Messy Coding in the XCS Classifier System for Sequence Labeling

Masaya Nakata[1], Tim Kovacs[2], and Keiki Takadama[1]

[1] The University of Electro-Communications, Japan
m.nakata@cas.hc.uec.ac.jp, keiki@inf.uec.ac.jp
[2] University of Bristol, UK
kovacs@cs.bris.ac.uk

Abstract. The XCS classifier system for sequence labeling (XCS-SL) is an extension of XCS for sequence labeling, a form of time-series classification where every input has a class label. In XCS-SL a classifier condition consists of some sub-conditions which refer back to previous inputs. Each sub-condition is a memory. A condition has n sub-conditions which represent an interval from the current time t_0 to a previous time t_{-n}. A problem of this representation (called interval coding) is, even if only one input at t_{-n} is needed, the condition must consist of n sub-conditions to refer to it. We introduce a messy coding based condition where each sub-condition messily refers to a single previous time. Unlike the original coding, the set of sub-conditions does not necessarily represent an interval, so it can represent compact conditions. The original XCS-SL evolutionary mechanism cannot be used with messy coding and our main innovation is a novel evolutionary mechanism. Results on a benchmark show that, compared to the original interval coding, messy coding results in a smaller population size and does not require as high a population size limit. However, messy coding requires more training with a high population size limit. On a real world sequence labeling task messy coding evolved a achieved higher accuracy with a smaller population size than the original interval coding.

1 Introduction

Time-series classification has attracted great interests in machine learning. As a kind of time-series classification, *sequence labeling* [3] has been applied in a wide range of real world applications, such as part of speech tagging [8] and recognition of human activity [2]. While typical time-series data is a sequence of values which share a class, sequence labeling data is a sequence of input/class pairs. For example, in a sequence such as speech tagging data, each word in a sentence is one input and is classified as a noun, verb etc. In sequence labeling, the class of the current input may depend on previous inputs, hence a learner may need to refer back to the previous inputs [1]. Our interest is in learning human-

[1] The class of sequence labeling data may also depend on future inputs. However, in many tasks, such as online learning, we want to predict the current or future class strictly from the past, and so we consider only previous inputs.

readable (simple, understandable and compact) solutions from sequence labeling data. Learning Classifier Systems (LCSs) evolve general condition-action rule (called *classifier*). Previously, we introduced XCS for sequence labeling (XCS-SL) [6], an extension of XCS [10]. In XCS-SL a classifier has a *variable-length* condition which consists of sub-conditions $C_0, \cdots, C_{-n}$ as memories that refer back to previous inputs. The condition can *grow* and *shrink* by evolution to find a suitable memory size (i.e., the number of sub-conditions which are needed). This variable-length condition is more useful than a *fixed-length* condition (a fixed memory size), since it can handle a larger memory size than the fixed-length one [6]. Some related LCS works, e.g., XCSM [4] which also uses memory and CCS [9] which uses a kind of variable length condition, have been presented but due to lack of space we must leave comparison of them and XCS-SL for future work. Similarly we must leave comparison with other sequence labeling algorithms for future work.

The variable-length condition C=$\{C_0, C_{-1}, \cdots, C_{-n}\}$ represents an interval of inputs from the current time t_0 to a previous time t_{-n}. However such coding (called *interval coding*) still has a limitation in representing compact conditions. That is, even if only one previous input at t_{-n} is needed to classify the current input at t_0, the condition must consist of all n memories from C_{-1} to C_{-n}. For example, if we only need the memory at time t_{-n} the minimal condition would be C=$\{C_0, C_{-n}\}$ (or even just $\{C_{-n}\}$ if C_0 is not needed to disambiguate the current input). But, the condition in the interval coding contains all memories.

We introduce a *messy-coding* based condition for XCS-SL. In the new condition, each sub-condition *messily* refers to a single different previous time, hence the condition is not necessarily an interval. For instance, a condition can be C=$\{C_0, C_{-5}, C_{-n}\}$. The messy-coding can remove redundant sub-conditions, so it can represent the minimal conditions. Accordingly, the most important thing when evolving classifiers is probably finding *where* and *how many* previous inputs are needed. To do so we present a novel evolutionary mechanism for messy coding in XCS-SL. We test XCS-SL with messy coding on a benchmark problem (the Layered Multiplexer Problem) and a Activity of Daily Living (ADL) recognition problem [7] as a real world application of sequence labeling.

