# Generalizing Rules by Random Forest-based Learning Classifier Systems for High-Dimensional Data Mining

Firstname Lastname
University
City, State
@@@@@@@@

Firstname Lastname
University
City, State
@@@@@@@@

Firstname Lastname
University
City, State
@@@@@@@@

Firstname Lastname
University
City, State
@@@@@@@@

## ABSTRACT

This paper proposes high-dimensional data mining technique by integrating two data mining methods: Accuracy-based Learning Classifier Systems (XCS) and Random Forests (RF). Concretely, the proposed system integrates RF and XCS: RF generates several numbers of decision trees, and XCS generalizes the rules converted from the decision trees. The convert manner is as follows: (1) the branch node of the decision tree becomes the attribute; (2) if the branch node does not exist, the attribute of that becomes # for XCS; and (3) One decision tree becomes one rule at least. Note that # can become any value in the attribute. From the experiments of Multiplexer problems, we derive that: (i) the good performance of the proposed system; and (ii) RF helps XCS to acquire optimal solutions as knowledge by generating appropriately generalized rules.

## CCS CONCEPTS

• **Computing methodologies → Rule learning**;

## KEYWORDS

Accuracy-based Learning Classifier System, Random Forest, Data mining, High-dimensional data

**ACM Reference Format:**
Firstname Lastname, Firstname Lastname, Firstname Lastname, and Firstname Lastname. 2018. Generalizing Rules by Random Forest-based Learning Classifier Systems for High-Dimensional Data Mining. In *Proceedings of the Genetic and Evolutionary Computation Conference 2018 (GECCO '18)*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/n.n

## 1 INTRODUCTION

Data mining is very required for human society, corporate management, stock price prediction, and self-driving cars. Since these data are high-dimentional, Machine learning-based data mining methods have been proposed. Especially, Deep Learning (DL) [7] is utilized for high-dimentional data mining: data compression by Auto Encoder (AE) [5], and feature extraction by Convolutional Neural Networks (CNNs) [4, 9] are effective. However these techniques cannot derive knowledge indicating reasons why they take the results. The knowledge is important to keep the reliability of the results. To tackle this issue, Matsumoto et al. proposed a hybrid system based on Accuracy-based Learning Classifier System (XCS) and AE [10]. This system utilizes XCS to extract knowledge from the compressed data from AE. The system performs well and acquire the knowledge in Connectionist Bench (Sonar, Mines vs. Rocks) Data Set. However this system might loss data by the hybrid, and might not perform well in data with characteristic factors, *e.g.* multiplexer problem for many bits because XCS extracts important factors from the data, while AE compresses the data to other forms, and each attribute of compressed data has too many kinds of information to extract.

Random Forest (RF) [1] is one of the effective regression and clustering methods. RF is also utilized for high-dimentional data mining. In particular, RF can extract knowledge without changing the data format. However, the knowledge might not be optimal because RF relies on random to generate the decision trees, and specialized for the data. This suggests that the knowledges might not be able to indicate the reasons why the data are acquired. On the other hand, since XCS can acquire whole knowledge to indicate the reasons, the hybrid is effective to acquire the whole knowledge for high-dimentional data with the characteristic factors. Concretely, XCS generalized knowledge enough to express a part of the data, and the ranges which the knowledges express tend not to overlap with each other (*i.e.*, the knowledge is not redundant). In addition, the system might be able to prevent from data loss by the hybrid because RF and XCS utilize the same data format with each other. To acquire all knowledges from high-dimensional data with characteristic factors, this paper proposes the knowledge generation system called RFXCS (Random Forest-based XCS) by combining RF and XCS. Concretely, RFXCS integrates RF and XCS: RF generates several numbers of decision trees, and XCS generalizes the classifiers converted from the decision trees. The convert manner is as follows: (1) the branch node of the decision tree becomes the attribute; (2) if the branch node does not exist, the attribute becomes #; and (3) One decision tree becomes at least one rule.

This paper is organized as follows. We introduce RF and XCS in Sections 2 and 3, and proposes RFXCS in Section 4. In Section

*GECCO '18, July 15–19, 2018, Kyoto, Japan*
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/n.n

5, we introduce multiplexer problem employed for the experiment in this paper, and discuss the result of the experiment in Section 6. We conclude this paper in Section 7.

