Get Real! XCS with Continuous-Valued Inputs*

Stewart W. Wilson
Prediction Dynamics
Concord, MA 01742 USA
wilson@prediction-dynamics.com

Abstract

Classifier systems have traditionally taken binary strings as inputs, yet in many real problems such as data inference, the inputs have real components. A modified XCS classifier system is described that learns a non-linear real-vector classification task.

1 Introduction

A classifier system is an on-line learning system that seeks to gain reinforcement from its environment based on an evolving set of condition-action rules called classifiers. Via a Darwinian process, classifiers useful in gaining reinforcement are selected and propagated over others less useful, leading to increasing system performance. The classifier system idea is due to John Holland (1986), who laid out a comprehensive framework that included generalization of classifier conditions, internal message-passing and reinforcement, and computational completeness. For a number of years, the idea has inspired many. A "population" of hypotheses, generated plausibly but with a random ingredient from other hypotheses and subject to confirmation by the environment, intuitively corresponds to our own mental experience, and indeed suggests a cognitive model. Classifiers that read input detectors and respond by posting messages seem to bridge the gap between physical stimuli and concepts. Generalization by classifiers to cover related input situations seems to correspond to the formation of rudimentary perceptions. The extension to learning and thinking of variation and selection—successful and ubiquitous elsewhere in life—seems natural and deserving of extensive investigation.

While its genes were promising, the evolution of the classifier system idea has not been rapid. As a complex adaptive system that also needed to be successful against a relatively independent environment, learning classifier systems (LCS) were found not to work well "out of the box". Instead, progress has come via simplifications of LCS structure, controlled experiments with environments that tested individual aspects of the system, and, of course, changes in the original spec. Among the problems encountered were performance better

---

*From *Festschrift in Honor of John H. Holland*, May 15-18, 1999, L. Booker, S. Forrest, M. Mitchell, and R. Riolo (eds.). Center for the Study of Complex Systems, The University of Michigan, Ann Arbor, MI.

than random but rarely reaching optimality, difficulty coupling sequential messages, and inaccurate generalization. One result of this uncertain workability is that an LCS is rarely the technique of choice in practical applications.

Yet applicability in at least one area—inference from data—may in fact be great. Data inference is classification, which LCSs are for! A classifier system—inherently non-linear—should be able to find non-linear categories in complex data. And because it consists of discrete rules, an LCS has the virtue, compared with other techniques such as networks and nearest-neighboring, of exhibiting its results with a transparency that permits human comprehension. Finally, compared with other techniques, classifier systems, because they evolve general rules, should be able to learn at a complexity cost proportional to the target concept and not the underlying input space.

XCS (Wilson 1995, 1998, 1999) is a new kind of LCS that reaches optimal performance on quite difficult problems and finds accurate maximal generalizations. Its applicability to data inference tasks is supported by good results with XCS on the Monk's Problems (Saxon and Barry 1999), adaptation of LCSs to risk-of-outcome analysis (Holmes 1998), demonstration of generalized (s-expression) conditions in XCS (Lanzi and Perrucci 1999), and demonstration of noise rejection by XCS (Lanzi and Colombetti 1999). One area that has not received attention, but is important in data inference, is classification of input vectors having real-valued components.

Continuous variables such as temperature, concentration, or age may be decisive in classification, with certain ranges of the variables implying one class and other ranges implying another. LCSs traditionally have taken binary (bitstring) inputs. Sometimes a binary variable is taken to represent a pre-thresholded continuous one, but then the thresholding has not been done adaptively. In a version of XCS taking real inputs—call it "XCSR"—optimally decisive thresholds should be found automatically. Fuzzy classifier systems (Bonarini 1996)—LCSs with rules based on fuzzy logic—take real inputs and generate discrete or continuous outputs. Most fuzzy systems, however, do not adapt their membership functions, though this may, and should, change eventually. Just the same, it is desirable to try to develop XCSR, which if it works will have the advantages of XCS and avoid the addition of fuzzy techniques.