2 Messy Coding in Sequence Labeling

This section describes sequence labeling in more detail by showing example data. We also explain a difference between the messy coding and the original interval coding (i.e., the variable-length condition) in XCS for sequence labeling.

2.1 Sequence Labeling

As shown in Figure 1, the sequence labeling dataset which is a part of a human-activity recognition can be represented as $\langle time, input{:}class \rangle$. The input "*kitchen*" is placed at different time stamps "*1pm*" and "*7pm*" but it has different classes "*lunch*" or "*dinner*" respectively. Note we do not use the time stamps except to

$\langle 9_{am}, \textit{office}{:}\textit{work}\rangle$, $\langle 1_{pm}, \textit{kitchen}{:}\textit{lunch}\rangle$, $\langle 6_{pm}, \textit{living}{:}TV\rangle$, $\langle 7_{pm}, \textit{kitchen}{:}\textit{dinner}\rangle$

Fig. 1. Example dataset of sequence labeling

order the inputs, i.e., a classifier cannot be represented as "IF time is 7_{pm} THEN *dinner*". Hence, the input "*kitchen*" does not unambiguously identify the current class, i.e., the input is *perceptually aliased* [2]. However, when a learner refers back to the previous input, it can successfully classify it when it considers current and previous inputs. For instance, a minimal condition for correctly predicting the "*dinner*" class in Figure 1 can be $\{(\textit{living}, t_{-1})\}$. While a minimal condition should consist of minimum elements, many accurate but not minimal conditions can exist such as the condition $\{(\textit{kitchen}, t_0), (\textit{living}, t_{-1}), (\textit{kitchen}, t_{-2})\}$.

A difficulty of sequence labeling is that a learner does not know *where* and *how many* previous inputs are needed to classify the current input. The learner explores many possible conditions to find minimal conditions, hence it may need many memories to refer back to previous inputs at different time stamps.

2.2 Messy coding vs. Original interval coding

The original interval coding (i.e., the variable-length condition) and the messy coding both are a memory-based approach for classifier conditions. In the original interval coding, the condition consists of sub-conditions as a memory, and includes a sub-condition C_0 for the current input at t_0. This is because, for not-aliasing inputs, classifiers consist of only one sub-condition for the current input. For instance, in Figure 1, classifiers using the original interval coding can be:

cl_1=$\{(\#, t_0), (\textit{living}, t_{-1})$: $\textit{dinner}\}$ $\quad$ cl_2=$\{(\#, t_0), (\textit{living}, t_{-1}), (\#, t_{-2})$: $\textit{dinner}\}$

Here, *don't care symbol* # can be any symbol. While the classifier cl_1 has a minimal condition *in the original interval coding*, cl_2 does not since it includes a redundant sub-condition $(\#, t_{-2})$. However, cl_1 is also *not* minimal in Figure 1 because it has $(\#, t_0)$ and we saw in Figure 1 that only $\{(\textit{living}, t_{-1})\}$ is needed.

In the messy coding, the condition also consists of sub-conditions, but each sub-condition *messily* refers to a different time stamp. Accordingly, unlike the interval coding, the condition using messy coding may have no sub-condition for the current input. For instance, classifiers using the messy coding can be:

cl_3=$\{(\textit{living}, t_{-1}), (\textit{kitchen}, t_{-2})$: $\textit{dinner}\}$ $\quad$ cl_4=$\{(\textit{living}, t_{-1})$: $\textit{dinner}\}$

cl_3 does not have the minimal condition in Figure 1 because of $(\textit{kitchen}, t_{-2})$; cl_4 has the minimal one. So the messy coding can represent more compact conditions than the interval coding. However, the messy coding makes many possible conditions which do not exist in the interval coding, such as cl_3. Hence, an evolution

[2] This work does not use a history of previous classes, since our interest is in online-learning where we do not know if the actions were correct in unlabeled data and so the class history may be unsure information.

of classifiers is important in finding the minimal conditions. We note there are many minimal conditions in the messy coding which may result in many overlapping classifiers. E.g., in Figure 1, a condition $\{(kitchen, t_{-2})\}$ is also the minimal condition for correctly predicting "*dinner*". These overlapping classifiers should increase the population size but it is unclear whether they will otherwise affect the performance of XCS-SL. We do not consider this issue further.

