What Should a Classifier System Learn?

Tim Kovacs
School of Computer Science
University of Birmingham
Birmingham, B15 2TT England
T.Kovacs@cs.bham.ac.uk

Abstract- We consider the issue of how a classifier system should learn to represent a Boolean function. We identify four properties which may be desirable of a representation; that it be complete, accurate, minimal and non-overlapping, and distinguish variations on two of these properties for the XCS system. We question whether the bias against overlapping rules evident in some systems is appropriate, and find that XCS's bias against overlapping rules is very strong.

1 Introduction

This work was prompted by an earlier study of metrics for Learning Classifier Systems (LCS) during which the need to examine the objectives of learning in classifier systems soon became apparent. In order to address questions like "How well is it learning?" we found we must first address questions like "What is it trying to learn?". The latter turns out to be less straightforward than we might think, and, although we have addressed the issue in earlier work (e.g., [5]), it has been somewhat neglected in the LCS literature. Consequently, the current study was undertaken.

By "What is it trying to learn?", we mean to ask: What kind of solution is it trying to learn? That is, how should it represent a solution? What properties might a representation have, and which properties are desirable? What is an ideal solution for a given problem? What properties should an ideal solution have?

Some would suggest default hierarchies as an ideal representation of a solution, but, as we explain in section 3.5.2, Wilson's popular XCS classifier system [13] does not support them, and so, for this system at least, we are forced to consider alternatives.

1.1 The Scope of this Study

We consider the application of classifier systems to the learning of single-step problems, or, more specifically, arbitrary Boolean functions. Boolean functions are a well-defined, and, for strings of finite length, finite class of problems. Boolean functions are also easily represented and manipulated in the standard ternary LCS language, as demonstrated in [8] and to some extent in section 2.

We do not consider any issues specific to multi-step problems, and we consider only a few extensions to the framework for learning Boolean functions we introduce in the next section, but much of this work should be relevant to other cases. XCS [13] is taken as the focus of this work, but, again, much of it should apply to other LCS.

1.2 Experimental Framework

Wilson used a particular evaluation framework with XCS in [13] which has become the standard approach with XCS, and which we will refer to as *Wilson's pure explore/exploit scheme*. This scheme is less than fully satisfactory in multi-step problems as it handles exploration very inefficiently, and tends to get stuck in loops in deterministic environments. Although the scheme is more suitable for the single-step tasks we consider here, we note that the practise of using separate training and test data sets, or cross-validation, is the norm in the machine learning literature [10].

In Wilson's scheme, each time the system is presented with an input it chooses with even probability whether to perform an *explore trial* or an *exploit trial*.[1] In explore trials, the system selects an action at random from those advocated by the matching rules; that is, it performs an unbiased exploration of the available options. On such trials no attempt is made to exploit the system's current knowledge of how to maximise rewards. In exploit trials, in contrast, the system deterministically selects the action which is most highly recommended by the matching rules. That is, its behaviour is maximally biased towards exploitation of its current knowledge – there is no exploration involved in action selection on such trials. We refer to this as the *pure* explore/exploit scheme because the two cases represent the extremes of exploratory and exploitative behaviour.

In this scheme, learning only occurs on explore trials, and only exploit trials are used for monitoring performance, a consequence of which is that exploration has no direct effect on the statistics generated. To see this, consider the alternative of having only one type of trial, in which the system combines both exploration and exploitation. It could, for example, take the most highly recommended action 90% of the time, and some other action at random otherwise. We can then consider all trials in monitoring performance, but even if the system learns the correct action for each input, performance statistics will reach only 90% because they are influenced by exploratory behaviour.

1.2.1 Learning Boolean Functions

To use a Boolean function f as a test, on each time step we generate a random binary string of the appropriate length as input to the classifier system, and allocate one of two rewards

[1] Instead, we simply alternate between the two.

0-7803-6657-3/01/$10.00 ©2001 IEEE

depending on its response. A high reward r_h is given for responses which match the output returned by f on the input string, otherwise a low reward r_l is returned (where $r_h > r_l$).

Please note that because we alternate between explore and exploit trials we interleave training and testing, rather than make them separate phases. Note also that we train the system on all possible input strings, meaning that our set of training inputs and our set of testing inputs are the same.