This paper reports two first experiments with XCSR. The results are interesting, and the real domain is found to introduce issues new to LCSs. The next section explains the test problem used, the "real multiplexer". XCSR is introduced in the following section, first by explaining how it differs from XCS, and then briefly reviewing the elements that remain the same. The two experiments are then presented, followed by discussion and conclusion.

2 Test Problem

To test XCSR, we wanted a non-linear classification problem for real vectors. As a simplification, it was assumed that in the input vector $x = (x_0, ..., x_n)$, each component x_i is restricted to the range $0.0 \leq x_i < 1.0$. If in an actual problem the data ranges are known in advance, such scaling implies no loss of generality. If not, XCSR may still be capable of adapting to the actual ranges, but this has not been investigated as yet. We also assumed the vector was to be classified into one of just two classes, 0 or 1, e.g., healthy (0) or diseased (1), as might occur in epidemiological inference. Finally, it was assumed that the class value depended on whether or not certain input variables were in certain ranges, e.g., $(0.12 \leq x_2 < 0.38) \wedge (0.31 \leq x_4 < 1.0) \Rightarrow$ class 1 .

Given these assumptions, our test problem was an adaptation to real values of the relatively well-known Boolean 6-multiplexer problem. This provided a non-linear two-class task of moderate difficulty, and the results could be compared with previous results on the Boolean version.

The Boolean 6-multiplexer function takes as input a six-bit string and outputs a truth value, 1 or 0. One way to get the output is to consider the two leftmost string bits as an unsigned binary number addressing one of the four remaining bits. Thus in the string 011010, the first two bits "01" address bit number 1 of the remaining bits. The value of that bit is 0, which becomes the function value. Alternatively, in disjunctive normal form, the function is given by $F_6 = b'_0 b'_1 b_2 + b'_0 b_1 b_3 + b_0 b'_1 b_4 + b_0 b_1 b_5$, where the subscripts index the bits left to right and primes denote negation.

To define the corresponding "real 6-multiplexer" (RF_6), assume we are given a real vector $x = (x_0, ..., x_5)$ in which $0.0 \leq x_i < 1.0$. Then for thresholds $0.0 \leq \theta_i < 1.0$, interpret x_i as 0 if $x_i < \theta_i$, else 1. The value of RF_6 is then the value of F_6 applied to the bitstring that results from these interpretations. Given random vectors x, XCSR is supposed to learn to return the value of RF_6. XCSR has of course no prior knowledge of F_6 nor of the thresholds, θ_i.

Note that in the earlier broad problem statement, it was assumed that classification would depend on variables being in certain ranges. In RF_6, the dependence is on variables being above or below thresholds, and this is not as general. Testing on ranges—and on negations of ranges, or disconnected ranges—awaits further research. However, as will be seen next, XCSR is at present designed for connected ranges and could presumably be extended.

3 XCSR

3.1 Changes from XCS

XCSR differs from XCS only at the input interface, in its mutation operator, and in the details of covering. The differences arise solely from changing the classifier condition from a string from {0,1,#} to a concatenation of "interval predicates", $int_i = (c_i, s_i)$, where c_i and s_i are reals. To address RF_6, XCSR uses classifiers whose conditions have six interval predicates. A classifier matches an input x if and only if $c_i - s_i \leq x_i < c_i + s_i$, for all x_i. Thus c_i can be thought of as the center value of int_i, and s_i, termed "spread", is a delta defined with respect to c_i. A consequence of using interval predicates is that the number of numerical values or alleles in the condition is twice the number of components in x.

Crossover operates in direct analogy to crossover in XCS. The crossover point can occur between any two alleles, i.e., it can occur within an interval predicate as well as between predicates. Mutation, however, is different. Several preliminary experiments were done, and the best method appears to be to mutate an allele by adding an amount $\pm rand(m)$, where m is 0.1, $rand$ picks a value uniform randomly from $[0, m)$, and the sign is chosen uniform randomly. The result is mutation by a scaled random increment.

