=Hypothesis]
  generator -> performer [label="New problem"]
}
```

- restricted type of knowledge to a single linear eval function; constrained eval function to depend on only six specific board features; if not, best we can hope for is that it will learn a good approximation.

- let us suppose a good approximation to V can be represented thus; question as to whether this learning technique is guaranteed to find one.

- linear function representation for $\hat{V}$ too simple to capture well the nuances of the game.

  - program represents the learned eval function using an artifical neural network that considers the complete description of the board state rather than a subsect of board features.

- nearest neighbor :: store training examples, try to find "closest" stored situation
- genetic algorithm :: generate large number of candidate checkers programs allow them to play against each other, keeping only the most successful programs
- explanation-based learning :: analyze reasons underlying specific successes and failures
- learning involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner.
- many chapters preset algorithms that search a hypothesis space defined by some underlying representation (linear functions, logical descriptions, decision trees, neural networks); for each of these hypotheses representations, the correpsponding learning algorithm takes advantage of a different underlying structure to organize the search through the hypothesis space.
- . . . confidence we can have that a hypothesis consistent with the training data will correctly generalize to unseen examples
- what algorithms exist?
- how much training data?
- prior knowledge?
- choosing useful next training experience?
- how to reduce the learning task to one of more function approximation problems?
- learner alter its representation to improve ability to represent and learn the target function?
- determine type of training experience (games against experts, games against self, table of correct moves, . . . ); determine target function (board -> move, board -> value, . . . ); determine representation of learned function (polynomial, linear function, neural network, . . . ); determine learning algorithm (gradient descent, linear programming, . . . ).

## 3.2 Exercises

### 3.2.1 1.3

From page 11: "The LMS training rule can be viewed as performing a stochastic gradient-descent search through the space of possible hypotheses (weight values) to minimize the squared error $E$."

- Gradient descent is a first-order optimization algorithm. To find a local minimum of a function . . . one takes steps proportional to the negative of the gradient of the function at the current point.
  - If one takes steps proportional to the positive of the gradient, one approaches a local maximum: gradient ascent.
- Known as steepest descent.
- If $F(x)$ is defined and differentiable in a neighborhood of point $a$, $F(x)$ decreases fastest if one goes from $a$ in the direction of the negative gradient of $F$ at $a$, $-\bigtriangledown F(a)$.
- If $b = a - \gamma \bigtriangledown F(a)$ for $\gamma > 0$, then $F(a) \geq F(b)$.
- One starts with a guess $x_0$ for a local minimum of $F$, and considers the sequence $x_0, x_1, \ldots$ such that $x_{n+1} = x_n - \gamma_n \bigtriangledown F(x_n), n \geq 0$.
- We have $F(x_0) \geq F(x_1) \geq \cdots$.
- Gradient descent can be used to solve a system of linear equations, reformulated as a quadratic minimization problem, e.g., using linear least squares.
- Convergence can be made faster by using an adaptive step size.

### 3.2.2 1.4

$$\begin{aligned} \text{Training} &\xrightarrow{\text{Games against self}} V \\ &\xrightarrow{\text{Board}\to\text{value}} \text{Representation} \\ &\xrightarrow{\text{Linear function}} \text{Algorithm} \\ &\xrightarrow{\text{Gradient descent}} \text{Design} \end{aligned}$$

Figure 1: Summary of design

**Experiment generator** Take as input the current hypothesis and output a new problem for the performance system to explore. Our experiment generator always proposes the same initial board game. More sophisticated

strategies could involve creating board positions designed to explore particular regions of the state space.

### 3.2.3 1.5

"Non-operational" definition of $V(b)$:

$$V(b) = \begin{cases} 100 & b \text{ is a final winning board state} \\ -100 & b \text{ is a final losing board state} \\ 0 & b \text{ is a final drawing board state} \\ V(b') & \text{otherwise, where } b' \text{ is an optimal final board state} \end{cases} \tag{7}$$

of which the operational approximation is $\hat{V}(b)$.

Whereas for checkers:

$x_1$ black pieces

$x_2$ red pieces

$x_3$ black kings

$x_4$ red kings

$x_5$ black pieces threatened

$x_6$ red pieces threatened

For tic-tac-toe, maybe we can use the following features as a starting hypothesis:

$x_1$ number of Xs

$x_2$ number of Os

$x_3$ number of potential 3s for X

$x_4$ number of potential 3s for O

It covers forks, doesn't it? Or should we explicitly enumerate it?

Maybe, on the other hand, number of Xs and Os doesn't matter; since they increase perforce. A full enumeration of features will probably slow down analysis, won't it? Maybe that's the tradeoff: speed for precision.

$x_1$ X 3-in-a-row?

$x_2$ O 3-in-a-row?

$x_3$ X fork?

$x_4$ O fork?

$x_5$ X center?

$x_6$ O center?

$x_7$ X opposite corner?

$x_8$ O opposite corner?

$x_9$ X empty corner?

$x_10$ O empty corner?

$x_11$ X empty side?

$x_12$ O empty side?

Page 8: "In general, this choice of representation involves a crucial tradeoff. On one hand, we wish to pick a very expressive representation to allow representing as close an approximation as possible to the ideal target function $V$. On the other hand, the more expressive the representation, tho more training data the program will require in order to choose among the alternative hypotheses it can represent."

Here's a crazy thought: since the space-state complexity of tic-tac-toe is utterly tractable, let's have nine features: one corresponding to each of the squares.

How do we deal with training the opposite direction, by the way: invert the outcome of the training data?

I have no idea how much training data nine variables need: we'll have to plot it; interesting to compare a strategy containing e.g. forks and wins.

Is it interesting that each variable is binary?

Let's start with the generalizer and a catalog of games; in order to map the number of training-examples . . . Ah, I see: the second player has a fixed evaluation function. Can we abstract xkcd? Problem is, the space for O is much more complicated. Maybe we can abstract the Wikipedia strategy: # wikipedia-strategy

1. Win
2. Block
3. Fork
4. Block a fork
5. Center
6. Opposite corner
7. Empty side

(It looks like the Wikipedia strategy was abstracted from here, by the way; damn: it looks like there are separate X- and O-heuristics.)

Represent the board as a vector of nine values; can we set up abstractions for $< x, y >$ as well as `{map,reduce,for-each}-{row,column,diagonal,triplet}`?

Meh; maybe we can implement the X/O-agnostic heuristics.