3 XCS-SL Classifier System

This section describes the mechanism of XCS-SL [6]. XCS-SL almost works the same as standard XCS [1] but some mechanisms in the performance and the discovery components are modified. We also explain *subsumption* [1] for the interval coding and the *shrinker*, which can help to find compact conditions.

XCS-SL Classifiers. A classifier in XCS-SL is the same as the standard XCS classifier [1] but it has a new memory size parameter m to determine the number of sub-conditions in its condition C_0, C_{-1}, $\cdots$, C_{-m}. Each sub-condition C_{-n} corresponds to the input at the time stamp t_{-n}. The memory size m is determined and fixed when the classifier is generated but the maximum memory size M for all classifiers is set to a fixed value.

Performance component. The population $[P]$ is initially empty. At the current time t_0, XCS-SL stacks the current input to the *input list*. When the number of inputs in the list is larger than M, XCS-SL deletes the input at the oldest time stamp t_{-M} in the list. Next, XCS-SL builds a *match set* $[M]$ containing the classifiers in $[P]$ whose sub-conditions C_{-n} each match the stacked input at the corresponding time t_{-n}. If $[M]$ does not contain all the possible actions *covering* [1] generates classifiers; their memory size m is set uniform randomly but the maximum value is the number of inputs in the input list (so it does not have more memory than there are past time steps). Each sub-condition C_{-n} is copied from the corresponding input at the time stamp t_{-n} but each element of the sub-condition is replaced by # with a probability $P_{\#}$. From here, XCS-SL works the same as XCS in the performance component (see [1]). After that, the reinforcement component [1] is performed the same way as in XCS.

Discovery component. XCS-SL evolves classifiers using a Genetic Algorithm (GA). In sequence labeling, each input can have its own suitable memory size (i.e., each input may need a different number of previous inputs). Hence, XCS-SL is required to evolve classifiers which have the suitable memory size. Accordingly, XCS-SL builds *subsets* $[A(t_{-n})]$ of the action set which each consists of classifiers in $[A]$ whose memory size m is equal to n. Then XCS-SL selects one *subset* from among the subsets $[A(t_0)]$, $\cdots$, $[A_{(t_{-M})}]$ to perform the GA on. Selection is done by a roulette wheel on the average fitness of each subset. After selection, the GA is applied to classifiers in the selected subset and generates two new offspring with the same memory size as their parents. Evolution finds classifiers with a suitable memory size because classifiers with enough memory have higher fitness than classifiers with too little memory. Classifiers with more memory than they need

also have high fitness, but subsumption removes them. Two offspring are generated as copies of two selected parents and the crossover and mutation operators are applied to the offspring with probabilities χ and μ respectively. In crossover, each sub-condition is recombined with the corresponding sub-condition of the other offspring. The mutation changes elements in each sub-condition, after that it also changes the memory size m of a classifier to a random value with probability μ. If the memory size *shrinks*, the extra sub-conditions C_{-n} $(n > m)$ are removed. If the memory size *grows*, new sub-conditions C_{-n} $(n > m)$ are added which are copies of the corresponding input at the time stamp t_{-n} in the input list and they are generalized as in covering.

Subsumption and Shrinker. Subsumption is a generalization operator that helps to decrease the population size by subsuming a classifier to a more general classifier. In XCS-SL, subsumption applies to classifiers which have different condition lengths from each other. To compare the generality of these classifiers, we assume the shorter classifier has extra virtual maximally general sub-conditions (that have only #) to fit the condition length of the longer classifier. For instance, as shown below, to compare the generalities of the classifiers cl_a and cl_b, we consider that cl_a has two maximally general sub-conditions "###" added. Accordingly, the sub-conditions C_{-1} and C_{-2} of cl_a are more general than the corresponding sub-conditions of cl_b, hence, cl_a is more general than cl_b.