1.3 Outline of the Paper

This work is broadly structured as follows. In section 2 we review a number of subjects to do with the representation of Boolean functions and condition-action rules, in preparation for later sections. In section 3 we consider what properties the solutions found by a classifier system should possess. Then, in section 4, we investigate the popular XCS classifier system's tendency to find solutions with a particular property; a lack of overlaps between rules. Finally, section 5 concludes.

2 Representation

Here we review certain subjects to do with representation, which will enable us to deal more easily with Boolean functions and special sets of rules in later sections. In section 2.1 we review various ways of representing Boolean functions: truth tables, on-sets, off-sets, and Sigma notation. We also cover disjunctive normal form. Next, in section 2.2, we briefly review the representation of condition-action rules using the standard ternary LCS language, which should be familiar to anyone who has worked with LCS. Finally, in section 2.3 we consider the representation of Boolean functions using sets of rules in the LCS language.

2.1 Representing Functions

We consider functions which are mappings of each possible binary string of a given length to either 0 or 1. Perhaps the most straightforward way to represent such a function is by its *truth table*, which is an exhaustive listing of all input/output pairs of the function. As an example, figure 1 shows the truth table for the 3-bit multiplexer function, so called because the first bit is used as an index into the remaining two bits, and the value of an input string is the value of the indexed bit. For example, the value of 010 is 1 as the first bit dictates that the value of the string is that of the first of the remaining two bits. In contrast, the value of 110 is 0 as the first bit dictates that the second bit of the remaining two determines the value of the string.

2.1.1 On-sets and Off-sets

One way to represent a Boolean function more parsimoniously than with its truth table is to represent only the inputs which map to 1, called the *on-set* of the function, since we can assume the unrepresented inputs map to 0. Figure 2 represents the 3 multiplexer using its on-set.[2]

2.1.2 Sigma Notation

We can represent functions even more compactly by assuming the inputs in the truth table are in a regular order, and by identifying table rows, rather than explicitly listing the inputs they contain. This is the approach taken with sigma notation, in which the 3 multiplexer is represented as: $\Sigma(2,3,5,7)$. (The first row in the table is considered row 0, and we list only the rows in the on-set.) This notation actually specifies a class of functions unless we know how many rows the table has. We can either assume a priori that we are working with a table of a given length, or we can assume the length of the table is the lowest power of 2 which will contain the specified rows. We will take the latter approach here.

2.1.3 Disjunctive Normal Form

The strings shown in the input column of figure 1 are implicit conjunctions of their components. We could have written the conjunctions explicitly, in which case the string in the first row would be $0 \wedge 0 \wedge 0$, where $\wedge$ is the logical AND operator. The two forms are logically equivalent so explicit use of $\wedge$ is redundant. An expression which is a conjunction of the characters in the input to a Boolean function is called a *minterm*. Each row in the input column of figure 1 is a minterm. A representation which expresses disjunctions of minterms is in *Disjunctive Normal Form* (DNF) (see, e.g., [4]).

2.2 Representing Rules

A number of representations for condition-action rules (called *classifiers*) have been used with LCS, in particular a number of variations based on binary and ternary strings. Using what we will call the *standard ternary LCS language* each rule has a single condition and a single action. Conditions are fixed length strings from $\{0, 1, \#\}$, while rule actions and environmental inputs are fixed length strings from $\{0, 1\}$.

A rule's condition c matches an environmental input m if for each character m_i the character in the corresponding position c_i is identical or the wildcard (#). For example, the condition 00# matches two inputs: 000 and 001. The wildcard is the means by which rules generalise over environmental states; the more #s a rule contains the more general it is. Since actions do not contain wildcards the system cannot generalise over them.

We will write a rule as a condition followed by an arrow $\rightarrow$ followed by the action advocated by that rule, e.g., the rule $000 \rightarrow 0$ says "If the input is 000 then take action 0".

[2] Equivalently, we can represent a function just by its *off-set*, the set of inputs which map to 0, although to avoid unnecessary complications we will not make further reference to this approach.