```
;;;; Tic-tac-toe with heuristic player

(use debug
```

```
     vector-lib
     srfi-1
     srfi-11
     srfi-26)

;;;; General tic-tac-toe definitions

(define n (make-parameter 3))

(define (n-by-n)
  (* (n) (n)))

(define (row start)
  (iota (n) start))

(define (column start)
  (iota (n) start (n)))

(define (a) 0)

(define (b) (- (n) 1))

(define (c) (- (n-by-n) 1))

(define (d) (- (c) (- (n) 1)))

(define (ac-diagonal)
  (iota (n) (a) (+ (n) 1)))

(define (bd-diagonal)
  (iota (n) (b) (- (n) 1)))

(define (rows)
  (map row (iota (n) (a) (n))))

(define (columns)
  (map column (iota (n))))

(define (diagonals)
  (list (ac-diagonal)
        (bd-diagonal)))

(define (tuplets)
  (append (rows)
          (columns)
          (diagonals)))
```

```
(define (corners)
  (list (a) (b) (c) (d)))

(define (opposite-corner corner)
  (- (n-by-n) corner 1))

(define  -1)

(define X 0)

(define O 1)

(define X?
  (case-lambda
   ((mark) (= X mark))
   ((board space) (X? (board-ref board space)))))

(define O?
  (case-lambda
   ((mark) (= O mark))
   ((board space) (O? (board-ref board space)))))

(define empty?
  (case-lambda
   ((mark) (=  mark))
   ((board space) (empty? (board-ref board space)))))

;;;; Board mechanics

(define make-board make-vector)

(define board vector)

(define board-ref vector-ref)

(define board-set! vector-set!)

(define board-copy vector-copy)

(define board->list vector->list)

(define (board->string board)
  (apply format
         "~a~a~a~%~%~a~a~a~%~%~a~a~a~%"
         (vector->list
```

```
          (vector-map
           (lambda (i mark)
             (cond ((X? mark) "X")
                   ((O? mark) "O")
                   (else " ")))
           board))))

(define (display-board board)
  (display (board->string board)))

(define (make-empty-board)
  (make-board (n-by-n) ))

;;; Functional variant of Knuth shuffle: partitions the cards around a
;;; random pivot, takes the first card of the right-partition, repeat.

(define shuffle
  (case-lambda
   ((deck) (shuffle '() deck))
   ((shuffled-deck deck)
    (if (null? deck)
        shuffled-deck
        (let ((pivot (random (length deck))))
          (let ((left-partition (take deck pivot))
                (right-partition (drop deck pivot)))
            (shuffle (cons (car right-partition) shuffled-deck)
                     (append left-partition (cdr right-partition)))))))))

(define (make-random-board)
  (let ((board (make-empty-board)))
    (let iter ((moves (random (n-by-n)))
               (indices (shuffle (iota (n-by-n)))))
      (if (zero? moves)
          board
          (let ((mark (random (length indices))))
            ;; You may end up with a board where there are more Os
            ;; than Xs.
            (vector-set! board
                         (car indices)
                         (if (even? moves) X O))
            (iter (- moves 1) (cdr indices)))))))

(define (fold-tuplet cons nil board)
  (fold (lambda (tuplet accumulatum)
          (cons tuplet accumulatum))
        nil
```

```
        (tuplets)))

;;;; Play mechanics

(define (empty-spaces board)
  (vector-fold (lambda (space empty-spaces mark)
                 (if (empty? mark)
                     (cons space empty-spaces)
                     empty-spaces))
               '()
               board))

;;; Putting tuplet first would allow you to use many boards.
(define (first-empty-space board tuplet)
  (find (cute empty? board <>) tuplet))

(define (winning-tuplet? player? tuplet board)
  (let ((non-player-marks
         (filter (cute (complement player?) board <>) tuplet)))
    (equal? (map (cute board-ref board <>) non-player-marks)
            `(,))))

;;; The solutions here may be non-unique: in which case, we have a
;;; convergent fork.
(define (winning-spaces player? board)
  (fold-tuplet
   (lambda (tuplet winning-spaces)
     (if (winning-tuplet? player? tuplet board)
         (cons (first-empty-space board tuplet) winning-spaces)
         winning-spaces))
   '()
   board))

