TTT LM: Modeling Players — Random Machine and Human

Definitions for human and random machine players, and demo games.

Jacob Peck

CSC 466

Professor Craig Graci

Spring 2011

Listing of `ttt.l`:

```
;; tic-tac-toe machine learning

;; select
(defmethod select ((l list))
  (nth (random (length l)) l)
)

;; snoc
(defmethod snoc ((s symbol) (l list))
  (append l (list s))
)

;; play
(defmethod play (&aux play avail move)
  (setf play ())
  (setf avail '(nw n ne w c e sw s se))
  (dolist (player '(x o x o x o x o x))
    (cond
      ((eq player 'x)
        (setf move (select avail))
        (setf avail (remove move avail))
        (setf play (snoc move play))
      )
      ((eq player 'o)
        (setf move (select avail))
        (setf avail (remove move avail))
        (setf play (snoc move play))
      )
    )
  )
  play
)

;; helper class - board
(defclass board ()
  (
    (nw :accessor board-nw :initform "--")
    (n :accessor board-n :initform "--")
    (ne :accessor board-ne :initform "--")
    (w :accessor board-w :initform "--")
    (c :accessor board-c :initform "--")
    (e :accessor board-e :initform "--")
    (sw :accessor board-sw :initform "--")
    (s :accessor board-s :initform "--")
    (se :accessor board-se :initform "--")
  )
)

(defmethod populate-board ((l list) &aux board turnnum player)
  (setf board (make-instance 'board))
  (setf turnnum 1)
  (setf player 'x)
  (dolist (element l)
```

```
    ; go through the list, assigning move values to the board positions
    (cond
      ((eq element 'nw)
        (setf (board-nw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'n)
        (setf (board-n board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'ne)
        (setf (board-ne board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'w)
        (setf (board-w board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'c)
        (setf (board-c board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'e)
        (setf (board-e board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'sw)
        (setf (board-sw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 's)
        (setf (board-s board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'se)
        (setf (board-se board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
    )
    (if (eq player 'x)
      (setf player 'o)
      ;else
      (progn (setf player 'x) (setf turnnum (+ 1 turnnum)))
    )
  )
  board
)

(defmethod analyze-board ((l list) &aux board value turnnum player)
  (setf board (make-instance 'board))
  (setf value 'd)
  (setf turnnum 1)
  (setf player 'x)
  (dolist (element l)
```

```
    ;; populate board, checking after each move for a win/loss/draw in terms
of player X
    (cond
      ((eq element 'nw)
        (setf (board-nw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'n)
        (setf (board-n board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'ne)
        (setf (board-ne board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'w)
        (setf (board-w board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'c)
        (setf (board-c board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'e)
        (setf (board-e board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'sw)
        (setf (board-sw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 's)
        (setf (board-s board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'se)
        (setf (board-se board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
    )
    (if (eq player 'x)
      (setf player 'o)
      ;else
      (progn (setf player 'x) (setf turnnum (+ 1 turnnum)))
    )
    ; only change if game is currently a draw
    (if (eq value 'd) (setf value (check-for-win board)))
  )
  value
)

(defmethod check-for-win ((b board) &aux value)
  (setf value (check-for-win-row-1 b))
  (if (eq value 'd) (setf value (check-for-win-row-2 b)))
  (if (eq value 'd) (setf value (check-for-win-row-3 b)))
```

```
  (if (eq value 'd) (setf value (check-for-win-col-1 b)))
  (if (eq value 'd) (setf value (check-for-win-col-2 b)))
  (if (eq value 'd) (setf value (check-for-win-col-3 b)))
  (if (eq value 'd) (setf value (check-for-win-diag-1 b)))
  (if (eq value 'd) (setf value (check-for-win-diag-2 b)))
  value
)

(defmethod check-for-win-row-1 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-nw b) :test #'equalp)
      (find #\x (board-n b) :test #'equalp)
      (find #\x (board-ne b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-nw b) :test #'equalp)
      (find #\o (board-n b) :test #'equalp)
      (find #\o (board-ne b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-row-2 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-w b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-e b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-w b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-e b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-row-3 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-sw b) :test #'equalp)