Input	Output
000	0
001	0
010	1
011	1
100	0
101	1
110	0
111	1

Figure 1: Truth table for the 3 multiplexer.

Input	Output
010	1
011	1
101	1
111	1

Figure 2: The 3 multiplexer, with only the on-set (rows mapping to 1) visible.

$$00\# \rightarrow 0$$
$$01\# \rightarrow 1$$
$$1\#0 \rightarrow 0$$
$$1\#1 \rightarrow 1$$

Figure 3: The 3 multiplexer function represented using a set of classifiers.

2.3 Representing Functions with Sets of Rules

A classifier maps some subset of the possible input strings to an action, and this can be taken as a partial specification of a function (or a full specification if the classifier happens to match all inputs). We can think of the classifier as equivalent to the rows in a truth table whose input column it matches. Any Boolean function can by fully specified by an appropriate set of classifiers; as an example, figure 3 shows one way of representing the 3 multiplexer using a set of classifiers.

2.3.1 Sets of Classifier Conditions are in DNF

Note that when we denote a classifier as, e.g., $\#\#\# \rightarrow 0$, we are implicitly conjuncting the characters in its condition, as we did with the minterms of section 2.1.3. Since the conditions of classifiers are satisfied (or not) independently, a set of classifiers is an implicit disjunction of them. Thus, a set of classifier conditions in the standard ternary language is in DNF.

3 How Should a Classifier System Represent a Solution?

In simple, widely-used test problems like the multiplexers it is straightforward to determine by inspection whether a particular rule acts correctly and whether it is too general, or not general enough. But what should a set of rules look like? Even for simple tasks this is not so obvious.

Michalski provides another perspective on this issue:

> "For any given set of facts, a potentially infinite number of hypotheses can be generated that imply these facts. Background knowledge is therefore necessary to provide the constraints and a preference criterion for reducing the infinite choice to one hypothesis or a few most preferable ones.
>
> A typical way of defining such a criterion is to specify the preferable properties of the hypothesis - for example, to require that the hypothesis is the shortest or the most economical description consistent with all the facts..." [9] p. 89.

Michalski suggests that defining a preference for some hypotheses is not an academic issue but a necessity, given the number of possible hypotheses. In the following we suggest and discuss four properties which we may wish to require of hypotheses learnt by LCS. These four apply to other machine learning systems, and other properties are of course possible.

We begin this section by briefly contrasting the value of individual rules and the value of a set of rules. In section 3.1 we demonstrate how we can determine the value of a single rule in isolation. Then, in following sections, we suggest four properties which are inherent to *sets* of rules and which may be desirable. In section 3.3 we identify two important properties of any set of rules an LCS uses to represent a Boolean function, that the set be complete and correct. Then, in sections 3.4 and 3.5, we identify two further properties of a set, that it be minimal and that it lack overlaps between rules, which may also be desirable.

Following investigation of these four properties we discuss some additional subjects. In section 3.6 we discuss what we call "optimal" sets of rules, in section 3.7 we cover the interpretation of conflicting sets of rules, and in section 3.8 we discuss some differences between representation in XCS and other LCS.

Before we begin, we note that we can distinguish between requiring that a rule set as a whole possess some property, or, less strictly, that some subset of it does. For example, we can insist that the classifier population *is* a minimal representation of a function. Alternatively, we may only insist that it *contains* a minimal representation. The latter may be sufficient depending on our purposes.

Input	Output
000	0
001	0
010	0
011	0
100	0
101	0
110	0
111	0

Figure 4: Truth table for the 3-bit constant 0 function.

3.1 The Value of a Single Rule

Part of the appeal of classifier systems is that they work with condition-action rules which are easily interpreted. Using a simple, well-defined problem like the 3 multiplexer from section 2.1, it is straightforward to evaluate a single rule in isolation. Let's assume the use of some kind of accuracy-based classifier system like XCS, in which the value of a rule is some function of its classification accuracy and its generality.

We can easily determine the classification accuracy of a rule. For example, it should be clear that $00\# \rightarrow 0$ is a fully accurate rule for the 3 multiplexer, as it advocates the correct action (0) for both inputs it matches (000 and 001). The (formal) generality of a rule is even easier to assess, being equal to the number of # symbols in the rule's condition. (It is more difficult to assess the effective generality of a rule, that is, the number of inputs it matches in practise.)

