

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:

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

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):

```
1  import copy, itertools, random
2
3  X = "X"
4  O = "O"
5  PLAYERS = (X, O)
6
7  class Board:
8      _dimensions = 3
9
10     def __init__(self, initial_board=None):
11         if initial_board:
12             self._board = copy.deepcopy(initial_board)
13         else:
14             self._board = [[None for _ in range(self._dimensions)] \
15                             for _ in range(self._dimensions)]
16
17     def play(self, coords, player):
18         """coords being a tuple of x and y, zero-indexed."""
19         x, y = self._validate_coords(coords, allow_occupied=False)
20         self._board[y][x] = player
21         return self
22
23     def simulate_play(self, coords, player):
24         return Board(self._board).play(coords, player)
25
26     def get(self, coords):
27         x, y = self._validate_coords(coords)
28         return self._board[y][x]
29
30     def valid_plays(self):
31         return [(j, i) for i in range(self._dimensions) \
32                     for j in range(self._dimensions) \
33                     if self._board[i][j] is None]
```

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` combomethod finds its right in the next task.)

```
35 def diagonals(self):
36     d1 = [self._board[i][i] for i in range(self._dimensions)]
37     d2 = [self._board[i][self._dimensions - i - 1] \
38           for i in range(self._dimensions)]
39     return [d1, d2]
40
41 def columns(self):
42     return [[self._board[i][j] \
43             for i in range(self._dimensions)] \
44             for j in range(self._dimensions)]
45
46 def alignments(self):
47     return self.columns() + self.rows() + self.diagonals()
48
49 def rows(self):
50     return [self._board[i] for i in range(self._dimensions)]
51 #####
52 def alignments_for_player(self, player):
53     return [Board.filter_by_player(cells, player)
54             for cells in self.alignments()]
```

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.

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 lead

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

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

...