```

```
      (find #\x (board-s b) :test #'equalp)
      (find #\x (board-se b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-sw b) :test #'equalp)
      (find #\o (board-s b) :test #'equalp)
      (find #\o (board-se b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-col-1 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-nw b) :test #'equalp)
      (find #\x (board-w b) :test #'equalp)
      (find #\x (board-sw b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-nw b) :test #'equalp)
      (find #\o (board-w b) :test #'equalp)
      (find #\o (board-sw b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-col-2 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-n b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-s b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-n b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-s b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
```

```
  value
)

(defmethod check-for-win-col-3 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-ne b) :test #'equalp)
      (find #\x (board-e b) :test #'equalp)
      (find #\x (board-se b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-ne b) :test #'equalp)
      (find #\o (board-e b) :test #'equalp)
      (find #\o (board-se b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-diag-1 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-nw b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-se b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-nw b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-se b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-diag-2 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-ne b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-sw b) :test #'equalp)
    )
  )
  (setf owin
```

```
    (and
      (find #\o (board-ne b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-sw b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod visualize ((b board))
  (format t "~%~A ~A ~A~%~A ~A ~A~%~A ~A ~A~%"
    (board-nw b) (board-n b) (board-ne b)
    (board-w b)  (board-c b) (board-e b)
    (board-sw b) (board-s b) (board-se b)
  )
  NIL
)

;; visualize
(defmethod visualize ((l list) &aux board)
  (setf board (populate-board l))
  (visualize board)
  NIL
)

;; analyze
(defmethod analyze ((l list))
  (analyze-board l)
)

;; demo
(defmethod demo (&aux p)
  (setf p (play))
  (format t "~A~%" p)
  (visualize p)
  (format t "~A~%" (analyze p))
  NIL
)

;; stats
(defmethod stats ((f function) (n number) (demo t) &aux w l d p result
results)
  (setf w 0 l 0 d 0)
  (dotimes (i n)
    (setf p (apply f ()))
    (if demo (format t "~A~%" p))
    (if demo (visualize p))
    (setf result (analyze p))
    (if demo (format t "~A~%" result))
    (cond
      ((eq result 'w) (setf w (+ 1 w)))
      ((eq result 'l) (setf l (+ 1 l)))
      ((eq result 'd) (setf d (+ 1 d)))
```

```
    )
  )
  (setf results (mapcar #'probability (list w l d) (list n n n)))
  (mapcar #'list '(w l d) results)
)

;; probability
(defmethod probability ((special integer) (total integer))
  (/ (float special) (float total))
)

;; player classes
(defclass player ()
  (
    (name :accessor player-name :initarg :name)
  )
)

(defclass random-machine-player (player) ())

(defclass human-player (player) ())


;; generic play
(defmethod generic-play ((x player) (o player) &aux play move)
  (setf play ())
  (setf *avail* '(nw n ne w c e sw s se))
  (setf *play-so-far* ())
  (dolist (player '(x o x o x o x o x))
    (cond
      ((eq player 'x)
        (setf move (make-move x))
        (setf play (snoc move play))
      )
      ((eq player 'o)
        (setf move (make-move o))
        (setf play (snoc move play))
      )
    )
    (setf *play-so-far* (snoc move *play-so-far*))
  )
  play
)

;; rmp make move
(defmethod make-move ((p random-machine-player) &aux move)
  (setf move (select *avail*))
  (setf *avail* (remove move *avail*))
  move
)

;; rmp-rmp game
(defmethod demo-random-random (&aux p x o)
  (setf x (make-instance 'random-machine-player))
  (setf o (make-instance 'random-machine-player))
```

```
  (setf p (generic-play x o))
  (format t "~A~%" p)
  (visualize p)
  (format t "~A~%" (analyze p))
  NIL
)

;; human make move
(defmethod make-move ((p human-player) &aux move)
  (format t "Play so far = ~A~%" *play-so-far*)
  (visualize *play-so-far*)
  (format t "Please select a move from ~A~%" *avail*)
  (setf move (read))
  (cond
    ((not (member move *avail*))
      (make-move p)
    )
    (t
      (setf *avail* (remove move *avail*))
      move
    )
  )
)

;; rmp-human game
(defmethod demo-random-human (&aux p x o)
  (setf x (make-instance 'random-machine-player))
  (setf o (make-instance 'human-player))
  (setf p (generic-play x o))
  (format t "~A~%" p)
  (visualize p)
  (format t "~A~%" (analyze p))
  NIL
)
```