3.2 The Value of a Set of Rules

Although evaluating individual rules for simple problems is easy, evaluating *sets* of rules is not so straightforward. What makes one representation of a Boolean function more valuable than another? This question naturally applies not only to LCS but to other learning systems, and has been addressed in the machine learning literature. [9], for example, lists a number of criteria one might use to evaluate sets of rules.

In this section we identify four properties of rule sets relevant to learning in classifier systems.

3.3 Complete and Correct Representations

Let's consider a particularly simple function, the 3-bit constant 0 function, whose truth table is shown in figure 4. Figure 5 shows four representations of this function using the ternary LCS language.

The four representations in figure 5 do indeed represent the 3-bit constant 0 function, whereas the two sets in figure 6 do not; the left represents only a subset of the input/output cases, while the right simply does not represent the desired input/output mapping.

The four rule sets in figure 5 suffer from neither of these problems. To emphasise this, we define two properties which they share:

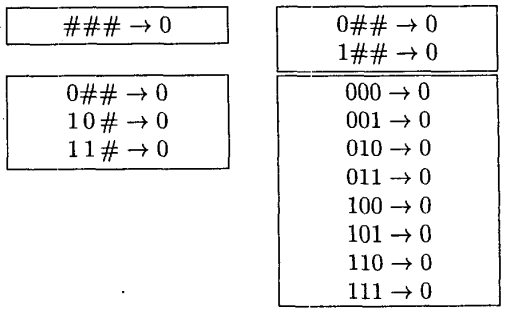

Figure 5: Four ways of representing the 3-bit constant 0 function with sets of classifiers.

Figure 6: Two sets of rules which misrepresent the constant 0 function.

Property 1. Completeness. *The rule set maps each possible input to an action.*

Property 2. Correctness. *The rule set correctly represents the intended input/output function by mapping each input to the correct action.*

Michalski formalises these two concepts in [9]. We'll see in section 3.8 that these two properties are somewhat different when we apply them to XCS.

Although these properties may seem essential requirements of any ideal solution an LCS finds, in the next section we question to what degree we should enforce correctness.

3.3.1 How Much Emphasis Should we Place on Correctness?

In section 1.2 we outlined a testing framework in which the objective of the learning agent is to approximate the data upon which it is trained as closely as possible. In this framework the training data is perfect; all input/output cases are available, and the system is tested on the same data on which it is trained. In this case, it is indeed desirable that a rule set be a correct representation of what it has learnt.

Now consider dealing with more complex frameworks, e.g., mining data from a real world database, in which the training data is incomplete, may contain uncertainty (conflicting exemplars), or for some other reason only approximates the test data (which may contain its own uncertainty). If we emphasise obtaining a rule set which is correct – that is, fully consistent with the *training* data – we risk overfitting the training data (see, e.g., [10]).

3.4 Minimal Representations

While the four representations of the constant function in figure 4 are all complete and correct, the leftmost seems prefer-

able in that it represents the function most compactly. In fact, it is a *minimal* representation of this function, using this language. In this section we consider the value of minimising the number of rules in a representation.

Property 3. Minimality. *The rule set contains the minimum number of rules needed to represent the function correctly and completely.*

3.4.1 Minimal DNF Representations

A set of rules with the three properties we have seen so far (a set which is correct, complete, and minimal) is a minimal DNF representation of a Boolean function. Thus, if the goal of our LCS is to evolve a complete, correct, and minimal set of rules, its goal is to evolve a minimal DNF representation.

3.4.2 The Desirability of Minimality

The search for general, accurate rules – and consequently smaller populations of rules – has been a continuing theme in XCS research. Indeed, the minimisation of population size has been an explicit goal of some work (e.g., [13, 5, 14, 6]). Why has work with XCS assumed smaller, but nonetheless complete and accurate, populations are more desirable than larger ones?

First, in the absence of a drop in performance, a reduction in the size of the population can be taken as an indirect indication that an LCS is finding accurate and general rules. To decrease population size without affecting performance an LCS must remove redundant rules from its population. One way to make rules redundant, without increasing population size, is to increase the proportion of accurate, general rules in the population, since such rules subsume other rules.