(define (forking-space? player? player space board)
  (let ((board (board-copy board)))
    (board-set! board space player)
    (> (length (winning-spaces player? board)) 1)))

(define (forking-spaces player? player board)
  (filter (lambda (space)
            (forking-space? player? player space board))
          (empty-spaces board)))

(define (center-space board) (/ (- (n-by-n) 1) 2))

(define (center-empty? board)
```

```
  (empty? board (center-space board)))

(define (opposite-corners player? board)
  (let* ((corners (filter (lambda (corner)
                            (player? (board-ref board corner)))
                          (corners)))
         (opposite-corners
          (map opposite-corner corners)))
    (filter (lambda (opposite-corner)
              (empty? board opposite-corner))
            opposite-corners)))

(define (empty-corners board)
  (filter (cute empty? board <>) (corners)))

(define (random-ref list)
  (list-ref list (random (length list))))

(define (random-empty-space board)
  (random-ref (empty-spaces board)))


(define (win? player? board)
  (fold-tuplet
   (lambda (tuplet win?)
     (or (every player?
                (map (cute board-ref board <>) tuplet))
         win?))
   #f
   board))

(define (X-win? outcome) (X? outcome))

(define (O-win? outcome) (O? outcome))

(define (draw? outcome) (empty? outcome))

(define (outcome board)
  (cond ((win? X? board) X)
        ((win? O? board) O)
        ((null? (empty-spaces board)) )
        (else #f)))

;;;; Player mechanics

(define (make-random-player player? player opponent? opponent)
```

```
  (lambda (board)
    (random-empty-space board)))

;;; http://www.buzzle.com/articles/tic-tac-toe-strategy-guide.html
(define (make-heuristic-player player? player opponent? opponent)
  (lambda (board)
    (let ((my-winning-spaces (winning-spaces player? board)))
      (if (null? my-winning-spaces)
          (let ((losing-spaces (winning-spaces opponent? board)))
            (if (null? losing-spaces)
                (let ((my-forking-spaces (forking-spaces player? player board)))
                  (if (null? my-forking-spaces)
                      (let ((opponent-forking-spaces
                             (forking-spaces opponent? opponent board)))
                        (if (null? opponent-forking-spaces)
                            (if (center-empty? board)
                                (center-space board)
                                (let ((opposite-corners (opposite-corners player? board)))
                                  (if (null? opposite-corners)
                                      (let ((empty-corners (empty-corners board)))
                                        (if (null? empty-corners)
                                            (random-empty-space board)
                                            (random-ref empty-corners)))
                                      (random-ref opposite-corners))))
                            (random-ref opponent-forking-spaces)))
                      (random-ref my-forking-spaces)))
                (random-ref losing-spaces)))
          (random-ref my-winning-spaces)))))

(define (make-heuristic-X-player)
  (make-heuristic-player X? X O? O))

(define (make-heuristic-O-player)
  (make-heuristic-player X? X O? O))

(define make-default-X-player
  (make-parameter make-heuristic-X-player))

(define make-default-O-player
  (make-parameter make-heuristic-O-player))

;;; Can we get rid of =move= if we simply cycle through X and O;
;;; thence recurse?
(define play
  (case-lambda
   (()
```

```
    (play (make-empty-board)))
   ((board)
    (play ((make-default-X-player))
          ((make-default-O-player))
          board))
   ((X-player O-player board)
    (play 0 X-player O-player board))
   ((move X-player O-player board)
    (debug move)
    (display-board board)
    (or (outcome board)
        (let-values (((token player)
                      (if (even? move)
                          (values X X-player)
                          (values O O-player))))
          (let ((next-move (player board)))
            (board-set! board next-move token)
            (play (+ move 1)
                  X-player
                  O-player
                  board)))))))

(use test
     debug)

(include "tic-tac-toe.scm")

(let ((board (board X X
                    O O X
                    O O X)))
  (test "winning-spaces with X"
        '(2 2)
        (winning-spaces X? board))
  (test "winning-spaces with O"
        '(2)
        (winning-spaces O? board)))

(let ((board (board X X
                    O O X
                    O  X)))
  (test "empty-spaces"
        '(7 2)
        (empty-spaces board)))

(let ((board (board X
                     X O
```

```
                     O)))
  (test "forking-spaces"
        '(6 2 1)
        (forking-spaces X? X board)))

(let ((board (make-empty-board)))
  (test-assert "center-empty? on empty board"
               (center-empty? board)))

(let ((board (board
                     X
                      )))
  (test-assert "center-empty? on non-empty board"
               (not (center-empty? board))))

(let ((board (board X

                    X  )))
  (test "opposite-corners"
        '(8 2)
        (opposite-corners X? board)))

(let ((board (make-empty-board)))
  (test-assert "random-empty-space"
               (< (random-empty-space board) (n-by-n))))

(let* ((board (board X
                     X O O
                     X  ))
       (outcome (outcome board)))
  (test-assert "Win for X"
               (X-win? outcome))
  (test-assert "No win for O"
               (not (O-win? outcome))))