"Covering" occurs when no existing classifier matches x. In both XCS and XCSR a new classifier is created which does match. In XCSR the new condition has components $(c_0, s_0, ..., c_5, s_5)$, where $c_i = x_i$ and $s_i = rand(s_0)$, with s_0 a constant such as 0.5. In XCSR, as in recent versions of XCS, a covering classifier (with separately generated condition) is created for each possible action. If the population of classifiers has reached its allowable maximum, deletion of classifiers occurs to make room for the new ones (deletion is generally not necessary; initial populations are empty and covering normally occurs only at the very beginning of a run).

3.2 Common features of XCSR and XCS

The following is an abbreviated description of XCS (see Wilson (1995, 1998) and Kovacs (1997a) for more detail). XCS is designed for both single- and multiple-step tasks, but this description will apply only to XCS for independent single-step tasks in which an input is presented, the system makes a decision, and the environment provides some reward. All material in this section applies equally to XCSR.

Each classifier C_j in the population [P] has three principal parameters: (1) *payoff prediction* p_j, which estimates the payoff the system will receive if C_j matches and its action is chosen by the system; (2) *prediction error* ϵ_j, which estimates the error in p_j with respect to actual payoffs received; and (3) *fitness* F_j, computed as later explained. It is convenient to divide the description of a single operating cycle or time-step into the traditional performance, update (reinforcement), and discovery components.

3.2.1 Performance

Upon presentation of an input, XCS forms a match set [M] of classifiers whose conditions are satisfied by the input. If no classifiers match, covering takes place as described earlier. Then for each action a_k represented in [M], the system computes a fitness-weighted average P_k of the predictions p_j of each classifier in [M] having that action: $P_k = \sum_j F_j p_j / \sum_j F_j$. P_k is termed the *system prediction* for action k.

Next, XCS chooses an action from those represented in [M] and sends it to the environment. According to the action-selection regime in force, the action may be picked randomly or otherwise probabilistically based on the P_k, or it may be picked deterministically—i.e., the action with the highest P_k is chosen. Finally, an action set [A] is formed consisting of the subset of [M] having the chosen action.

3.2.2 Update

In this component, the parameters of the classifiers in [A] are re-estimated according to the reward R returned by the environment as a consequence of the system's taking action a_k. First, the predictions are updated: $p_j \leftarrow p_j + \beta(R - p_j)$. Next, the errors: $\epsilon_j \leftarrow \epsilon_j + \beta(|R - p_j|)$. Third, for each C_j, an *accuracy* κ_j is computed: $\kappa_j = 0.1(\epsilon_j/\epsilon_0)^{-n}$, for $\epsilon_j > \epsilon_0$, else 1.0. Then, from the κ_j, each classifier's *relative accuracy* κ'_j is computed: $\kappa'_j = \kappa_j / \sum_j \kappa_j$. Finally the fitnesses F_j are updated according to: $F_j \leftarrow F_j + \beta(\kappa'_j - F_j)$.

3.2.3 Discovery

On some time-steps, XCS executes a genetic algorithm within [A]. Two classifiers are chosen probabilistically based on their fitnesses and copied. The copies are crossed (two-point crossover) with probability χ, and then mutated with probability μ per allele. The resulting offspring are inserted into [P]; if the population size is already at its maximum value, N, two classifiers are deleted. The probability of deletion of a classifier is determined by Kovacs's (1997b) method and is designed to preferentially remove low-fitness classifiers that have participated in a threshold number of action sets—that is, have had sufficient time for their parameters to be accurately estimated.

Whether or not to execute the GA on a given time-step is determined as follows. The system keeps a count of the number of time-steps since the beginning of a run. Every time a GA occurs, the classifiers in that [A] are "time-stamped" with the current count. Whenever an [A] is formed, the time-stamp values of its members are averaged and subtracted from the current count; if the difference exceeds a threshold θ_{GA}, a GA takes place.