As an example of subsumption, suppose a population includes two accurate rules $000 \rightarrow 0$ and $100 \rightarrow 0$. Now suppose the system introduces $\#00 \rightarrow 0$, which subsumes the first two rules. If the last rule is accurate, the first two become redundant and can be removed, resulting in a net reduction of the population size.

Another reason why a reduced population size is taken as a good sign is that a very low population size and good performance are only possible when existing rules are quite general. Otherwise the population cannot cover the entire function, which would result in poor performance.

General (but accurate) rules are valuable because, without generalisation, learning systems cannot hope to adapt to complex environments. (See [14] for a brief discussion.) One consequence is that we cannot hope to understand learning in animals without understanding generalisation.

Of course, it is possible for an LCS to find accurate, general rules but not remove the rules they subsume. A reduction in population size confirms that redundant rules are indeed being removed, and suggests the method by which the LCS selects rules for deletion is effective.

Smaller populations have also been sought because parsimony has at least two benefits, the first being efficiency. The main computational cost of running a classifier system lies in processing its population of rules. Minimising the rule population by removing less valuable rules reduces the cost of running an LCS. One way in which it does so is by reducing run-times on serial machines, and the number of processors used on parallel machines. Another way in which removing redundant rules improves efficiency is that it can focus the search engine (typically a genetic algorithm) on the more valuable rules which remain in the population.

$$\begin{array}{|c|}\hline \#000 \rightarrow 1 \\ 01\#1 \rightarrow 1 \\ 1\#1\# \rightarrow 1 \\ 10\#\# \rightarrow 1 \\ \hline \end{array}$$

Figure 7: Minimal DNF rule set for $\Sigma(0, 5, 7, 8, 9, 10, 11, 14, 15)$.

A second reason to value parsimony is the argument that more parsimonious representations are more comprehensible. In general, the fewer rules the human mind needs to interpret, the more comprehensible the set of rules should be. By removing subsumed rules, such as $000 \rightarrow 0$ above, the solutions found by a classifier system may be easier for us to understand.[3] Having said this, it is debatable whether direct inspection of rule populations is the best means of extracting information from them. Perhaps tools will be developed to assist us in this.

Finally, a bias towards parsimony is commonly used in choosing between inductive hypotheses, in both machine learning systems and scientific enquiry in general. However, the issues concerning the appropriateness of this bias are somewhat involved and require far more lengthy discussion than is possible here. See [10] for discussion of Occam's razor and the minimal description length principle.

3.4.3 Disadvantages of Minimality

In addition to the advantages suggested in the last section, the pursuit of minimality may have some disadvantages. For one, the more rules the system contains, the more hypotheses it evaluates, on average, for each input. This might allow the system to find good hypotheses more quickly. Additional disadvantages of minimality will be discussed in section 3.5.

3.5 Non-Overlapping Representations

One feature of the rules in figure 7 is that they overlap; that is, there are inputs which are matched by more than one rule.[4] For example, rule 10## matches the inputs $\{1000, 1001, 1010, 1011\}$ and rule #000 matches the inputs $\{0000, 1000\}$. These rules overlap as they both match 1000.

[3] Wilson [14] also speculates that future introspective learning systems may benefit from more comprehensible representations just as humans do.

[4] Note that these rules all advocate the same action. When rules advocating different actions match the same input we have a conflict – see section 3.7.

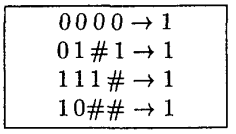

Figure 8: Minimal non-overlapping DNF rule set for $\Sigma(0, 5, 7, 8, 9, 10, 11, 14, 15)$.

Some sets of rules do not have such overlaps, which is the fourth and final property we will consider:

Property 4. A Lack of Overlaps. *No two rules advocating the same action match a common input.*

We can produce a non-overlapping version of the rules in figure 7 by reducing the generality of some of the rules, as shown in figure 8.