$$\begin{array}{ll} cl_a=\{(1\#0,\ t_0)\} & \rightarrow \{(1\#0,\ t_0),\ (\#\#\#,\ t_{-1}),\ (\#\#\#,\ t_{-2})\} \\ cl_b=\{(1\#0,\ t_0),\ (10\#,\ t_{-1}),\ (11\#,\ t_{-2})\} & \rightarrow \{(1\#0,\ t_0),\ (10\#,\ \ t_{-1}),\ (11\#,\ \ t_{-2})\} \end{array}$$

Shrinker is a compaction operator that helps to find compact conditions; it decreases the memory size of classifiers whose sub-conditions are maximally general. Specifically, if the sub-condition C_{-m} for the oldest time stamp is coded by only #, then C_{-m} is removed and the memory size m is decreased by 1. This process is repeated recursively. For instance, as shown below, the sub-condition C_{-2} of classifier cl_c is removed, since C_{-2} is the maximally general condition "###", and the memory size of cl_c is reduced to 1. Note that the shrinker is not applied to classifiers which consist of only sub-condition C_0. The shrinker is applied to classifiers which are generated by covering and the GA.

$$cl_c=\{(1\#0,\ t_0),\ (\#1\#,\ t_{-1}),\ (\#\#\#,\ t_{-2})\} \rightarrow \{(1\#0,\ t_0),\ (\#1\#,\ t_{-1})\}$$

4 Messy Coding in the XCS-SL Classifier System

This section presents a modified XCS-SL with messy coding (XCS-SL-messy). Normally in LCS, conditions are fixed-length ternary strings from {0, 1, #}. Lanzi [5] introduced messy coding for LCS, in which the # is not represented, and the position of 0s and 1s are explicit. For example, the normal ternary condition {1#0} is equivalent to {(1,0), (0,2)} in messy coding. We use a different kind of messy coding. Lanzi encoded conditions on the single current input messily; he did not use memory. In contrast, we encode memories messily: we do not represent fully general memories (###), but we do represent the time-stamp

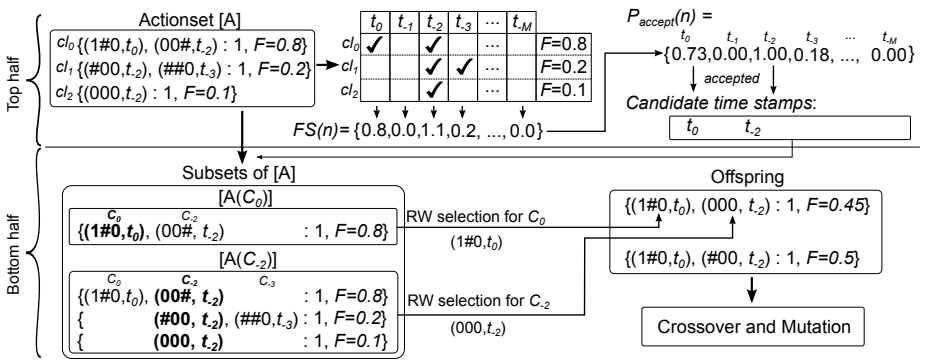


Fig. 2. Discovery component of XCS-SL-messy

of memories which are not fully general e.g., $\{(1\#0, t_0), (11\#, t_{-2})\}$. Note that when a memory is not fully general we use a normal ternary string.

The original XCS-SL evolves a suitable memory size for a classifier but with messy coding we evolve not only how much memory but where (which time steps a classifier refers to). Because we have changed the representation we also have to change the discovery component. This is our most important contribution and we explain it next. We also explain the covering and the shrinker operators which are also modified for the messy coding. Note a classifier for the messy coding is the same as the original XCS-SL except for the condition, which consists of sub-conditions, which each messily refer to a different time stamp.

Covering. When covering takes place, XCS-SL-messy generates classifiers using the messy coding. Firstly, their memory size m is set uniform randomly to determine *how many* sub-conditions are generated. Next, for each sub-condition C_{-n}, a time stamp t_{-n} is set to a random value to determine *where* its sub-condition refers. The time stamp is set except for values which are already assigned in other sub-conditions. The maximum value for the memory size and the time stamp is the number of inputs in the input list.