## 2 RANDOM FOREST (RF)

Random Forest (RF) is one of ensemble learning methods which derive one result by integrating hypotheses from weak learners. The weak learners of RF are called Decision Tree (DT). In this paper, we explain RF as in [6].

### 2.1 Decision Tree (DT)

Decision Tree (DT) is the graph to decide something, and decision tree learning is to generate DT from data. Since RF in this paper is a binary tree, we explain binary DT. Figure 1 shows one DT. DT is composed of one root, and several nodes and leafs in Fig. 1. The node has split function $h(\mathbf{v}, \theta_j)$ to classify data $\mathbf{v}$ to each node or leaf, while the leaf outputs $p(c|\mathbf{v})$. In addition, the number near each node is node number $j$. For example, we input $\mathbf{v} = (x_1, x_2, ..., x_d) \in \mathbb{R}^d$ as $d$-dimensional data to this DT. First, $\mathbf{v}$ is in a root, then classified to right or left node by split function in the root. This is repeated recursively until the data is classified to the leaf. When the data reaches the leaf, the result of the leaf is an output of this DT.

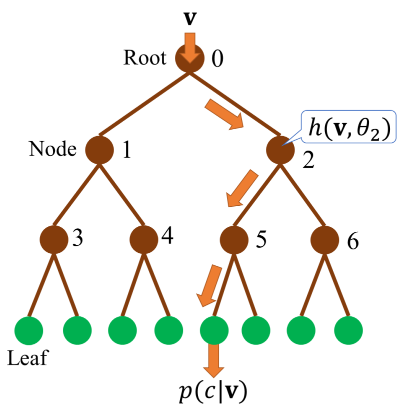


**Figure 1: Decision Tree**

#### 2.1.1 *Split function.*

In a binary tree, the split function is formed as follow.

$$h(\mathbf{v}, \theta_j) \in \{0, 1\}, \tag{1}$$

Where, $j$ is node number, $\mathbf{v}$ is input data in node $j$, and $\theta_j$ is parameter vector in node $j$. Concretely, $\theta$ is composed of $\theta = (\phi, \psi, \tau)$. $\phi$ is the filter extracting several variables (dimensions) from the data $\mathbf{v}$. $\psi$ is the parameter vector to determine the geometric feature of the split function. $\tau$ is threshold vector to split the data $\mathbf{v}$.

#### 2.1.2 *Output from DT.*

DT classifies the data by utilizing already classified data as outputs. In clustering tasks, the output is a posteriori probability $p(c|\mathbf{v})$ of the cluster $c$

### 2.2 Random Forests Learning

Since RF generates a number of DTs, and integrates the outputs of the DTs, Random Forests learning is composed of the learnings by the DTs. In the following sentence, the number of DTs generated by RF is $T$. If the input data set is $S = \{\mathbf{v}\}$, RF samples learning data $S_0 (\in S)$ for each DT, and starts learning the DT by this data. RF selects the split function to separate data appropriately, and repeats this process until filling certain conditions. The current proceeded node becomes leaf node when the conditions have been filled.

#### 2.2.1 *Selection for split function.*

RF selects the optimal split functions for nodes in order from the root node. We consider the node $i$, the learning data $S_i = \{\mathbf{v}\}$ in the node $i$, $S_i^L$ and $S_i^R$ are the data separated to left node and right node from the node $i$, respectively. From the variables, $S_j = S_j^L \cup S_j^R, S_j^L \cap S_j^R = \emptyset, S_j^L = S_{2j+1}, S_j^R = S_{2j+2}$ are satisfied. To learn the split function $h$ is to estimate $\theta$ as the parameter of the split function. In the node $j$, $\theta$ is estimated by the following equation.

$$\theta_j^* = \underset{\theta_j \in \tau_j}{\arg\max}\, I_j \tag{2}$$

In this equation, $I_j$ is the objective function, and is defined as follows.