We note that we can classify a pair of overlapping rules into two categories: 1) one rule is subsumed by the other, and 2) neither rule subsumes the other.[5]

3.5.1 Why Non-Overlapping Solutions?

Two LCS which penalise overlapping rules are XCS, which has an *implicit* fitness bias against them, and Booker's endogenous-fitness LCS [1], which has an *explicit* fitness bias against them. In both cases, the bias against overlapping rules is designed to promote diversity in the rule population. That is, if we want to encourage a broad covering of the input (or input/action) space, we can do so by penalising redundancy in the existing covering. In other words, overlaps are penalised to encourage the completeness property. Wilson suggests another reason to penalise overlaps is to penalise subsumed rules, on the assumption that the more general of two valid hypotheses is preferable ([12] p. 227).

By penalising overlapping rules, however, we increase the odds of producing a non-overlapping solution, which is *not* a goal entailed by the desire to maintain diversity (completeness). That is, non-overlapping solutions are a byproduct of pressure to maintain diversity in a population of rules. Non-overlapping solutions are not entailed by the goal of removing subsumed rules either, but again may be a byproduct of penalties intended for such rules.

It is not clear that solutions with fewer (or no) overlaps, are advantageous. Consider what happens during training when the learner is presented with an input. If only one rule matches this input, it only has the opportunity to evaluate a single rule. But if many rules match, it can evaluate many rules using the same amount of experience.[6]

Now consider the problem of training on a subset of the input space and generating inductive hypotheses for the missing subset. In this case too it would seem advantageous to have overlaps in the missing inputs. In this case, the overlaps constitute additional hypotheses about the unknown subset which can be confirmed or refuted if experience with it later becomes available. (The same argument may hold when the training set consists of the entire input space, but when some inputs may occur infrequently, as when the training set is very large.)

As an example, consider the partially specified reward function for a 3-bit problem shown in figure 9. The inputs 000 and 101 occur, but the input 100 never occurs during the training phase. Figure 10 shows two rules which are consistent with the rewards in figure 9, and the rewards they predict they will receive. Note that these rules overlap, and that they predict different rewards: $10\# \rightarrow 0$ predicts a reward of 800 while $\#00 \rightarrow 0$ predicts a reward of 900. In other words, the two rules represent two hypotheses about how much reward will be received for taking action 0 in response to input 100, if it ever occurs. Which rule is correct? We cannot know, since figure 9 doesn't specify a value for this case. Either, or neither, rule may turn out to be correct in light of further information.

Suppose our learning system is attempting to extract regularities from some data which are then to be inspected by a human expert. Given the absence of input 100 in the training set the best our learner can do is output a range of hypotheses regarding it.

With a powerful enough syntax the system could generalise *any* rule to cover the unseen input. So the hypotheses output by the system regarding 100 may just be a result of the system's bias, and there may be no reason to think they are particularly valuable. However, if it generates hypotheses regarding 100 by generalising from nearby regions of the syntactic space, such hypotheses will tend to have above average value if the data has appropriate local continuity. (See [11].)

A final problem with a lack of overlaps is that redundancy can help make a system robust. If a valuable rule is deleted by chance, rules which are otherwise redundant can help the system cope without it.

These difficulties suggest that rather than stressing minimality, or a lack of overlaps in general, perhaps we should specifically penalise subsumed rules. This would minimise the size of the population to some extent, without being prejudicial to diversity.[7]

3.5.2 Default Hierarchies

Default hierarchies (see, e.g., [2]), in which more specific rules provide exceptions to more general rules, inherently involve overlaps between rules. Because XCS evaluates the accuracy of each rule individually, exception rules, which on their own are sometimes incorrect, are assigned low accuracy and hence low fitness. Consequently, XCS does not support default hierarchies. In other types of LCS, however, default hierarchies constitute a special form of overlap between rules, one which is clearly not undesirable (unless we consider de-

[5] We could distinguish a third case in which the two rules are equivalent.

[6] This is reminiscent of Holland's idea of implicit parallelism in genetic algorithms [3].

[7] This is precisely the effect of Wilson's subsumption deletion [14].

Input	Action	Reward
000	0	900
100	0	?
101	0	800

Figure 9: Part of a reward function.

Condition	Action	Prediction
#00	0	900
10#	0	800

Figure 10: Two overlapping rules which are consistent with the reward function in figure 9.

fault hierarchies undesirable).