Discovery component. We introduce a heuristic to estimate how many and where previous inputs are needed to classify the current input to the correct class. Figure 2 shows an overview of the discovery component we introduced. As shown in the top half of Figure 2, we firstly calculate a *fitness summation* $FS(n)$ for each time stamp t_{-n}. Here, we assume a sub-condition of a classifier which has high fitness is a key memory to disambiguate the current input. $FS(n)$ is calculated by Equation (1), which is a summation of fitness of classifiers which have sub-condition C_{-n}. In Equation (1), $cl_k \in [A](C_{-n})$ denotes classifiers cl_k in $[A]$ which their conditions have sub-condition C_{-n}. Next, an *acceptance probability* $P_{accept}(n)$ is calculated by Equation (2), which is the normalized value of the fitness summation. Next, *candidate time stamps* are selected from among all possible time stamps. For each time stamp t_{-n}, we decide either to accept it as a candidate time stamp with the probability of $P_{accept}(n)$ or to reject it.

$$FS(n) = \sum_{cl_k \in [A](C_{-n})} F_k \qquad (1) \qquad P_{accept}(n) = \frac{FS(n)}{max_n FS(n)} \qquad (2)$$

After that, XCS-SL-messy generates offspring based on the candidate time stamps. As shown in the bottom half of Figure 2, like the original XCS-SL it builds *subsets* of the action set, but they are built in a different view point from the original one. Specifically, the subset $[A(C_{-n})]$ consists of classifiers in $[A]$ whose conditions include the sub-condition C_{-n}. Next, the offspring are generated from the classifiers in the subsets. The offspring is given a sub-condition for each candidate time stamp. Firstly, for each candidate time stamp t_{-n}, one parent is selected from the corresponding subset $[A(C_{-n})]$. The sub-condition C_{-n} of the offspring is generated as a copy of C_{-n} of the selected parent. This process repeats two times to generate two offspring. The parameters of offspring are set to averages of the corresponding parameters of their parents. The crossover is the same way as the original XCS-SL. The mutation changes the memory size m of a classifier to a random value with probability μ. If the memory size shrinks, the sub-conditions C_{-n} are randomly selected and removed. If the memory size grows, new sub-conditions C_{-n} are added but their time stamp t_{-n} is randomly selected except for time stamps which are already assigned in other sub-conditions. The C_{-n} are copies of the corresponding input at t_{-n} in the input list which are generalized as in covering.

Shrinker. In XCS-SL-messy, if the sub-condition C_{-n} for *any* time stamp t_{-n} is coded only by #, then its sub-condition is removed and the memory size m is decreased. Note in the original XCS-SL, the shrinker takes place *only* on the sub-condition C_{-m} at the oldest time stamp t_{-m}.

5 Experiment on Benchmark Problem

This section compares XCS-SL and XCS-SL-messy on the Layered Multiplexer Problem [6] as a benchmark sequence labeling task.

In the well-known family of l-bit Boolean multiplexer functions [10], the first k bits are converted to a decimal index into the remaining bits and the value of the string is the value of the indexed bit. E.g., with l-6, the class of 110001 is 1 as the first 2 bits index the final bit. We introduced the n-Layered l-bit Multiplexer Problem (n-l LMP) in [6] as a sequence labeling task. We make a list of D random l-bit binary strings. To train the learner we iterate through them, using one string as input on each time step $t_0, t_1, \cdots, t_D$. In the LMP, the class of the current input may depend on another input. Specifically, the first n bits of the current input are converted to a decimal number as a *reference time* rt. To determine the class of the current input, the LMP refers to the input at t_{-rt} and computes the normal l-bit multiplexer function on it. If the reference time would be negative, i.e., $t_{-rt}<t_0$, we wrap around to the end of the dataset and use t_{D-rt} as the reference input. For instance, on 3-6LMP, for the sequence of inputs $\{\cdots$, 000000, 001000, $\cdots\}$, the correct class of the "000000" is 0 since the class is determined by own input due to rt=0 (and index=0); the correct

class of the "001000" is determined by the previous input "000000" due to rt=1 (which means the class is referred to the last input). We use a reward of 1000 for a correct action, otherwise 0. We use the 0-6 and 3-6 LMP with D=50,000. Note a 0-l LMP is the normal l-bit multiplexer. Note also the minimal condition *in interval coding* is $\{C_0, \cdots, C_{-rt}\}$, but *in the messy coding* it is $\{C_0, C_{-rt}\}$.

