# Assignment 1

TDT4173: Machine Learning and Case-Based Reasoning

Jonas Myrlund

September 2, 2013

# I

## 1 Give two examples of relevant machine learning problems and describe them as "well-posed learning problems"

A well-posed learning problem is a problem that is expressed in terms of a task $T$, a performance measure $P$, and experience $E$. It is said that a program is able to learn if it improves its performance $P$ at task $T$ given experience $E$.

### 1.1 Diagnosing patients

**Task**
Determining correct diagnoses for patients, given their symptoms.

**Performance measure**
Percentage of correctly diagnosed patients.

**Experience**
Patient journals, with manual correct diagnoses. Manual feedback from medical professional.

### 1.2 Playing tic-tac-toe

**Task**
Beating opponents at tic-tac-toe.

**Performance measure**
Percentage of games won.

**Experience**
Games played against itself.

## 2

### 2.1 What is inductive bias? Why is it so important in machine learning?

Inductive bias is a learning algorithm's ability to use previous experience to solve problems it hasn't explicitly faced during training.

In machine learning, we use training data to calibrate our algorithms to solve a *general* problem. Without the ability to solve previously unencountered problems, we're only solving the problem for the training data – for which we already know the solution. That really doesn't get us anywhere interesting.

Occam's razor is one example: choosing the simplest solution to a problem leads to better generalization, thereby introducing a form of inductive bias.

### 2.2 The candidate elimination algorithm for learning in version spaces and learning of decision trees with ID3 are two different learning methods. What can you say about the inductive bias for each of them?

# II

## 1 What would be a good target function representation for learning to play tic-tac-toe?

The target function $\hat{V}(b)$ being a linear combination of the board $b$'s feature vector of length $n$, we start off with defining it as:

$$\hat{V}(b) = w_0 + \sum_{i=1}^{n} w_i x_i \tag{1}$$

As features, we could choose something like the following:

$x_1$ The number of X's occurring aligned with other X's.

$x_2$ The number of O's occurring aligned with other O's.

$x_3$ The number of squares eligible for three X's in a row.

$x_4$ The number of squares eligible for three O's in a row.

The weights could be set to something seemingly reasonable to begin with, then tweaked by playing a human adversary, a random playing bot, or by letting the algorithm play against itself.

## 2 How would you represent the tic-tac-toe board in a programming language of your choice?

I would represent the board state as a two-dimensional array, and supply some simple helper functions to access its various traits.

For example in Python, I would wrap the board in a class and mix in some methods for manipulating and reasoning about it – something along these lines (quite a few methods omitted, but it expresses the general idea):

```
import copy, itertools, random

X = "X"
O = "O"
PLAYERS = (X, O)

class Board:
    _dimensions = 3

    def __init__(self, initial_board=None):
        if initial_board:
            self._board = copy.deepcopy(initial_board)
        else:
            self._board = [[None for _ in range(self._dimensions)] \
                                 for _ in range(self._dimensions)]

    def play(self, coords, player):
        """coords being a tuple of x and y, zero-indexed."""
        x, y = self._validate_coords(coords, allow_occupied=False)
        self._board[y][x] = player
        return self

    def get(self, coords):
        x, y = self._validate_coords(coords)
        return self._board[y][x]

    def diagonals(self):
        d1 = [self._board[i][i] for i in range(self._dimensions)]
        d2 = [self._board[i][self._dimensions - i - 1] \
                for i in range(self._dimensions)]
        return [d1, d2]

    def columns(self):
        return [[self._board[i][j] \
                     for i in range(self._dimensions)] \
```

```
36                      for j in range(self._dimensions)]
37
38      def alignments(self):
39          return self.columns() + self.rows() + self.diagonals()
40
41      def rows(self):
42          return [self._board[i] for i in range(self._dimensions)]
```

## 3 How would you detect the final win, loss or draw situations?

Win and loss is a simple matter of checking all alignments on the board for three of the same player symbol. There is a draw whenever there is no winner, and no more space on the board.