(let ((board (board O X O
                    X O O
                    X O X)))
  (test-assert "Draw outcome"
               (draw? (outcome board))))

(let ((board (board O X O
                    X O O
                    X O )))
  (test-assert "No outcome yet"
               (not (outcome board))))
```

```
(debug (play))
```

Instead of this heuristics-based approach, we could map out the entire game tree from any arbitrary position; either in real time, or before: greedily picking whichever position yields the most wins.

Maybe that's better; should we plot it as an alternative to the heuristics-based player: the deterministic player?

Each node has three values: wins for X, wins for O, draws; it may be possible to do it functionally through memoization: let's just ascend up the tree when we've explored a child, however, and update the parent accordingly.

W.r.t. the heuristic player, by the way, can't we get rid of `space-marks` and rely on `board-ref` instead? Triplets (tuplets) are just lists of indices, then. Good call.

We should have two graphs, by the way: training games vs. games one against 1) the heuristic and 2) the deterministic player.

The heuristic player has the interesting property that it can continue from illegal positions; the deterministic player would be confined to legal positions, wouldn't it?

Should we create directories for modules? Should the players be stateful or stateless? If the e.g. deterministic player is stateless, we should probably analyse the game off-line. We can instantiate the player with a token and predicate (i.e. X and X? or O and O?); the player receives a game state and returns a space. Some kind of intermediary updates the game space and, in theory, arbitrates for legality, etc. Initially, though, we can trust the agents.

The deterministic player is going to have to turn the state into a tree if we're using something precomputed: but we don't necessarily have the benefit of history; we'd have to flatten the history by using an e.g. hash-table. Even then, how do we distinguish among parents and look-alikes when calculating win-draw-loss?

Let's compute the tree in real time to see what sort of complexity we're dealing with; then we can think about optimizing. Since we have a stateful player, let's memoized by history; even though there are equivalent branches. We might be able to use rotation to prune the terminal by a factor of ten.

```
(use
 debug
 srfi-9
 srfi-69
 )

(include "tic-tac-toe.scm")

(define-record move
  parent
  space
```

```
  X-wins
  O-wins
  draws)

#;
(define-record-printer move
  (lambda (move out)
    (format out
            "#<move space: ~a X-wins: ~a O-wins: ~a draws: ~a>"
            (move-space move)
            (move-X-wins move)
            (move-O-wins move)
            (move-draws move))))

(define-record-printer move
  (lambda (move out)
    (format out
            "#,(move ~a ~a ~a ~a ~a)"
            #f
            (move-space move)
            (move-X-wins move)
            (move-O-wins move)
            (move-draws move))))

(define-reader-ctor 'move make-move)

;;; Simple plumb without update
(define (plumb board)
  (let ((outcome (outcome board)))
    (or outcome
        (let ((empty-spaces (empty-spaces board)))
          (let-values (((player player?)
                        (if (odd? (length empty-spaces))
                            (values X X?)
                            (values O O?))))
            (map (lambda (empty-space)
                   (cons empty-space
                         (let ((board (board-copy board)))
                           (board-set! board empty-space player)
                           (plumb board))))
                 empty-spaces))))))

(define (hash-board board)
  (list->string (map integer->char (board->list board))))

(define (update-parents! hash board outcome move)
```

```
  (let ((board (board-copy board)))
    (let ascend ((move move))
      (if move
          (begin
            ;; (display-board board)
            (cond ((X-win? outcome)
                   (move-X-wins-set! move (+ 1 (move-X-wins move)))
                   (hash-table-update!/default
                    hash
                    (hash-board board)
                    (lambda (move)
                      (move-X-wins-set! move (+ 1 (move-X-wins move)))
                      move)
                    (make-move #f 0 0 0 0)))
                  ((O-win? outcome)
                   (move-O-wins-set! move (+ 1 (move-O-wins move)))
                   (hash-table-update!/default
                    hash
                    (hash-board board)
                    (lambda (move)
                      (move-O-wins-set! move (+ 1 (move-O-wins move)))
                      move)
                    (make-move #f 0 0 0 0)))
                  (else
                   (move-draws-set! move (+ 1 (move-draws move)))
                   (hash-table-update!/default
                    hash
                    (hash-board board)
                    (lambda (move)
                      (move-draws-set! move (+ 1 (move-draws move)))
                      move)
                    (make-move #f 0 0 0 0))))
            (board-set! board (move-space move) )
            (ascend (move-parent move)))))))

;;; We should be able to flatten this considerably, since the
;;; game-state is history-agnostic (context-free); not to mention
;;; rotation.
;;;
;;; Can we flatten this by hashing the game state and updating it as
;;; though it were a tree (by e.g. mapping parents to hashes as well)?
(define (plumb hash move board)
  ;; (display-board board)
  (let ((outcome (outcome board)))
    (if outcome
        (begin (update-parents! hash board outcome move)
```

```
                ;; Could just cap the tree here with a '(), too.
                ;; outcome
                '())
          (let ((empty-spaces (empty-spaces board)))
            (let-values (((player player?)
                          (if (odd? (length empty-spaces))
                              (values X X?)
                              (values O O?))))
              (map (lambda (empty-space)
                     (let ((new-move (make-move move empty-space 0 0 0)))
                       (cons new-move
                        (let ((board (board-copy board)))
                          (board-set! board empty-space player)
                          (plumb hash new-move board)))))
                   empty-spaces))))))