A *macroclassifier* technique is used to speed processing and provide a more perspicuous view of population contents. Whenever a new classifier is generated by the GA (or covering), [P] is scanned to see if there already exists a classifier with the same condition and action. If so, the *numerosity* parameter of the existing classifier is incremented by one, and the new classifier is discarded. If not, the new classifier is inserted into [P]. The resulting population consists entirely of structurally unique classifiers, each with numerosity ≥ 1. If a classifier is chosen for deletion, its numerosity is decremented by 1, unless the result would be 0, in which case the classifier is removed from [P]. All operations in a population of macroclassifiers are carried out as though the population consisted of conventional classifiers; that is, the numerosity is taken into account. In a macroclassifier population, the sum of numerosities equals N, the traditional population size. [P]'s actual size in macroclassifiers, M, is of interest as a measure of the population's space complexity.

4 Experiments

In each of the following experiments, random real vectors, with components $0.0 \leq x_i < 1.0$, were formed and presented to XCSR. Each such presentation was termed a *problem*. The action-selection regime consisted of *explore* problems which occurred with probability 0.5, and *test* problems the rest of the time. On an explore problem, XCSR chose its action uniform randomly from among those in [M], and update and discovery operations were carried out as previously described. Test problems were designed to determine how well the system could do if it chose its action deterministically. The measure of *system performance* was a moving average of the fraction of correct actions (correct values of RF_6) on the previous 50 test problems. Also measured was a similar moving average of $|R - P_k|$ (with P_k the prediction for the chosen action), termed *system error*. On test problems, the update and discovery components were disabled.

In the experiments, $R = 1000$ for the correct action, 0 otherwise. XCSR parameters were as follows: $N = 800$, $\beta = 0.2$, $\epsilon_0 = 10$, $n = 5$, $\theta_{GA} = 12$, $\chi = 0.8$, $\mu = 0.04$, $m = 0.1$, $s_0 = 1.0$. Other values of N, θ_{GA}, χ, μ, m and s_0 were tried in Experiment 1; from this

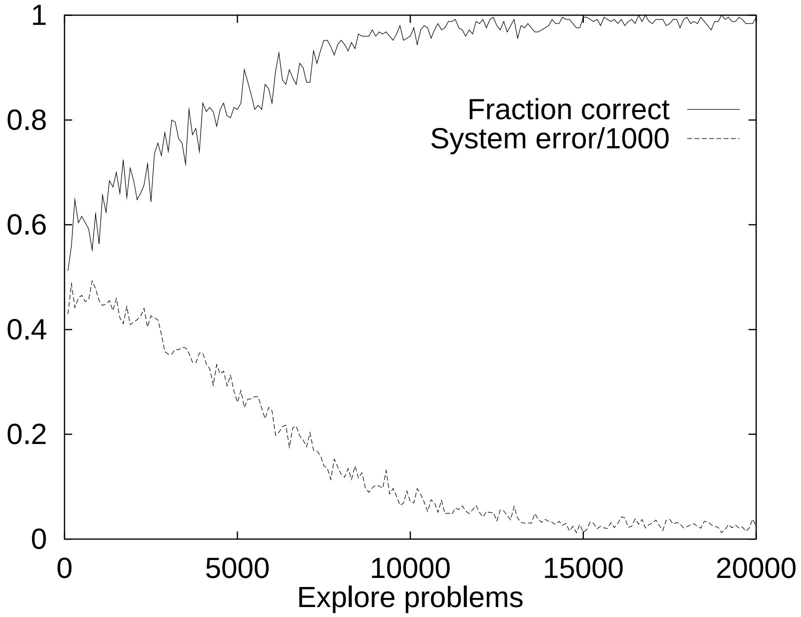

Figure 1: Fraction correct (solid line) and system error/1000 (dashed) vs. number of explore problems in Experiment 1.

limited evaluation, the values given appeared to be best and were used in both experiments.