$$I_j = I(S_j, S_j^L, S_j^R, \theta_j), \tag{3}$$

$I_j$ utilizes entropy and information gain. We consider the situation that the data of $S$ have the class label $c \in C$. In this situation, the entropy and the information gain are calculated as follow.

$$H(S) = -\sum_{c \in C} p(c) log(p(c)), \tag{4}$$

$$I = H(S) - \sum_{i \in 1,2} \frac{|S^i|}{|S|} H(S^i) \tag{5}$$

If the entropy becomes 0, there is only one class in data $S$, and if it becomes maximal, then all classes have the same probability. On the other hand, if the information gain becomes large, the distribution of the class is biased after separating the data. We calculate $\theta_j^*$ by applying $I$ to Eq. 2 as the objective function.

#### 2.2.2 *Learning end condition.*

To prevent from overlearning, we have to apply the learning end conditions. There are several conditions: (1) whether the max length between root and leaf nodes becomes over the threshold; (2) whether the size of the separated data becomes under the threshold; and (3) whether the information gain becomes under the threshold. Generally, all conditions are applied.

#### 2.2.3 *Outputs from RF.*

Figure 2 shows output integration by RF. In RF, we apply input data $\mathbf{v}$ to all DTs, and integrate all outputs from all DTs. We consider $t (\in \{1, ..., T\})$ as identification numbers of DTs. To integrate all outputs, the following equation is utilized.

$$p(c|\mathbf{v}) = \frac{1}{T} \sum_{t=1}^{T} p_t(c|\mathbf{v}), \tag{6}$$

## 3 ACCURACY-BASED LEARNING CLASSIFIER SYSTEM (XCS)

Accuracy-based Learning Classifier System [12] is one of popular LCSs. XCS can extract the valuable knowledge from a lot of data, especially, is good at acquiring the knowledge which represents whole of target problem. In XCS, environment, state, and reward exist. The environment behaves as the target problem with the state and the reward. Concretely, XCS observes the state of the environment (acquires the information of the environment as the state), XCS sends an answer called an action to the environment, and the environment sends the reward corresponding to the state and the action to XCS in each iteration. XCS has multiple classifiers as the knowledge. A classifier indicates one rule and information acquired through learning. Essentially, XCS send an action to the environment by following a rule of the certain classifier, and evaluates the classifier based on the reward acquired by the sending to learn. Repeating these processes, XCS acquires the general classifiers which have an appropriate action for the environment as the target problem.

### 3.1 Classifier

Concretely, a classifier consists of a condition, an action, and main parameters as follow [12].

- **Condition** $C$
  $C$ defines the state which this classifier can match. $C$ is a string consisting of {0, 1, #}. Note that # is called "do not care", and matches both 0 and 1 in the input.
- **Action** $A$
  $A$ is the action which this classifier proposes.
- **Prediction** $p$
  $p$ is the average of the rewards which XCS has received by selecting this classifier before.
- **Prediction error** $\varepsilon$
  $\varepsilon$ is the average among the differences between current $p$ and previous $p$ calculated whenever $p$ is calculated.
- **Experience** $exp$
  $exp$ is how many times this classifier is carried into the action set [A], *i.e.*, this classifier is updated.
- **Time stamp** $ts$
  $ts$ is the number of the step spent until now since this classifier is last proceeded by GA.
- **Fitness** $f$
  $f$ is the value from 0 to 1 as the fitness of the classifier in the genetic algorithm.
- **Action set size** $as$
  $as$ is the average of the sizes of the action set [A] in every learning
- **Numerosity** $num$
  $num$ is how many classifiers this classifier has subsumed until now.