#;
(let ((board (board X
                    O
                    O  X))
      (hash (make-hash-table)))
  (time (plumb hash #f (make-empty-board)))
  (debug (hash-table->alist hash)))

#;
(let ((board (make-empty-board)))
  (with-output-to-file
      "tic-tac-toe-game-tree.scm"
    (lambda () (write (plumb #f board)))))

#;
  (let ((board (make-empty-board)))
    (with-output-to-file
        "tic-tac-toe-hash-table.scm"
      (lambda ()
        (let ((hash (make-hash-table)))
          (plumb hash #f board)
          (write (hash-table->alist hash))))))

  #;
  (time (with-input-from-file
            "tic-tac-toe-game-tree.scm"
          read))

#;
  (debug (time (with-input-from-file
```

```
                  "tic-tac-toe-hash-table.scm"
                read)))

(define (make-deterministic-player game-hash
                                   player?
                                   move-player-wins
                                   player
                                   opponent?
                                   move-opponent-wins
                                   opponent)
  (lambda (board)
    (let* ((possible-moves (empty-spaces board))
           (possible-outcomes
            (map (lambda (possible-move)
                   (let ((board (board-copy board)))
                     (board-set! board possible-move player)
                     (cons possible-move
                           (hash-table-ref game-hash (hash-board board)))))
                 possible-moves)))
      ;; Disadvantage of not having a tree: we have to keep track of
      ;; the move auxiliarily; no biggy, right?
      (let ((max-wins (fold (lambda (possible-max-wins max-wins)
                              (if (> (move-player-wins
                                      (cdr possible-max-wins))
                                     (move-player-wins
                                      (cdr max-wins)))
                                  possible-max-wins
                                  max-wins))
                            (cons -1 (make-move #f 0 0 0 0))
                            possible-outcomes))
            (max-draws (fold (lambda (possible-max-draws max-draws)
                               (if (> (move-draws
                                       (cdr possible-max-draws))
                                      (move-draws
                                       (cdr max-draws)))
                                   possible-max-draws
                                   max-draws))
                             (cons -1 (make-move #f 0 0 0 0))
                             possible-outcomes))
            (min-losses (fold (lambda (possible-min-losses min-losses)
                                (if (< (move-opponent-wins
                                        (cdr possible-min-losses))
                                       (move-opponent-wins
                                        (cdr min-losses)))
                                    possible-min-losses
                                    min-losses))
```

```
                    (cons -1 (make-move #f 0 +Inf +Inf 0))
                    possible-outcomes))
    (max-win-ratio (fold (lambda (win-ratio max-win-ratio)
                           (if (> (cdr win-ratio)
                                  (cdr max-win-ratio))
                               win-ratio
                               max-win-ratio))
                         (cons -1 0)
                         (map (lambda (possible-outcome)
                                (cons (car possible-outcome)
                                      (let ((denominator
                                             (+ (move-opponent-wins (cdr possible-ou
                                                (move-player-wins (cdr possible-outc
                                                (move-draws (cdr possible-outcome)))
                                        (if (zero? denominator)
                                            +Inf
                                            (/ (move-player-wins (cdr possible-outco
                                               denominator)))))
                              possible-outcomes))))
(debug possible-outcomes
       ;; This is an alternative, not taking draws into
       ;; consideration.
       (map (lambda (possible-outcome)
              (cons* (car possible-outcome)
                     (let ((denominator
                            (+ (move-opponent-wins (cdr possible-outcome))
                               (move-player-wins (cdr possible-outcome))
                               (move-draws (cdr possible-outcome)))))
                       (if (zero? denominator)
                           +Inf
                           (/ (move-player-wins (cdr possible-outcome))
                              denominator)))
                     (let ((denominator
                            (+ (move-opponent-wins (cdr possible-outcome))
                               (move-player-wins (cdr possible-outcome))
                               (move-draws (cdr possible-outcome)))))
                       (if (zero? denominator)
                           +Inf
                           (/ (move-opponent-wins (cdr possible-outcome))
                              denominator)))
                     (let ((denominator
                            (+ (move-opponent-wins (cdr possible-outcome))
                               (move-player-wins (cdr possible-outcome)))))
                       (if (zero? denominator)
                           +Inf
                           (/ (move-player-wins (cdr possible-outcome))
```

```
                                        denominator)))))
                     possible-outcomes)
                max-win-ratio
                max-wins
                max-draws
                min-losses)
        (if (zero? (move-player-wins (cdr max-wins)))
            (if (zero? (move-draws (cdr max-draws)))
                (car min-losses)
                (car max-draws))
            #;(car max-wins)
            (car max-win-ratio))
        (car min-losses)))))

(define game-hash
  (make-parameter (alist->hash-table
                   (with-input-from-file
                       "tic-tac-toe-hash-table.scm"
                     read))))

(define (make-deterministic-X-player)
  (make-deterministic-player (game-hash)
                             X?
                             move-X-wins
                             X
                             O?
                             move-O-wins
                             O))

(define (make-deterministic-O-player)
  (make-deterministic-player (game-hash)
                             O?
                             move-O-wins
                             O
                             X?
                             move-X-wins
                             X))