4.1 Experiment 1

The aim of this experiment was to imitate, as closely as possible, the Boolean 6-multiplexer task, except that the input would be real. In the Boolean task as traditionally conducted, both outcome cases (1 and 0) are equally probable, as are all input bitstrings. An equivalent regime can be obtained in the real case by setting all thresholds θ_i at 0.5, since the underlying real vectors are uniform randomly generated. These were the thresholds in Experiment 1. Figure 1 shows performance and system error averaged over five runs each consisting of 20,000 explore problems. Performance reaches its maximum—approximately 98%—at about 15,000 problems. System error reaches a minimum at a similar point. Compared with typical results on the Boolean 6-multiplexer, arrival at high performance is substantially slower. Recent experiments with XCS on F_6 get to 100% performance in about 1,500 problems, so there is a factor of difference of about ten.

Two sources for the difference come to mind. One is simply the fact that on RF_6, XCSR's classifiers have 12 alleles in their conditions vs. six for XCS on F_6; thus the search space has higher dimensionality. Experiments with XCS on the 11-multiplexer (Wilson 1998)—which requires 11 alleles in the condition—took about 10,000 problems to reach 100% performance. A second possible source of difference is that whereas each Boolean dimension has just two allele values, each dimension in the real case has in principle indefinitely many values.

Of course the effective resolution of a real dimension depends on the quality of performance desired. High performance can only be reached if the system accurately estimates the thresholds, and this will take longer compared with the Boolean case, where the "thresholds"—in effect, choosing between 0 and 1 in the condition—are in a sense maximally

```
                                                                      ACT  PRED ERR   FITN  NUM
 0. |0000000000|.....o0000|0000000000|.....o0000|o000000000|0000000o..| 1    0.  .000   14.   1
 1. |.....o0000|.....o0000|0000000000|0000000000|o000000000|0000o.....| 1    0.  .000   53.   2
 2. |.....o0000|.....o0000|0000000000|.....o0000|0000000000|0000o.....| 1    0.  .000   40.   1
 3. |.....o0000|.....o0000|0000000000|0000000000|o000000000|0000o.....| 1    0.  .000   50.   1
 4. |.....o0000|.....o0000|0000000000|0000000000|0000000000|0000o.....| 1    0.  .000   50.   1
 5. |.....o0000|.....o0000|0000000000|0000000000|0000000000|0000o.....| 1    0.  .000  140.   3
 6. |.....o000o|00000o....|0000000000|0000000000|0000o.....|0000000000| 1   34.  .081    5.   2
 7. |....o00000|.....o0000|0000000000|000000o...|0000000000|0000o.....| 1    0.  .000   56.   2
 8. |.....o0000|.....o0000|0000000000|....o00000|0000000000|0000o.....| 1    0.  .000   41.   1
 9. |....o00000|.....o0000|0000000000|.....o0000|0000000000|0000o.....| 1    0.  .000   58.   1
10. |.....o0000|.....o0000|0000000000|.....o0000|0000000000|0000o.....| 1    0.  .000   46.   1
11. |.....o0000|.....o0000|0000000000|....o00000|0000000000|0000o.....| 1    0.  .000   85.   2
12. |.....o0000|.....o0000|0000000000|.....o0000|0000000000|0000o.....| 1    0.  .000   43.   1
```

```
                                                                ACT  PRED ERR   FITN  NUM
 0. |.572,.985|.924,.393|.322,0.99|.948,.417|.818,.812|.331,.404| 1    0.  .000   14.   1
 1. |.786,.264|0.89,.364|.602,0.99|0.23,.884|.796,.769|.228,.268| 1    0.  .000   53.   2
 2. |.794,.264|0.89,.364|.262,0.99|.868,.344|.665,.769|.228,.268| 1    0.  .000   40.   1
 3. |.794,.264|.807,.262|.602,0.99|0.23,.884|.796,.769|.228,.268| 1    0.  .000   50.   1
 4. |.794,0.28|.807,.262|.684,0.99|0.23,.884|.717,.769|.228,.268| 1    0.  .000   50.   1
 5. |.794,.264|.807,.262|.602,0.99|0.23,.884|.717,.769|.228,.268| 1    0.  .000  140.   3
 6. |.743,.232|.172,.404|.813,.903|0.41,.841|.092,.366|.506,.658| 1   34.  .081    5.   2
 7. |.775,.332|.807,.262|.476,.687|.275,.344|.716,.874|.205,.233| 1    0.  .000   56.   2
 8. |.786,.264|.807,.262|.288,0.99|.818,.322|.717,.783|.181,.269| 1    0.  .000   41.   1
 9. |.798,.357|0.89,.364|.247,0.99|.894,.344|.665,.732|.207,.233| 1    0.  .000   58.   1
10. |.798,.264|0.89,.364|.247,0.99|.894,.344|.665,.732|.207,.269| 1    0.  .000   46.   1
11. |.798,.264|.807,.262|.288,0.99|.818,.322|.717,.783|.207,.269| 1    0.  .000   85.   2
12. |.798,.264|.807,.262|.288, 1.0|.818,.274|.717,.783|.207,.269| 1    0.  .000   43.   1
```