Default hierarchies have been praised as a means of increasing the number of solutions to a problem without increasing the size of the search space, and as a way of allowing LCS to adapt gradually to a problem. Consequently, they have been seen by many as an important part of the representational capacity of classifier systems. However, despite much early work, the problems of encouraging their formation and survival remain unsolved, and they have attracted little attention in recent years.

3.6 Optimal Rule Sets: [O]s

In earlier work we have called a rule set with the four properties just described (completeness, correctness, minimality and non-overlappingness) an *optimal rule set* (or an *optimal population*), denoted [O] [5].

The four properties of an [O] were chosen because XCS showed a tendency to evolve solutions with these properties. However, the suggestion in section 3.5.1 that a lack of overlaps is a side-effect of pressure to promote diversity, rather than a desirable feature of a rule set, in turn suggests that its inclusion in the definition of an optimal population is unwarranted. Rather than focus on learning [O], perhaps we should focus on learning minimal DNF representations.

3.7 Conflicting Rules

When an LCS evolves rules there is typically a great deal of disagreement between them as to which action to take in a given state. To choose an appropriate action we rely on the *strength* of the rules to give us an indication of the utility of taking their actions. Consequently, for an LCS to act correctly it must assign rules appropriate strengths.

3.8 Representation in XCS

XCS differs from other LCS in that it maintains a *complete map* of the input/action space [13, 7]. That is, it tends to evolve sets of rules in which each input/action pair is covered by at least one rule. Consequently, both the correct and incorrect actions for each input will be advocated, as demonstrated

→ 0
→ 1

Figure 11: XCS represents both the correct and incorrect actions for any function, in this case a 3-bit constant function.

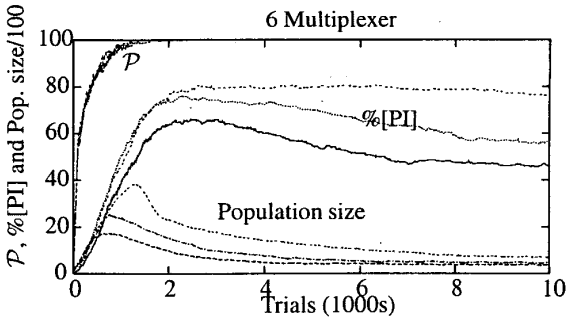

Figure 12: The effect of the population size limit on the proportion of the set of prime implicants %[PI] found by XCS for the 6 multiplexer. Population size limits of 400, 800 and 1600 were used. Curves are averages of 10 runs.

by XCS's representation of a 3-bit constant function in figure 11. (More generally, XCS will represent all actions, so for a constant function with a actions XCS will tend towards a representation with a rules.)

This tendency towards a complete map means that, when dealing with XCS, we must modify the completeness property we saw earlier.

- For XCS, a complete rule set must map each input/action pair (not just each input).

Like other LCS, XCS relies on rule strength in order to choose between rules when selecting an action. (It also factors rule fitness into its calculation [13].) Consequently, the conflicting actions represented by the rules in figure 11 are not a problem for XCS as long as it has assigned them appropriate strengths. However, a complete map means the rule set cannot simply advocate the correct action for each input.

- For XCS, a rule set must be accurate, not just correct.

For a rule to be accurate the error in its prediction of reward must fall within the threshold defined by the XCS rule updates, as defined in [13].

4 XCS's Bias against Overlapping Rules

In section 3.5.1 we questioned the desirability of penalising overlaps in the rule population and noted that XCS does so. To investigate the the extent of pressure against overlapping rules in XCS we ran it on the standard 6 multiplexer benchmark. In addition to the performance ($\mathcal{P}$) and population size statistics normally generated for XCS [13], we recorded, on each time step, the proportion of the complete set of optimally general rules that was present in the rule population.

Optimally general rules are accurate rules which cannot be made any more general without becoming inaccurate. (They are called maximally general rules in [13].) We denote this set %[PI] because the conditions of these rules are prime implicants for the function being learnt.

XCS is known to reliably find accurate, general rules for this problem [13, 5], and so, in the absence of any pressure to the contrary, should find the complete set of optimally general rules. However, this set contains many overlaps, and we hypothesised that XCS's bias against overlaps might prevent it from finding the complete set.