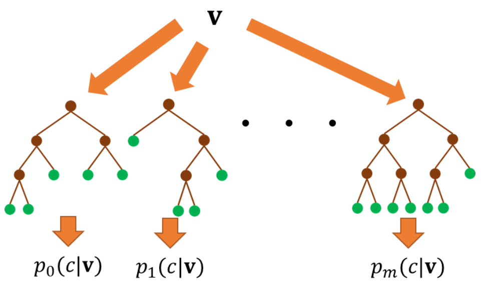
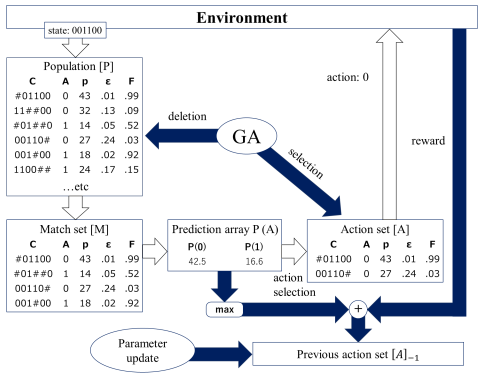


**Figure 2: Output integration by RF**



Population [P]

| C | A | p | ε | F |
|---|---|---|---|---|
| #01100 | 0 | 43 | .01 | .99 |
| 11##00 | 0 | 32 | .13 | .09 |
| #01##0 | 1 | 14 | .05 | .52 |
| 00110# | 0 | 27 | .24 | .03 |
| 001#00 | 1 | 18 | .02 | .92 |
| 1100## | 1 | 24 | .17 | .15 |

…etc

Match set [M]

| C | A | p | ε | F |
|---|---|---|---|---|
| #01100 | 0 | 43 | .01 | .99 |
| #01##0 | 1 | 14 | .05 | .52 |
| 00110# | 0 | 27 | .24 | .03 |
| 001#00 | 1 | 18 | .02 | .92 |

Prediction array P (A)

| P(0) | P(1) |
|---|---|
| 42.5 | 16.6 |

Action set [A]

| C | A | p | ε | F |
|---|---|---|---|---|
| #01100 | 0 | 43 | .01 | .99 |
| 00110# | 0 | 27 | .24 | .03 |

**Figure 3: XCS**

### 3.2 Mechanisms

Figure 3 shows the behavior of XCS. XCS is composed of three components: performance component (explained in 3.2.1), reinforcement component (explained in 3.2.2), and discovery component (explained in 3.2.3). XCS repeats these three components in rotation to learn.

*3.2.1 Performance Component.* The classifiers that matched the state of the environment are selected from the population [P], and their action is sent to the environment in Performance Component. The state is a binary string, and XCS carries the classifiers that match the state of the environment into the match set [M] by comparing the state with the condition $C$ of the classifier in the population [P]. In this process, if the number of the classifiers in the match set is less the threshold $\theta_{nma}$, XCS repeats generation of new classifiers by a process called covering until the number of the classifiers in the match set in over $\theta_{nma}$. A classifier generated by covering initially has the environment state as its condition $C$ but XCS exchanges each bit of the condition in the classifier to # with the probability $P_{\#}$.

To select an action, XCS first calculates the value of the prediction array $P(A)$ for each action by following Equation 9. XCS then selects an action using the prediction array to define the selection probabilities. In Eq. 7, $[M](A)$ is the set of the classifiers whose action is $A$. If not classifiers in [M] have a certain action, the prediction of the action becomes *nill* in order to never be selected. In learning mode XCS selects an action at random and in exploit mode it selects the action with the highest prediction. The classifiers in [M] with the selected action are, carried into the action set [A], the action is sent to the environment, and the reward $r$ is received from the environment. This sequential process is called a step in this paper.

$$P(A) = \sum_{cl_k \in [M](A)} cl_k.p \times \frac{cl_k.f}{\sum_{cl_k \in [M](A)} cl_k.f} \tag{7}$$

*3.2.2 Reinforcement Component.* After the performance component process is finished, XCS updates the parameters of the classifiers in [A]. XCS calculates the prediction $p$, the prediction error $\epsilon$, and the action set size $as$ by following Equations (8)-(11).

$$P \leftarrow r + \gamma \max P(A) \tag{8}$$

$$cl.p \leftarrow cl.p + \beta(P - cl.p) \tag{9}$$

$$cl.\epsilon \leftarrow cl.\epsilon + \beta(|P - cl.p| - cl.\epsilon) \tag{10}$$

$$cl.as \leftarrow cl.as + \beta\left(\sum_{c \in [A]} c.num - cl.as\right) \tag{11}$$

