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?

We consider two aspects of inductive bias separately: representation bias and preference bias.

ID3 uses preference bias only, arranging decision nodes in order of projected information gain.

Candidate elimination, conversely, utilises representation bias only, and assumes that the underlying concept is part of the hypothesis space.

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):

```python
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)] \
                    for j in range(self._dimensions)]

    def rows(self):
        return [self._board[i] for i in range(self._dimensions)]

    def alignments(self):
        return self.columns() + self.rows() + self.diagonals()
```

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.)

```
# class Board: (cont.)
    def alignments_for_player(self, player):
        return [Board.filter_by_player(cells, player)
                    for cells in self.alignments()]

    @staticmethod
    def filter_by_player(cells, player):
        return filter(lambda cell: cell == player, cells)

    @staticmethod
    def is_winning_combo(cells, player):
        return len(Board.filter_by_player(cells, player)) == Board._dimensions

    def is_winner(self, player):
        return any(Board.is_winning_combo(cells, player) \
                    for cells in self.alignments())

    def has_winner(self):
        return any(self.is_winner(p) for p in PLAYERS)

    def has_free_spaces(self):
```

```
        for i in range(self._dimensions):
            for j in range(self._dimensions):
                if self._board[i][j] is None:
                    return True
        return False

    def is_draw(self):
        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.

```
# class Board: (cont.)

    def total_aligned_for_player(self, player):
        player_alignments = self.alignments_for_player(player)
        n_aligned = map(len, player_alignments)
        doubles = filter(lambda l: l > 1 and l < self._dimensions, n_aligned)
        return sum(doubles)

    def total_spaces_eligible_for_win(self, player):
        n = 0
        for cells in self.alignments():
            player_cells = Board.filter_by_player(cells, player)
            empty_cells  = Board.filter_by_player(cells, None)
            if len(player_cells) == (self._dimensions - 1) and \
               len(empty_cells) == 1:
                n += 1
        return n

    def total_adjacent_opponents(self, player):
        other_player = X if player == O else O

        n = 0
        for cells in self.alignments():
            if player in cells:
                opponent_cells = Board.filter_by_player(cells, other_player)
                n += len(opponent_cells)
        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.

```
# class Board (cont.)

    def valid_plays(self):
        return [(j, i) for i in range(self._dimensions) \
                       for j in range(self._dimensions) \
                           if self._board[i][j] is None]

    def simulate_play(self, coords, player):
        return Board(self._board).play(coords, player)

def evaluate_board(board, player):
    other_player = X if player == O else O

    if board.is_winner(player): return 100.0
    if board.is_winner(other_player): return -100.0
    if board.is_draw(): return 0.0

    x = [
        board.total_aligned_for_player(player),
        board.total_aligned_for_player(other_player),
        board.total_spaces_eligible_for_win(player),
        board.total_spaces_eligible_for_win(other_player),
        board.total_adjacent_opponents(player),
    ]

    w = [2.0, -2.0, 4.0, -50.0, -0.1]

    # Return the $\sum_i x_i w_i$
    return sum(map(lambda (x, w): x * w, zip(x, w)))

def fixed_weight_play(board, player):
    valid_plays = board.valid_plays()

    results = {}
    best_play = None
    for play in valid_plays:
```

```
            simulated_board = board.simulate_play(play, player)
            results[play] = evaluate_board(simulated_board, player)

            if best_play is None or results[play] >= results[best_play]:
                best_play = play

        return best_play
```

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

Starting off with an initial weight vector w, I would through the training examples use a refining algorithm to incrementally improve the weight vector to minimize the squared error E:

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

The weight update function itself could be as simple as that of LMS (least mean squares), performing the following update for each training example number i:

$$w_i \leftarrow w_i + \eta(V_{\text{train}} - \hat{V}(b))x_i \tag{3}$$

7 What are the main differences between using this approach for tic-tac-toe game playing and using search approaches known from AI?

Learning algorithms don't assume to know the value of a node up front, but rather specifies *which factors are relevant* to node value. Furthermore, in our case, we only descend one level and try to *learn how to reason* about the node in itself.

The strengths of the minimax and alpha-beta pruning techniques lie in their ability to search deeply, without bringing previous experience into the mix.

III

1

Training

Initial

$G : \{\langle ?, ?, ?, ?, ? \rangle\}$
$S : \{\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}$

Example 1

$G : \{\langle ?, ?, ?, ?, ?\rangle\}$
$S : \{\langle Hair, Live, False, False, Flat\rangle\}$

Example 2

$G : \{\langle Hair, ?, ?, ?, ?\rangle, \langle ?, Live, ?, ?, ?\rangle, \langle ?, ?, False, ?, ?\rangle, \langle ?, ?, ?, False, ?\rangle, \langle ?, ?, ?, ?, Flat\rangle\}$
$S : \{\langle Hair, Live, False, False, Flat\rangle\}$

Example 3

$G : \{\langle Hair, ?, ?, ?, ?\rangle, \langle ?, Live, ?, ?, ?\rangle, \langle ?, ?, False, ?, ?\rangle, \langle ?, ?, ?, False, ?\rangle\}$
$S : \{\langle Hair, Live, False, False, ?\rangle\}$

Test classification

Example 1

$\langle Hair, Live, False, False, None\rangle \to True$

Example 2

$\langle Feathers, Egg, False, True, Pointed\rangle \to ?$

Example 3

$\langle Scales, Egg, True, False, Flat\rangle \to ?$

2 What can the system say about the classification of the three new examples?

While the first test example falls within the general and specific classification boundaries, the two others do not. Hence, we cannot say anything about their classification.

3 Assume that the system can ask for another training example. Which criteria should the system use to choose the training example?

A training example with the *Body* feature *Scales* would enable us to reason about test example 3 above.

Example 4

$\langle Scales, Egg, True, False, Pointed\rangle \to False$

4 Is the CE algorithm well-suited for all machine learning problems?

No.

The CE algorithm does not handle noisy training data well – in fact, it doesn't handle it at all. One single erronous training example is all that is required to disrupt the entire classifier, no matter the size of the training set.

IV

1 What rules would you create to gradually cover the training examples given for the Mammal domain in this way?

I would select the variable with the least amount of entropy, that is the variable with the least uncertainty. In our training data we have three equal candidates so far in $Birth$, $EatsMeat$ and $CanFly$.

Choosing in this way ensures a small and general decision tree, and thereby satisfies the principle of Occam's razor.

```
if Birth == "Live":
    True
else:
    False
```

2 Assume that three more training examples are added for the Mammal domain. How does this affect what rules you would pick to describe the Mammal concept?

The training data is no longer consistent with the *IfThen* model. We will need to add some branching factors where the outcome is erronous.

Birth is still a good starting point, but we will need to specify that pointy-teethed animals born from eggs are also mammals.

```
if Birth == "Live":
    True
else:
    if Birth == "Egg" and Teeth == "Pointed":
        True
    else:
        False
```

Although the tactics of *IfThen* and *CE* are similar in the way they work after training, they differ in how they build their classifiers: CE works with general and special boundaries that are pushed towards each other, accepting examples that fall between them, while the IfThen-approach builds a decision tree considering all the training examples "at once".