5.1 Results

Each experiment consists of a number of problems that the system must solve. In each problem as one iteration, LCS alternatively solves a *learning* problem and an *evaluation* problem (see [10]). We use the standard parameter settings [1]: ϵ_0=1, μ=0.04, $P_\#$=0.33, χ=0.8, β=0.2, α=0.1, δ=0.1, ν=5, θ_{GA}=25, θ_{del}=20, θ_{sub}=20, M=8, and Action set subsumption and GA subsumption are turned on. We use different population size limits N=60,000 and 6,000. The maximum iteration is 2,000,000. The *performance*, which is the rate of correct actions the LCS executed, and *population size*, which is the number of (macro) classifiers [10], are reported as the moving average of 50,000 evaluation problems. All the plots are averages over 30 experiments.

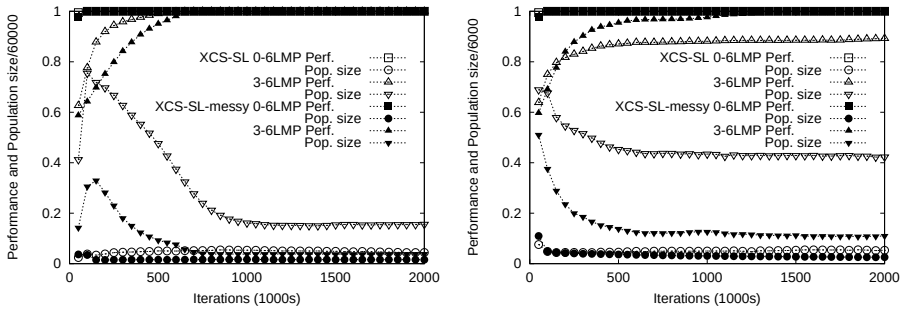
Figure 3 shows the performances and the population sizes of XCS-SL and XCS-SL-messy on {0, 3}-6LMP with N=60,000 and 6,000. From Figure 3 a), with N=60,000, on {0, 3}-6LMP both systems reach 100% performance, but XCS-SL learns faster than XCS-SL-messy. In contrast, from Figure 3 b), with a small population size limit (N=6,000), XCS-SL performs *worse* than XCS-SL-messy: while XCS-SL fails to reach 100% due to the small population size limit, XCS-SL-messy successfully reaches it. While one algorithm outperforms the other depending on the population size limit, for both size limits, XCS-SL has many more classifiers than XCS-SL-messy. Specifically, on 3-6LMP with N=60,000, XCS-SL has 9367 classifiers but XCS-SL-messy has 2291.

In summary, results suggest 1) messy coding has a smaller population size than interval coding, 2) interval coding requires a larger population size limit to reach full accuracy, but 3) messy coding is slower to reach full accuracy when the population size limit is large. It is not clear why messy coding results in a smaller population size, but the smaller population explains observation 2) – because interval coding has a larger population size it needs a larger population size limit to function. We hypothesis that 3) is the case because it takes longer to search the larger space of messy classifiers than the smaller space of interval classifiers. Also, the larger population found with interval coding is searching more of the rule space in parallel than XCS-SL-messy's smaller population.

Results on the layered and regular (i.e., 0-l LMP) multiplexers are similar: the performance of interval coding reaches maximum faster than messy coding, but messy coding has a smaller population size.

6 Experiment on ADL Recognition

This section tests XCS-SL-messy on a real world Activity of Daily Living (ADL) recognition problem, which has the challenge of a small number of instances and



a) N=60,000 b) N=6,000

Fig. 3. Performances and Population sizes on {0, 3}-6LMP

a large number of classes. ADL recognition [7] is a classification task to recognize human activity from binary sensors. We modify the data (OrdonezA) to be a sequence labeling task. The format of each data point is a time/input/class; an input in the form of binary sensor data consists of three elements (*sensor*, *sensor type* and *room*); a class indicates a human activity. The *sensor*, *sensor type* and *room* can be one of 12 sensors, 5 sensor types and 5 rooms respectively; the class can be one of 10 human activities (see [6], [7]). The dataset has 397 data points.