Where $P$ is the target value to update the prediction $p$, while $\beta(0 \leq \beta \leq 1)$ and $\gamma(0 \leq \gamma \leq 1)$ are parameters called learning rate and discount factor, respectively. $\beta$ controls the learning speed, and $\gamma$ indicates how much XCS utilizes the reward $r$ for the prediction $p$[11]. Next, XCS calculate the accuracy of the classifier $\kappa(0 \leq \kappa \leq 1)$ to estimate the classifier's fitness $f$ by following Equations (12) and (13).

$$cl.\kappa = \begin{cases} 1 & if\ cl.\epsilon \leq \epsilon_0 \\ \alpha\left(\frac{cl.\epsilon}{\epsilon_0}\right)^{-\nu} & otherwise \end{cases} \tag{12}$$

$$cl.f \leftarrow cl.f + \beta\left(cl.\kappa \cdot \frac{cl.num}{\sum_{c \in [A]}(c.\kappa \cdot c.num)} - cl.f\right) \tag{13}$$

Where $\alpha, \nu, \epsilon_0$ is parameters of XCS. $\epsilon_0$ indicates the range how much XCS allows the prediction error $\epsilon$. If the $\epsilon$ is less than $\epsilon_0$, $\kappa$ becomes 1; otherwise, $\kappa$ becomes smaller as a function of $\epsilon$.

*3.2.3 Discovery Component.* In discovery component, XCS generates and deletes classifiers based on a Genetic Algorithm (GA). Concretely, if the average among $ts$ of the classifiers in [A] is over the parameter $\theta_{GA}$, XCS executes this component. In the process of GA, XCS selects the classifiers as parents based on their fitnesses by Roulette wheel selection [3], and crossover the condition $C$ of the classifiers to generate new classifiers as children. The classifiers' parameters $p, \epsilon, f$ are the average between the parameters of both parents. XCS changes each bit of both the condition $C$ and the action $A$ of each child with the probability $\mu$: if each bit of $C$ is 0 or 1, it becomes #, and if it is #, it becomes the corresponding bit of the state. If the action is changed the new action is chosen uniform randomly. If the parents cannot subsume a child it is carried into [P]. As the result, if the number of the classifiers in [P] is over the parameter $N$, XCS repeats deleting a classifier until the number of the classifiers in [P] becomes $N$. Concretely, XCS selects the classifier to delete by following the equation below as the selection probability, and decreases the classifier's numerosity *num* by 1.

$$cl.deletion\ vote = cl.as \times cl.num \tag{14}$$

## 4 PROPOSED SYSTEM

For high-dimensional data mining, we propose Random Forest and Accuracy-Based Learning Converted Classifier System (RFXCS). Figure 4 shows the detail of RFXCS. From this figure, RFXCS generates several DTs by RF, converts the DTs to the classifiers, and generalizes the classifiers by XCS. We explain these three processes as follows.

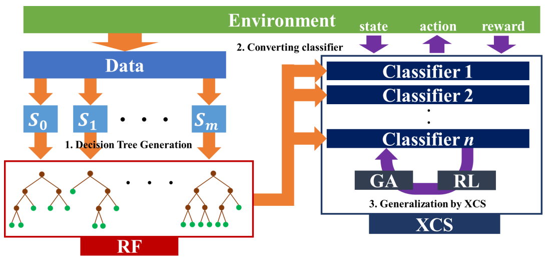


**Figure 4: RFXCS**

### 4.1 Decision Tree Generation

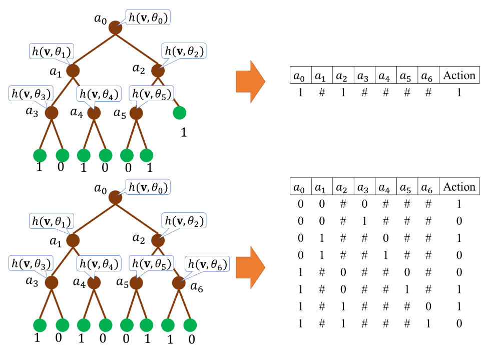
RFXCS randomly samples a subset of data called $S_i$. This process is repeated $m$ times to generate $m$ data sets (*i.e.*, $S_0 - S_{m-1}$). Note that the data sets can overlap with each other. After that, RFXCS generates a DT from each data set by RF, and stores i number of DTs in total.