(debug
 (play
  (make-deterministic-X-player)
  (make-heuristic-O-player)
  (make-empty-board))
 (play
  (make-heuristic-X-player)
  (make-deterministic-O-player)
```

```
(make-empty-board)))
```

Deterministic TTT will look like a tree with complete games as leaves; the tree is not complete. Eventually: every node should have summary statistics: wins, draws, losses. Moving algorithm: maximize wins; otherwise, maximize draws; otherwise, minimize losses.

Each node will contain: square, wins, draws, losses. The player moving is implicit: it depends upon the root and alternates. Root is always X, though.

Can we implement the game tree as some kind of priority queue? It will, nevertheless, be a tree of queues.

This is an interesting question, actually: is wins, draws, losses the correct order?

The deterministic player, by the way, is going to have to keep track of the game history; or plumb from the current board. We can also prune an initial game tree with the current move.

Beware: it doesn't look like opposite corner is working, by the way; nor fork, for that matter.

Here's an interesting solution to the rotation/reflection problem, by the way: reduce before hashing!

State of the union: heuristic doesn't seem to recognize e.g. forks or opposite corners; deterministic doesn't play well.

For deterministic, let's try minimax; we may need to revise our position evaluation function, even going back to trees. Interestingly, we can do a basic win/loss calculus and $+1$ and $-1$ (how to deal with draws: 0, 0.5?);

See minimax with alternate moves and alpha-beta pruning; also, principal variation. Negamax for two-player, zero-sum games; by the way.

This is interesting:

> The nodes that belong to the MAX player receive the maximun value of its [sic] children. The nodes for the MIN player will select the minimun value of its [sic] children.

See also this:

> For a win-or-lose game like chess or tic-tac-toe, there are only three possible values-win, lose, or draw (often assigned numeric values of 1, -1, and 0 respectively).

Draws are indeed ignored; continuing:

> Ultimately, each option that the computer currently has available can be a assigned a value, as if it were a terminal state, and the computer simply picks the highest value and takes that action.

Does the following imply that the heuristic approach is better?

> As veteran tic-tac-toe players know, there is no opening move which guarantees victory; so, a computer running a minimax algorithm without any sort of enhancements will discover that, if both it and its opponent play optimally, the game will end in a draw no matter where it starts, and thus have no clue as to which opening play is the "best."

A hybrid heuristic-minimax approach:

> However, minimax is still useful when employed with heuristics which approximate possible outcomes from a certain point.

Yeah, naïve minimax is worthless for tic-tac-toe; you have to rely on heuristics. Maybe we can repurpose our win-fork-center-corner-side heuristics for minimax.

On using the number of winning or losing boards:

> When counting the number of possible winning or losing boards in a subtree, you are essentially assuming that each player will always make a random move. As you noted, this will not be very effective if you play against a smart player. The scheme I outlined above instead assumes that the opponent always makes a perfect move, trying to win.

The minimax tree assumes the opponent decides perfectly.

Mark Chu-Carroll did a piece on tic-tac-toe, too:

> First, you can prune the tree: any strategy that doesn't include a potential path for you to win, you can just eliminate from consideration: it has a utility of 0. For other moves, you look at the potential paths: if move A leads into a subtree where 72% of the paths lead to a leaf in which you win, then the utility of A is 0.72. Each turn, the best strategy is the one with the highest utility; moves with equal utility are equivalent.

Interesting; with some $n$, is it possible to convert the . . . no: we can just do ratio of wins to losses. It turns out that you don't really need heuristics unless you're approximating the game-tree, *n'est-ce pas?*

> So we to compute approximate utilities. We can compute approximate utility values using a variety of techniques, including heuristics, prunings, board valuations, and partial trees.

(Gradient descent, by the way, is one of the "aha!" things that seems obvious after the fact.)

What about this comment?

> "If move A leads into a subtree where 72% of the paths lead to a leaf in which you win, then the utility of A is 0.72."
>
> No . . . if any of the moves available to the next player are a win for him, then that move is a loss for you. If all of the moves available to the next player are a loss for that player, then it's a win for you. TicTackToe is uninteresting because there's no percentage utility about it.

This is kind of funny:

> If you were really building a representation of a game tree, you wouldn't implement it that way - but no one would really implement tic-tac-toe as a game tree. If you study the TTT game tree and its utility function, you can collapse it down to virtually nothing. The full tree is interesting as an understandable abstraction, not as a practical implementation.

We did implement it as a game tree, then as a hash; but we're not taking symmetries into account and maybe we should. It could just be a matter of normalizing the board before we hash it or perform lookups.

The rotations of the board (`012 345 678`), by the way:

```
258 147 036
876 543 210
631 741 852
```

It seems like if you imagine a tenth square exists, and take  mod 3; you get the rotations. The $(n^2+1)^{\text{th}}$ square also becomes a sentinel to stop rotating.

The symmetries seem to happen by reversing triplets and swapping rows, respectively:

```
210 543 876
678 345 012
```

Naïvely, you could test the rotations and symmetries against the hash tree until you found a match; otherwise, add the hash. Lookup is similarly expensive, though.

We could hash each board as a 17-bit integer (i.e. 32 bits) instead of a string, by the way: bits e.g. 0 through 8 are the X positions; 9 through 17, O.

The win ratio does yield different results than the max analysis; give it a shot?

Just to confirm out intuition:

> The real problem with this algorithm (and where you may be getting results that you aren't expecting) is when all sub-trees result in possible losses. At that point, you will need to use a heuristic to get any better information on which move you should take. You will

> need something better than simply {-1, 0, 1}, because some moves could allow you to win, but you'd block them out because you could also lose.

Also:

> This behavior is quite easy to implement in min/max using a decay for each recursion. I.e. whenever you return something from a recursive call multiply the result by 0.9 or something like this. This will lead to higher scores for longer negative paths and smaller scores for longer positive paths.

If we do ratio of wins to wins, draws, losses; and wins are zero, we can't distinguish states anymore. At that point: ratio of draws to draws, losses; after which: minimize losses?

This guy successfully applied a heuristic to minimax, by the way:

- For each row, if there are both X and O, then the score for the row is 0.
- If the whole row is empty, then the score is 1.
- If there is only one X, then the score is 10.
- If there are two Xs, then the score is 100.
- If there are 3 Xs, then the score is 1000, and the winner is Player X.
- For Player O, the score is negative.
- Player X tries to maximize the score.
- Player O tries to minimize the score.
- If the current turn is for Player X, then the score of Player X has more advantage. I gave the advantage rate as 3.

Doesn't mention anything about forks, etc.

Let's abandon minimax, since we have a heuristic player: we'd have to integrate the heuristic with minimax, anyway; at which point minimax is superfluous.

Let's package tic-tac-toe as a module with tests, too; thet way the dependencies can be pulled with `chicken-install`.

`with-debug-level` in `debug` would be nice, too.

I'd like to take the opportunity to have the tic-tac-toe module conform to Riastradh's style rules, too.

> One simple approach has been found to be surprisingly successful: assign the training value of $V_{\text{train}}(b)$ for any intermediate board state $b$ to be $\hat{V}(\text{Successor}(b))$, where $Successor(b)$ denotes the next board state following $b$ for which it is again the program's turn to move:
>
> $$V_{train}(b) \leftarrow \hat{V}(Successor(b))$$

Not utterly different from minimax, then, in the sense that we're going to have to descend to the leave and propagate up; aren't we?

(If features *qua* board-position is insufficient, by the way, we might have to come up with features based on our heuristics.)

Problem even says, by the way, to "use a fixed evaluation function you create by hand;" which doesn't rule out minimax, of course.

We'll rate a given board on the basis of: wins, losses, forks, opponent forks, center, corners, sides.

> We will find it useful to reduce the problem of improving performance $P$ at task $T$ to the problem of learning some particular *target function* such as *ChooseMove*.
>
> An alternative target function, is an evaluation function that assigns a numerical score to any given board state.
>
> If the system can successfully learn such a target function $V$, then it can easily use it to select the best move from any current board position. This can be accomplished by generating the successor board state produced by every legal move, then using $V$ to choose the best successor state and therefore the best legal move.
>
> 1. if $b$ is not a final state in the game, then $V(b) = V(b')$, where $b'$ is the best final board state that cane be achieved starting from $b$ and playing optimally until the end of the game (assuming the opponent plays optimally as well).

This smacks of minimax; what about tic-tac-toe, where everything leads to draw? "Nonoperational" definition, anyway.

> We could allow the program to represent $\hat{V}$ using a large table with a distinct entry specifying the value for each distinct board state. Or we could allow it to represent $\hat{V}$ using a collection of rules that match against features of the board state, or a quadratic polynomial function of predefined board features, or an artifical neural network.
>
> To learn $\hat{V}$, we require a set of training examples, each describing a specific board state $b$ and the training value $V_{train}(b)$ for $b$. Each training example is an ordered pair of the form $\langle b, V_{train}(b)\rangle$.
>
> Assign the training value of $V_{train}(b)$ for any intermediate board state $b$ to be $\hat{V}(Successor(b))$, where $\hat{V}$ is the learner's current approximation to $V$ and where $Successor(b)$ denotes the next board state following $b$ for which it is again the program's turn to move:
>
> $$V_{train}(b) \leftarrow \hat{V}(Successor(b))$$
>
> Intuitively, we can see this will make sense if $\hat{V}$ tends to be more accurate for board states closer to the game's end.

Performance system takes $\hat{V}$, evaluates the possible boards, selects the best one; repeat. Produces a game history. The critic takes the trace, produces $\langle b, V_{train}(b)\rangle$. The generalizer takes the training examples and the hypothesis; updates the hypothesis accordingly. The experiment generator takes the current hypothesis and outputs a new problem for the performance system to explore. Let's say: blank board vs. heuristic player.

$V_{train}$ from $\hat{V}(Successor(b))$ is still a little fuzzy: do we trace to an endpoint, and propagate up? And what's the initial state of the algorithm: some guess; or ones for all the weights?

```
(use srfi-95)

(include "tic-tac-toe.scm")

;;; Each variable corresponds to a feature; each feature corresponds
;;; to an evaluative lambda (evaluative in the sense that evaluates
;;; the board and returns some scalar corresponding to the feature).