Here is a sample implementation, building on the above code. (The slightly obscure `is_winning_combo` method finds its right in the next task.)

```
44  # class Board: (cont.)
45      def alignments_for_player(self, player):
46          return [Board.filter_by_player(cells, player)
47                      for cells in self.alignments()]
48
49      @staticmethod
50      def filter_by_player(cells, player):
51          return filter(lambda cell: cell == player, cells)
52
53      @staticmethod
54      def is_winning_combo(cells, player):
55          return len(Board.filter_by_player(cells, player)) == Board._dimensions
56
57      def is_winner(self, player):
58          return any(Board.is_winning_combo(cells, player) \
59                      for cells in self.alignments())
60
61      def has_winner(self):
62          return any(self.is_winner(p) for p in PLAYERS)
63
64      def has_free_spaces(self):
65          for i in range(self._dimensions):
66              for j in range(self._dimensions):
67                  if self._board[i][j] is None:
68                      return True
69          return False
70
```

```
71      def is_draw(self):
72          return not (self.has_free_spaces() or self.has_winner())
```

## 4 How would you calculate the features ($x_i$) you chose for your representation?

Much in the same manner as the above. I'll extract some examples.

```
74  # class Board: (cont.)
75
76      def total_aligned_for_player(self, player):
77          player_alignments = self.alignments_for_player(player)
78          n_aligned = map(len, player_alignments)
79          doubles = filter(lambda l: l > 1 and l < self._dimensions, n_aligned)
80          return sum(doubles)
81
82      def total_spaces_eligible_for_win(self, player):
83          n = 0
84          for cells in self.alignments():
85              player_cells = Board.filter_by_player(cells, player)
86              empty_cells  = Board.filter_by_player(cells, None)
87              if len(player_cells) == (self._dimensions - 1) and \
88                 len(empty_cells) == 1:
89                  n += 1
90          return n
91
92      def total_adjacent_opponents(self, player):
93          other_player = X if player == O else O
94
95          n = 0
96          for cells in self.alignments():
97              if player in cells:
98                  opponent_cells = Board.filter_by_player(cells, other_player)
99                  n += len(opponent_cells)
100         return n
```

## 5 How would you determine which move to play next for a given board position?

The `valid_plays` method returns a list of all valid plays. What we then would like is to play the move that leads to the most desirable board setup.

We can devise a simple target function with fixed weights to show the gist of it.

```
118 # class Board (cont.)
119
120     def valid_plays(self):
121         return [(j, i) for i in range(self._dimensions) \
122                        for j in range(self._dimensions) \
123                            if self._board[i][j] is None]
124
125     def simulate_play(self, coords, player):
126         return Board(self._board).play(coords, player)
127
128 def evaluate_board(board, player):
129     other_player = X if player == O else O
130
131     if board.is_winner(player): return 100.0
132     if board.is_winner(other_player): return -100.0
133     if board.is_draw(): return 0.0
134
135     x = [
136         board.total_aligned_for_player(player),
137         board.total_aligned_for_player(other_player),
138         board.total_spaces_eligible_for_win(player),
139         board.total_spaces_eligible_for_win(other_player),
140         board.total_adjacent_opponents(player),
141     ]
142
143     w = [2.0, -2.0, 4.0, -50.0, -0.1]
144
145     # Return the £\sum_i{x_iw_i}
146     return sum(map(lambda (x, w): x * w, zip(x, w)))
147
148 def fixed_weight_play(board, player):
149     valid_plays = board.valid_plays()
150
151     results = {}
152     best_play = None
153     for play in valid_plays:
154         simulated_board = board.simulate_play(play, player)
155         results[play] = evaluate_board(simulated_board, player)
156
157         if best_play is None or results[play] >= results[best_play]:
158             best_play = play
159
160     return best_play
```

**6 How would you use training examples to improve your target function?**