### 4.2 Converting Classifier

Figure 5 shows how to convert a DT to a classifier. In this figure, the left side is the DT, while the right side is the classifiers converted from the DT. $a_j$ $(j = 0, 1, \ldots)$ is the attribute corresponding to the split function $h(\mathbf{v}, \theta_j)$ when the DT is converted. The number 0 or 1 near each leaf indicates output number. RFXCS generates classifiers from the shortest route in the DT. In the upper half of Figure 5, since the route $\{0, 2, leaf\}$ is the shortest of all, RFXCS generates a classifier 1#1###1, while, the lower half, since all routes have the same shortest length, RFXCS generates 8 classifiers corresponding to the leafs. The generation criteria are that (1) the attribute corresponding to each node in the route becomes value of the split function in each node, (2) the attribute corresponding to each node outside of the route becomes #, and (3) the action of the classifier becomes value of the leaf in the route.

### 4.3 Generalization by XCS

RFXCS converts the classifiers from the DTs as initial classifiers in population [P], and generalizes those. RFXCS selects from the classifiers to make an initial population, and changes the initial parameters of the classifiers. Concretely, there are 6 versions of the above process as shown in Table 1. In this table, the left column indicates the name of each version, while the right column indicates the classifiers used in each version. The right column indicates the initial parameters for correct and incorrect classifiers, respectively. In Boolean functions like the multiplexer (section 5) a correct classifier is one which only receives the higher of the two possible


$a_0$
$h(\mathbf{v}, \theta_0)$
$h(\mathbf{v}, \theta_1)$
$a_1$
$h(\mathbf{v}, \theta_2)$
$a_2$
$h(\mathbf{v}, \theta_3)$
$a_3$
$h(\mathbf{v}, \theta_4)$
$a_4$
$h(\mathbf{v}, \theta_5)$
$a_5$
1
1 0 1 0 0 1
$a_0$
$h(\mathbf{v}, \theta_0)$
$h(\mathbf{v}, \theta_1)$
$a_1$
$h(\mathbf{v}, \theta_2)$
$a_2$
$h(\mathbf{v}, \theta_3)$
$a_3$
$h(\mathbf{v}, \theta_4)$
$a_4$
$h(\mathbf{v}, \theta_5)$
$a_5$
$h(\mathbf{v}, \theta_6)$
$a_6$
1 0 1 0 0 1 1 0

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | Action |
|---|---|---|---|---|---|---|---|
| 1 | # | 1 | # | # | # | # | 1 |

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | Action |
|---|---|---|---|---|---|---|---|
| 0 | 0 | # | 0 | # | # | # | 1 |
| 0 | 0 | # | 1 | # | # | # | 0 |
| 0 | 1 | # | # | 0 | # | # | 1 |
| 0 | 1 | # | # | 1 | # | # | 0 |
| 1 | # | 0 | # | # | 0 | # | 0 |
| 1 | # | 0 | # | # | 1 | # | 1 |
| 1 | # | 1 | # | # | # | 0 | 1 |
| 1 | # | 1 | # | # | # | 1 | 0 |

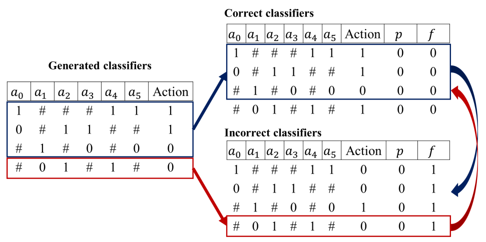
**Figure 5: Converting Classifier from DT**

rewards while an incorrect classifier receives only the lower of the two possible rewards. Versions Pa and Na put p to 0/1000 in the positive/negative classifier to make it incorrect. Versions Pb, Pc, Nb, and Nc put p to 1000/0 in the positive/negative classifier. In addition, the versions Pb and Nb, and Pc and Nc put $f$ to 0 and 1, respectively. If $f$ is 1, XCS learns the classifier as the classifier, otherwise, it learns the classifier as the inappropriate classifier. Note that RFXCS deletes the classifier same as the other classifiers in this process. For example, in 6-multiplexer problem, RFXCS generates