Figure 2: An action set [A] from Experiment 1. Upper half: condition predicates shown graphically (see text). Lower half: actual c and s values of predicates. Also shown: ACTion, PREDiction, ERRor, FITNess, and NUMerosity of each classifier.

coarse. However, the proper analysis of search time vs. resolution is not clear.

In the Boolean case, performance reaches a solid 100%, whereas here it does not, with the level reached depending on the mutation rate and technique. Preliminary experiments included mutation based on $\pm m$ instead of $\pm rand(m)$. This did not produce as good maximum performance, for any mutation rate. The random function, by introducing arbitrarily small mutation increments, probably contributes to a closer ultimate approach to 100%.

It is interesting to examine some classifiers evolved by XCSR. Figure 2 shows an action set. In the first part of the figure, the classifiers are represented using a crude graphic notation. The unit interval is divided into 10 equal parts. If a part contains ".", no value in that part is accepted by the predicate; if it contains "o", some values are; and if it contains "0" all values are accepted. Thus ".....o0000" is an interval predicate accepting inputs greater than some value between 0.5 and 0.6 and less than 1.0. The lower half of the figure shows the same classifiers but with the predicates notated directly with their c and s values. Also included in the figure, following the classifiers, are their action, prediction, error, fitness, and numerosity values. The error and fitness values are scaled. Error is shown as $\epsilon_j/1000$, fitness as $1000F_j$.

Notice how the system has "sculpted" the predicates and is in effect finding the thresholds. Most predicates either show ranges between 0.0 and 0.5, 0.5 and 1.0, or are "don't cares" i.e., "0000000000". In Boolean terms, most of the classifiers could be said to have the form

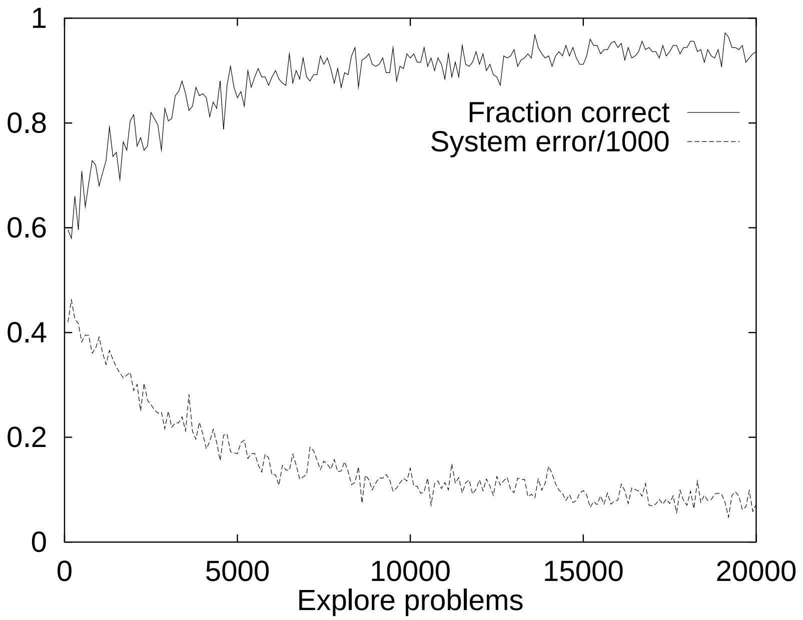

Figure 3: Fraction correct (solid line) and system error/1000 (dashed) vs. number of explore problems in Experiment 2.