(define (wins player? board)
  (fold-tuplet
   (lambda (tuplet wins)
     (+ (if (every player? tuplet) 1 0) wins))
   0
   board))

(define constant (constantly 1))

;;; Hat-V corresponds to a list of weight-(hypothesis)--feature pairs;
;;; order of the weight-feature pairs doesn't matter, I think. The
;;; hypothesis-feature pairs are collectively the target function.

;;; Perform takes a board and target-function; and returns a series of
;;; moves. Do the moves contain opponent moves, or merely mine? This
;;; implies merely mine: "This approach is to assign the training
;;; value of $V_{train}(b)$ for any intermediate board state $b$ to be
;;; $\hat{V}(Successor(b))$, where $\hat{V}$ is the learner's current
;;; approximation to $V$ and where $Successor(b)$ denotes the next
;;; board state following $b$ for which it is again the program's turn
;;; to move."

;;; More specifically, it implies that any board state corresponding
;;; to an opponent's move is the same as the one that preceded it
;;; (where it is the player's turn). Due to the current mechanics of
;;; =play=, I don't think the learning player will see the final board
;;; position; since the play-broker shuts it down. We can change that.
```

```
;;; Let's output our solutions. Can we have a =fold-game=, by the way,
;;; so I don't have to rely on mutation to keep track of state?
;;; =fold-game= is a =play= with an explicit accumulator. It might
;;; need to return both the accumulation and the outcome.

;;; If board were first, unlimited number of players.
(define (fold-game move board X-player X-cons X-nil O-player O-cons O-nil)
  (let ((outcome (outcome board)))
    (if outcome
        (values X-nil O-nil outcome)
        (if (even? move)
            (let ((next-move (X-player board)))
              (board-set! board next-move token)
              (fold-game (+ 1 move)
                         X-player
                         X-cons
                         ))))))

;;; Scratch =fold-game= for the time being; we have to keep track of
;;; state and move for each player: messy and unnecessary. Let's
;;; mutate.

(define (map-possible-moves morphism token board)
  (let iter ((empty-spaces (empty-spaces board))
             (domain '()))
    (if (null? empty-spaces)
        domain
        (iter (cdr empty-spaces)
              (let ((board (board-copy board))
                    (move (car empty-spaces)))
                (board-set! board move token)
                (cons (morphism move board) domain))))))

(define play
  (case-lambda
   (()
    (play (make-empty-board)))
   ((board)
    (play ((make-default-X-player))
          ((make-default-O-player))
          board))
   ((X-player O-player board)
    (play 0 (list board) X-player O-player board))
   ((move history X-player O-player board)
    ;; (debug move)
    ;; (display-board board)
```

```
    (let ((outcome (outcome board)))
      (if outcome
          (values history outcome)
          (let-values (((token player)
                        (if (even? move)
                            (values X X-player)
                            (values O O-player))))
            (let ((next-move (player board))
                  (board (board-copy board)))
              (board-set! board next-move token)
              (play (+ move 1)
                    (cons board history)
                    X-player
                    O-player
                    board))))))))

;;; Generalize this later to X or O.
(define (perform V-hat opponent problem)
  (play (lambda (board)
          (let* ((scores->move-boards
                  (map-possible-moves
                   (lambda (move board)
                     (cons* (V-hat board) move board))
                   X
                   board))
                 ;; Beware the corner case where we have no moves;
                 ;; we'll encounter this, too, if Mitchell is to be
                 ;; believed, since e.g. <x_1 = 3, ...> -> +100.
                 (max-score (car (car (sort scores->move-boards < car))))
                 (max-move-boards (filter (lambda (score->move-board)
                                            #;
                                            (debug score->move-board
                                                   max-score)
                                            (= max-score (car score->move-board)))
                                          scores->move-boards))
                 (max-move-board (random-ref max-move-boards)))
            (cadr max-move-board)))
        opponent
        problem))

(let-values
    (((history outcome)
      (perform (lambda (board) 1)
               (make-heuristic-O-player)
               (make-empty-board))))
  (display-board (car history))
```

```
(debug history outcome))
```

The performer will evaluate according to: wins, losses, my forks, their forks, center, corners, sides.

In addition to "subjunctive" features like `winning-spaces`, `forking-spaces`, etc.; we need indicative (i.e. enumerative) ones like: `wins`, `forks`, `center?`, `corners`, `sides`. They can employ slightly simplified logic, since we don't care about the actual spaces and we don't need to enumerate possible moves. It would be nice to unify them, however, with the subjunctive features; in which case, they merely employ enumeration of moves and indication of features.

Or, fuck it: let's just `length` on the space-enumerative features. Do we need to distinguish between I-have-the-center vs. other-has-the-center? Do we need to distinguish between corners and opposite corners?

Let's implement the learning system with one feature: wins; once we have the mechanics down, we can implement other features.

> According to our formulation of the learning problem, the only training information available to our learner is whether the game was eventually won or lost.

It seems like we recurse indeed to the leaves and propagate up with one quantum of data: win or loss.

What's the point of having the game trace: what do we do with the $< b, V_{train}(b) >$ pairs?

Ah: when it comes to adjusting the weights, the board-state is finally relevant; unexpressed features will not weigh in. The Critic, furthermore, is responsible for the win $\rightarrow +100$, loss $\rightarrow -100$, draw $\rightarrow 0$ heuristics.