**Table 1: Each version of RFXCS**

| version | converted classifiers |
|---|---|
| Pa | positive ($p = 0, f = 0$) |
| Pb | positive ($p = 1000, f = 0$) |
| Pc | positive ($p = 1000, f = 1$) |
| Na | positive ($p = 0, f = 0$) + negative ($p = 1000, f = 0$) |
| Nb | positive ($p = 0, f = 0$) + negative ($p = 0, f = 0$) |
| Nc | positive ($p = 0, f = 0$) + negative ($p = 0, f = 1$) |

the classifiers shown in the left side of Figure 6. There are three correct classifiers and one incorrect classifier in this figure. RFXCS generates new classifiers by reversing the action of each classifier in Fig. 6 (*i.e.*, RFXCS generates the correct classifiers based on the incorrect classifiers, and vice versa.) After that, RFXCS puts certain value to $p$ and $f$ in all classifiers (RFXCS with version Nc puts 0 to $p$ and $f$ in the correct classifiers, and puts 0 and 1 to those in the incorrect classifiers in Fig. 6.)


Generated classifiers
Correct classifiers
Incorrect classifiers

Generated classifiers

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | Action |
|---|---|---|---|---|---|---|
| 1 | # | # | # | 1 | 1 | 1 |
| 0 | # | 1 | 1 | # | # | 1 |
| # | 1 | # | 0 | # | 0 | 0 |
| # | 0 | 1 | # | 1 | # | 0 |

Correct classifiers

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | Action | $p$ | $f$ |
|---|---|---|---|---|---|---|---|---|
| 1 | # | # | # | 1 | 1 | 1 | 0 | 0 |
| 0 | # | 1 | 1 | # | # | 1 | 0 | 0 |
| # | 1 | # | 0 | # | 0 | 0 | 0 | 0 |
| # | 0 | 1 | # | 1 | # | 1 | 0 | 0 |

Incorrect classifiers

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | Action | $p$ | $f$ |
|---|---|---|---|---|---|---|---|---|
| 1 | # | # | # | 1 | 1 | 0 | 0 | 1 |
| 0 | # | 1 | 1 | # | # | 0 | 0 | 1 |
| # | 1 | # | 0 | # | 0 | 1 | 0 | 1 |
| # | 0 | 1 | # | 1 | # | 0 | 0 | 1 |

**Figure 6: Initial classifiers (Version Nc)**

## 5 MULTIPLEXER PROBLEM

Multiplexer problems are often employed to validate XCS generalizing performance. In this problem, XCS receives a bitstring as input, outputs an action, and receives a high reward for the correct action and a low reward for the incorrect action. The multiplexer problem has several conditions below:

- The action is the number 0 or 1.
- The length of the input string is $k + 2^k$ ($k$ is a constant that indicates one of a set of multiplexer functions)
- The correct action is the value of bit $d+k+1$ ($d$ is the decimal value of the first $k$ bits)

"011101:1" is an example where $k = 2$. The length of the input string is 6 and the 7th bit shows the correct action. In this example, $d$ is 1, so the correct action is the value of bit $d + k + 1 = 4$.

## 6 EXPERIMENT

### 6.1 Experimental setup

To validate the effectiveness of RFXCS, we apply it to two multiplexer problems, 6 and 37-multiplexer problems. Concretely, we compose RFXCS with XCS in the 6 and 37-multiplexer problems. Since the version N of proposed mechanism performs better than the version P in the 6-multiplexer problem, we apply only version N to the 37-multiplexer problem.

### 6.2 Evaluation criteria

In the experiment, we evaluate the performance of the proposed system based on three criteria below. Figures show only exploit iterations [2].

(1) **Performance**
Performance indicates how often RFXCS selected the correct action in the recent iterations. The higher accuracy is better.
(2) **Population size**
Population size indicates the number of the classifiers belonging to RFXCS. The smaller population size is better.
(3) **Optimal solutions**
Each length of multiplexer problem has an optimal solution consisting of maximally general, accurate, non-overlapping classifiers. This set is called [O]. The 6-multiplexer has 16 optimal classifiers and the 37-multiplexer has 128. This criterion measures how many optimal classifiers are in the population [P].