11#1#1 : 1 ⇒ 0 or 11###1 : 1 ⇒ 0 (the bit before the arrow is the action, that after the arrow is the prediction). The latter classifier is more general than the former, while still as accurate, and it is likely eventually to drive the former out (see Wilson (1995, 1998) or Kovacs (1997a) for a discussion of accurate generalization).

In the lower half of Figure 2, using the raw c and s values, we can see that the effective thresholds are not quite 0.5, which must account for a good part of the system's residual error. Also interesting is that the number of unique c and s values is relatively small, with the diversity among the classifiers caused mainly by different combinations of those values, i.e., by crossover. While it might be predicted that mutation of real values, especially using $rand(m)$, would make every predicate different in detail, this is not particularly the case.

4.2 Experiment 2

This experiment was like Experiment 1 except that the interpretation thresholds were 0.25, 0.75, 0.25, 0.75, 0.25, and 0.75. In actual data inference problems, the relevant data ranges or thresholds are not only unknown but are in general different from each other. Experiment 2 was designed to model this situation, if somewhat simplistically. The new thresholds do not affect the relative prevalence of the two outcome cases: they are still equally probable, which is generally unlike real-world data sets. In contrast to Experiment 1, however, some inputs in Experiment 2 are much more likely than others. For example, an input whose Boolean interpretation is 010101 has probability $(1/4)^6$, while an input with interpretation 101010 has probability $(3/4)^6$, a ratio of $3^6 = 729$. Thus input frequencies in Experiment 2 were highly variable.

Figure 3 shows the performance and error results. The maximum performance—about 93%—is noticeably lower than in Experiment 1, although it is reached sooner. System error

```
                                                                      ACT PRED   ERR  FITN  NUM
 0. |.o00000000|.......o00|0000000000|0000000000|0000000000|.......o0o| 1 1000.  .000   11.    1
 1. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o0o| 1 1000.  .000   11.    1
 2. |.o00000000|.......o00|0000000000|0000000000|0000000000|.......o0o| 1  667.  .278   19.    2
 3. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   16.    1
 4. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   16.    1
 5. |..o0000000|.......o00|0000000000|o000000000|0000000000|.......o0o| 1 1000.  .000  207.    1
 6. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   26.    1
 7. |o000000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1  738.  .262   19.    1
 8. |o000000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1  654.  .386   11.    2
 9. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   32.    1
10. |o000000000|.......o00|0000000000|0000000000|0000000000|.......o0o| 1  751.  .353    0.    1
11. |..o0000000|........o0|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   31.    1
12. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   33.    1
13. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   37.    1
14. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   33.    1
15. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   33.    1
16. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   29.    1
17. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   99.    3
18. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000  154.    6
19. |..o0000000|.......o00|0000000000|0000000000|0000000000|.......o00| 1 1000.  .000   34.    1
```

Figure 4: An action set [A] from Experiment 2. Same notation as upper half of Figure 2.

is greater than in Experiment 1. At present, we do not understand these differences, although the fact that many input interpretations are much less likely than other interpretations may be responsible. In 20,000 problems, the system sees some interpretations many fewer times than others—in fact, fewer times than any interpretation in Experiment 1.

Figure 4 shows an action set from late in one run of Experiment 2. The Boolean interpretation of the classifiers is 11###1 : 1 ⇒ 1000. It is clear that in most of the classifiers XCSR has "detected" and represented the 0.25 and 0.75 thresholds quite accurately. One can also see that the four classifiers with non-zero error have first predicates that are a little too large.