We use the first 70% as training data and the last 30% as test data. Each experiment consists of a *learning* phase and a *test* phase. The test phase happens after the learning phase. During the test phase, the system must solve the test data, and it does not apply the reinforcement and discovery components. We compare XCS, XCS-SL and XCS-SL-messy, and we employ the same parameter settings of the previous test except for $N = 5000$, the maximum iteration is 200,000 and Action Set subsumption was turned off to avoid overly strong generalization pressure. We calculate the *classification accuracy* and the population size during the test phase, which is the average over 30 experiments.

Table 1. a) Classification accuracies (the top half) and p-values (the bottom half) on ADL recognition. b) Population sizes (the top half) and p-values (the bottom half). Bold text indicates a significant difference ($p<0.01$).

a) Classification accuracies

	XCS	XCS-SL	XCS-SL-messy
	0.75	0.86	0.88
XCS	-	**8,15E-07**	**1.33E-07**
XCS-SL	-	-	**9.69E-03**

b) Population sizes

XCS	XCS-SL	XCS-SL-messy
122.4	844.2	769.2
-	**6.19E-36**	**3.82E-33**
-	-	**1.15E-05**

Table 1 shows the classification accuracies and populations sizes of all LCSs and p-values (for classification accuracies and for population sizes) which are calculated using the Two-tailed paired Student t-test. The population size of

XCS is quite smaller than other LCSs but the classification accuracies of XCS-SL and XCS-SL-messy are better than XCS and the positive significant differences for the classification accuracy are noted ($p<0.01$). XCS-SL-messy improves on the classification accuracy of XCS-SL, and the positive significant difference for the classification accuracy between both systems is noted ($p<0.01$). Additionally, XCS-SL-messy had a smaller population size than XCS-SL and the positive significant difference for the population size is noted ($p<0.01$).

7 Conclusion

We introduced XCS-SL with a novel messy coding for memories and a novel evolutionary mechanism to find how many and where previous inputs are needed to disambiguate the current input. On the Layered Multiplexer Problem we found messy coding results in a smaller population size and does not require as high a population size limit. However, messy coding requires more training with a high population size limit than the original interval coding. On a real world sequence labeling task messy coding had higher accuracy and smaller population size than the original interval coding. These results suggest that the messy-coding in XCS-SL, combined with our new evolutionary mechanism can successfully learn accurate and compact conditions. We will evaluate other memory-using LCS on sequence labeling tasks. Finally, we will compare XCS-SL with non-evolutionary sequence labeling algorithms on a range of datasets.

References

1. M. V. Butz and S. W. Wilson. An Algorithmic Description of XCS. *Journal of Soft Computing*, 6(3–4):144–153, 2002.
2. B. Kaluža, V. Mirchevska, E. Dovgan, M. Luštrek, and M. Gams. An Agent-based Approach to Care in Independent Living. In *Ambient Intelligence*, volume 6439 of *LNCS*, pages 177–186. Springer Berlin Heidelberg, 2010.
3. J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *ICML2001*, pages 282–289, 2001.
4. P. L. Lanzi and S. W. Wilson. Toward Optimal Classifier System Performance in Non-Markov Environments. *Evolutionary Computation*, 8(4):393–418, 2000.
5. P. L. Lanzi. Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding. *GECCO-99*, pages 337–344. Morgan Kaufmann, 1999.
6. M. Nakata, T. Kovacs, and K. Takadama. A Modified XCS Classifier System for Sequence Labeling. In *Proc.of GECCO2014*. ACM, 2014, accepted.
7. F. J. Ordóñez, P. de Toledo, and A. Sanchis. Activity Recognition Using Hybrid Generative/Discriminative Models on Home Environments Using Binary Sensors. *Sensors*, 13(5):5460–5477, 2013.
8. Helmut Schmid. Probabilistic Part-of-Speech Tagging Using Decision Trees. In *International Conf. on New Methods in Language Processing*, pages 44–49, 1994.
9. A. Tomlinson, and L. Bull. An Accuracy Based Corporate Classifier System. *Soft Computing*, Springer, 6(3-4):200–215, 2002.
10. S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, June 1995.