Listing of `ttt-rmp-human-demo.text`:

```
$ clisp

<...snip...>

[1]> (load "ttt.l")
;; Loading file ttt.l ...
;; Loaded file ttt.l
T
[2]> (demo-random-random)
(N SE NW E W NE C SW S)

X2 X1 O3
X3 X4 O2
O4 X5 O1
L
NIL
[3]> (demo-random-random)
(N NW E SE W SW NE S C)

O1 X1 X4
X3 X5 X2
O3 O4 O2
L
NIL
[4]> (demo-random-random)
(SW N W S NE SE E C NW)

X5 O1 X3
X2 O4 X4
X1 O2 O3
L
NIL
[5]> (demo-random-random)
(E SW W C N S SE NE NW)

X5 X3 O4
X2 O2 X1
O1 O3 X4
L
NIL
[6]> (demo-random-random)
(SE E NE C NW SW N W S)

X3 X4 X2
O4 O2 O1
O3 X5 X1
W
NIL
[7]> (demo-random-random)
(C NE E N SW W NW S SE)

X4 O2 O1
O3 X1 X2
X3 O4 X5
```

```
W
NIL
[8]> (demo-random-human)
Play so far = (S)

-- -- --
-- -- --
-- X1 --
Please select a move from (NW N NE W C E SW SE)
W
Play so far = (S W SW)

-- -- --
O1 -- --
X2 X1 --
Please select a move from (NW N NE C E SE)
E
Play so far = (S W SW E N)

-- X3 --
O1 -- O2
X2 X1 --
Please select a move from (NW NE C SE)
NW
Play so far = (S W SW E N NW SE)

O3 X3 --
O1 -- O2
X2 X1 X4
Please select a move from (NE C)
C
(S W SW E N NW SE C NE)

O3 X3 X5
O1 O4 O2
X2 X1 X4
W
NIL
[9]> (demo-random-human)
Play so far = (NW)

X1 -- --
-- -- --
-- -- --
Please select a move from (N NE W C E SW S SE)
C
Play so far = (NW C E)

X1 -- --
-- O1 X2
-- -- --
Please select a move from (N NE W SW S SE)
Sw
Play so far = (NW C E SW SE)
```

```
X1 -- --
-- O1 X2
O2 -- X3
Please select a move from (N NE W S)
Ne
Play so far = (NW C E SW SE NE S)

X1 -- O3
-- O1 X2
O2 X4 X3
Please select a move from (N W)
N
(NW C E SW SE NE S N W)

X1 O4 O3
X5 O1 X2
O2 X4 X3
L
NIL
[10]> (demo-random-human)
Play so far = (N)

-- X1 --
-- -- --
-- -- --
Please select a move from (NW NE W C E SW S SE)
NW
Play so far = (N NW SW)

O1 X1 --
-- -- --
X2 -- --
Please select a move from (NE W C E S SE)
NE
Play so far = (N NW SW NE E)

O1 X1 O2
-- -- X3
X2 -- --
Please select a move from (W C S SE)
W
Play so far = (N NW SW NE E W C)

O1 X1 O2
O3 X4 X3
X2 -- --
Please select a move from (S SE)
S
(N NW SW NE E W C S SE)

O1 X1 O2
O3 X4 X3
X2 O4 X5
D
NIL
```

```
[11]> (bye)
Bye.
```