5 Discussion

Work on XCSR can be taken in several directions. It is easy to suspect that the relative crudeness of the present mutation technique may account for the failure to reach higher performance levels, especially in Experiment 2 where smaller interval predicates were needed. As performance closes in on its maximum, smaller and smaller mutation increments are called for, yet in these experiments there was no such adaptation. It is possible that adapting the increment along lines used in the *Evolutionsstrategie* (Rechenberg 1994) may be appropriate. Adaptation may also increase the rate of learning, because larger mutation increments could presumably be used at the beginning of a run.

The present test tasks—derived from a Boolean problem—while relatively complex and non-linear, are in many respects not representative of actual data inference problems. There, it is often the case that only a few out of many input variables are relevant to the decision, and the challenge is to identify them in the presence of noise and data contradiction. For this, the noise filtering techniques of Lanzi and Colombetti (1999) may be helpful. In addition, the assumption in this paper of equally probable outcomes is often violated by real data, where the prevalences are likely to be skewed. This can be investigated by adapting XCSR (and XCS) to keep the "four-way" statistics characteristic of fields like epidemiology (i.e., $\{true, false\} \times \{positive, negative\}$ instead of machine learning's *correct/incorrect*). Experiments along these lines are under way.

A large step, obviously required if XCSR and XCS are to participate in practical data inference, is to test the systems in regimes where training is done on one data set and testing occurs on another—both sets often being drawn from a single larger set. Such training/testing regimes are standard in data inference. It is vital for XCS-like systems to meet the challenge of training sets that sample the environment incompletely.

6 Conclusion

This paper has demonstrated that XCSR—and thus classifier systems—can learn to classify real-vector inputs and form accurate maximal generalizations over them. The results are another step toward full realization of the promise of Holland's classifier system idea.

7 References

Bonarini, A., (1996). Evolutionary learning of fuzzy rules: competition and cooperation. In *Fuzzy Modelling: Paradigms and Practice*, W. Pedrycz (ed.), 265-284. Norwell, MA: Kluwer Academic Press.

Holland, J. H. (1986). Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning, an artificial intelligence approach. Volume II.* Los Altos, California: Morgan Kaufmann.

Holmes, J. H. (1998). Discovering risk of disease with a learning classifier system. In T. Baeck, ed., *Proceedings of the Seventh International Conference on Genetic Algorithms* (ICGA97). Morgan Kaufmann, San Francisco, CA., 426-433.

Kovacs, T. (1997a). XCS classifier system reliably evolves accurate, complete, and minimal representations for Boolean functions. In Roy, Chawdhry and Pant (Eds), *Soft Computing in Engineering Design and Manufacturing* (WSC2). Springer-Verlag, London.

Kovacs, T. (1997b). Steady state genetic algorithm deletion techniques. Internal Report, School of Computer Science, University of Birmingham.

Lanzi, P. L. and Colombetti, M. (1999). An extension to the XCS classifier system for stochastic environments. In *Proceedings of The 1999 Genetic and Evolutionary Computation Conference* (GECCO-99), W. Banzhaf (ed.)

Lanzi, P. L. and Perrucci, A. (1999). Extending the representation of classifier conditions, part II: from messy coding to s-expressions. In *Proceedings of The 1999 Genetic and Evolutionary Computation Conference* (GECCO-99), W. Banzhaf (ed.)

Rechenberg, I. (1994). *Evolutionsstrategie '94.* Stuttgart-Bad Cannstatt: Frommann-Holzboog.

Saxon, S, and Barry A. (1999). XCS and the Monk's Problem. Second International Workshop on Learning Classifier Systems (IWLCS-99), Orlando, FL, USA, July 13, 1999.

Wilson, S. W. (1994). ZCS: a zeroth level classifier system. *Evolutionary Computation 2(1)*: 1-18.

Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation 3(2)*: 149-175.

Wilson, S. W. (1998). Generalization in the XCS classifier system. In *Proceedings of the Third Annual Genetic Programming Conference*, J. Koza et al (eds.). San Francisco, CA: Morgan Kaufmann, 665-674.

Wilson, S. W. (1999). State of XCS classifier system research. Second International Workshop on Learning Classifier Systems (IWLCS-99), Orlando, FL, USA, July 13, 1999.