What Should a Classifier System Learn?

Tim Kovacs
School of Computer Science
The University of Birmingham
T.Kovacs@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~tyk
Some Questions

1. What should an LCS learn?

2. How should we measure learning?

3. What makes a problem hard?

4. How should we test an LCS?

5. Why are strength-based LCS unreliable?

6. How does XCS differ from strength-based LCS?

It seems easier to address these questions in the restricted domain of single step tasks.
Why Study Single-Step Tasks?

- Single step tasks are those in which the agent's actions do not affect which states it sees in the future (unlike multi-step tasks).

- Some applications (e.g. in data mining) are single-step.

- Single step tasks are much simpler (and so easier to study) than multi-step tasks, and yet rich enough to be interesting.

- My view is that issues and understanding transfer from the single-step case to the multi-step case.

- My approach is to first study how XCS/LCS work on single-step tasks and only later study multi-step tasks.
The Ternary Language and Boolean Functions

- We can represent BFs with (sets of) rules.

- A rule is a conjunction of literals.
  For example, the rule

  \[ \text{If } 010 \text{ then } 0 \]

  means:

  \[ \text{If the first input is } 0 \text{ and the second } 1 \text{ and the third } 0 \text{ then take action } 0. \]

- A set of rules is a disjunction of conjunctions (rules) ... so rule sets represent BFs in Disjunctive Normal Form (DNF).
Default Hiearchies

The idea:

● Code solution using general default rules and more specific exception rules.

The problems:

● Not all classifier systems support them (e.g. ZCS, XCS).

● Problems with evolving and maintaining them.
Representing Boolean Functions with LCS Syntax

- A given function $f$ may be represented many ways in this language, e.g. 4 ways to represent the 3-bit constant 0 function:

<table>
<thead>
<tr>
<th>#</th>
<th>#</th>
<th>#</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>#</td>
<td>#</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>#</td>
<td>#</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>#</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- Table:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>
Badly Representing Boolean Functions with LCS Syntax

• We can fail to represent the desired $f$. E.g. using $f =$ the constant 0 function, a representation can be:

<table>
<thead>
<tr>
<th>Incomplete</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0## → 0</td>
<td>## → 1</td>
</tr>
</tbody>
</table>

• We can also represent $f$ suboptimally:

<table>
<thead>
<tr>
<th>Overlapping</th>
<th>Non-minimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>## → 0</td>
<td>0## → 0</td>
</tr>
<tr>
<td>0000 → 0</td>
<td>1## → 0</td>
</tr>
</tbody>
</table>
XCS’s Four Properties

XCS tends to find solutions which are:

- complete
- correct
- minimal
- non-overlapping

We call these *optimal solutions* (or *optimal populations*) [O].
- The 4 constraints reduce the number of ways of representing a given function.

- However, not all functions have a unique $[O]$, e.g.:

\[
\begin{array}{c}
0 \# \rightarrow 0 \\
10 \rightarrow 0 \\
11 \rightarrow 1
\end{array} \quad = \quad \begin{array}{c}
\#0 \rightarrow 0 \\
01 \rightarrow 0 \\
11 \rightarrow 1
\end{array}
\]
Why Non-Overlapping Rules?

- XCS and Booker’s endogenous fitness LCS penalise overlapping rules to promote completeness:
  - Penalising overlaps prevents rules from bunching up in the same area.

- Completeness is desirable, but is there an intrinsic advantage in having non-overlapping rules?

- Is there a disadvantage to insisting on non-overlapping rules?
How Strong is XCS’s Bias Against Overlaps?

- Optimally general rules: rules whose conditions cannot be made any more general without becoming incorrect.
  - They are correct and general – i.e. desirable – rules.
  - Their conditions are Prime Implicants for the target function.

- The 6 multiplexer function has 36 Optimally General rules, denoted [PI].

- XCS is known to find a 16 rule subset of [PI] using a population of 400 rules.

- Can XCS find all 36 rules in [PI]?
Proportion of Prime Implicants Found by XCS

[Population size limits of 400, 800 and 1600 were used. Curves are averages of 10 runs with XCS.]
Discussion

Results:

- XCS did not find all 36 rules in [PI], even with a population of 1600 rules.

- This, despite the fact that XCS searches in a space of only $3^6 \cdot 2 = 1458$ possible rules, and generates about 12,000 rules during the test.

Conclusion:

- XCS has a very strong bias against overlapping rules.

- Consequently there are many correct and general rules it does not find.

- So maybe XCS needs a different way of encouraging completeness.
Conclusion

• In designing LCS we need to think about what properties we want a solution to have.

• We must be careful not to inhibit one property by trying to encourage another.