XCS was configured as in [13], except that we ran the niche GA in the action set, and we employed GA subsumption deletion [14]. We used the standard XCS settings from [13], and the t3 deletion scheme from [6] with a delay of 25.

It was found that XCS had great difficulty finding the set of optimally general rules, which we attribute to its strong pressure against overlapping rules. To investigate the effect of the population size limit on %[PI] we repeated the test with population size limits of 400, 800 and 1600, with results shown in figure 12. Increasing the population size limit improved the proportion of [PI] which XCS found, but even with a population of 1600 rules it was unable to find the complete set, even though this consists of only 36 rules.

At the same time, XCS is known to reliably find the complete [O] for the 6 multiplexer using a population of only 400 rules [5]. That [O] is a non-overlapping 16-rule subset of [PI] should give an indication of the magnitude of the bias against overlapping rules in XCS. We have been aware of this bias for some time, but this is the first indication of how very strong it is.

5 Summary

In response to the question "What should a classifier system learn?", we have distinguished four properties which may be desirable of a classifier system's representation, and distinguished between the required properties for strength-based and accuracy-based LCS (section 3.8). However, this work only begins to pose and answer questions of what an LCS should learn, and there is undoubtedly much more to be said on the subject. One direction in which this work should be pursued is to borrow from other areas of machine learning, some of which enjoy a greater level of sophistication than classifier systems currently do.

We have suggested that a bias against overlapping rules may be inappropriate, and demonstrated that in the popular XCS system this bias is very strong.

6 Acknowledgements

The author is grateful to the reviewers for their very helpful comments and to Manfred Kerber for our discussions.

Bibliography

[1] Lashon B. Booker. *Do We Really Need to Estimate Rule Utilities in Classifier Systems?* In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, (eds), Learning Classifier Systems. From Foundations to Applications, volume 1813 of *LNAI*, pages 125–142. Springer-Verlag, Berlin, 2000.

[2] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[3] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[4] Richard Johnsonbaugh. *Discrete Mathematics*. Macmillan, 3rd edition, 1993.

[5] Tim Kovacs. *Evolving Optimal Populations with XCS Classifier Systems*. MSc Thesis, School of Computer Science, University of Birmingham, U.K., 1996.

[6] Tim Kovacs. *Deletion schemes for classifier systems.* In W. Banzhaf et al., (eds), GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, pages 329–336. Morgan Kaufmann, 1999.

[7] Tim Kovacs. *Strength or Accuracy? Fitness Calculation in Learning Classifier Systems.* In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, (eds), Learning Classifier Systems. From Foundations to Applications, volume 1813 of *LNAI*, pages 143–160. Springer-Verlag, Berlin, 2000.

[8] Tim Kovacs and Manfred Kerber. *What makes a problem hard for XCS?* To appear in P. L. Lanzi, W. Stolzmann, and S. W. Wilson, (eds), Advances in Learning Classifier Systems, volume 1996 of *LNAI*. Springer–Verlag, Berlin, 2001.

[9] Ryszard S. Michalski. *A Theory and Methodology of Inductive Learning*. In R. Michalski, J. Carbonell and T. Mitchell, (eds), Machine Learning. An Artificial Intelligence Approach, pages 83–134. Springer–Verlag, 1983.

[10] Tom M. Mitchell. *Machine Learning*. McGraw–Hill, 1997.

[11] Geoffrey I. Webb. *Further Experimental Evidence against the Utility of Occam's Razor*. Journal of Artificial Intelligence Research (JAIR), 4: 397–417, 1996.

[12] Stewart W. Wilson. *Classifier Systems and the Animat Problem*. Machine Learning, 2: 199–228, 1987.

[13] Stewart W. Wilson. *Classifier Fitness Based on Accuracy*. Evolutionary Computation, 3(2):149–175, 1995.

[14] Stewart W. Wilson. *Generalization in the XCS classifier system.* In J. Koza et al., (eds), Genetic Programming 1998: Proceedings of the Third Annual Conference, pages 665–674. Morgan Kaufmann, 1998.