### 6.3 Parameters used

The general parameters are determined by following [2]: concretely, $\nu = 5, \alpha = 0.1, \beta = 0.2, \epsilon_0 = 1, \theta_{mna} = 2, \theta_{GA} = 25, \chi = 0.8, \mu = 0.04$. Table 2 shows parameters which are changed in each problem. In this table, $N$ indicates the maximum number of the classifiers RFXCS and XCS can store. $P_{\#}$ is the probability to add # in new classifier when it generates that. $S$ is the number of data for learning by RF. There are 30 trials with 30 seeds in each problem. Explore + exploit iterations are 10000 and 2000000 in 6 and 37-multiplexer problems, respectively.

**Table 2: Parameters**

| | 6-Multiplexer | 37-Multiplexer |
|---|---|---|
| $N$ | 400 | 5000 |
| $P_{\#}$ | 0.33 | 0.65 |
| $S$ | 40 | 10000 |
| iteration | 10000 | 2000000 |

## 6.4 Results

Figures 7, and 8 show the performance, the population size, and the number of the optimal classifiers in the 6 and 37-multiplexer problems, respectively. The upper side is the results in the 6-multiplexer problem, while the lower side is the results in the 37-multiplexer problem. Note that the results are averaged from results of 30 trials in these figures.

*6.4.1 Results in 6-multiplexer problem.* From Fig. 7a, the result of each version of RFXCS converges to 100% performance. From Fig. 7b, each version of RFXCS can solve the 6-multiplexer problem with smaller population size than that required by XCS. In Fig. 7c, RFXCS discovered all 16 optimal classifiers, unlike XCS, although XCS will consistently discover all 16 optimal classifiers for this problem given more iterations [8].) From these results, we conclude the version N performs better than the version P in RFXCS as noted above.

*6.4.2 Results in 37-multiplexer problem.* From Fig. 8a, the result of each version of RFXCS converges to 100% performance. From Fig. 8b, each version of RFXCS can solve the 37-multiplexer problem with a smaller population size than that required by XCS. In fig. 8c, RFXCS discovered all 128 optimal classifiers, unlike XCS, although XCS will discover all 128 optimal classifiers given enough iterations.

## 6.5 Discussion

Since RFXCS can perform well in both multiplexer problems, these results suggest that the proposed system can solve higher dimensional multiplexer problems. In addition, RF has an important role for RFXCS performing faster than XCS (*i.e.*, RFXCS needs fewer iterations). We discuss the roles of XCS for RFXCS in the below.

*6.5.1 Rule generated by RF.* Table 3 shows the performance of each version of RFXCS in first learning by XCS and the 37-multiplexer problem. In this table, the performance of the version Na is 16.1%, while those of the versions Nb and Nc are 83.1%. That is because the version Na has the only classifiers have $p$ which does not indicate its true value of them, unlike the other versions. From this table, XCS can discriminate between correct and incorrect classifiers converted from DTs. On the other hands, the performances of the versions Nb and Nc are 83.1% not being 100%. This indicates that the classifiers converted DTs cannot derive the best performance without XCS.

*6.5.2 Optimal solutions.* RF can acquire the correct solutions, but it cannot always acquire the optimal classifiers [O]. Table 4 shows the consistently correct half of [O] for the 6-multiplexer. (Each classifier in Table 4 has a complementary classifier which has the same condition bu the other action and which is consistently incorrect. These classifiers are the other half of [O] [8, 12].) Table 5 shows another set of classifiers that are consisitently correct and optimally general for the 6-multiplexer. Comparing the classifiers between Tables 4 and 5, the numbers of # in the classifiers are the same. The classifiers in Table 5 are different from the classifiers in [O] because of overlaps (redundancy). No two classifiers in [O] match the same input: they are non-overlapping (irredundant). In contrast, the classifiers in Table 5 overlap with the classifiers in [O] and some of them overlap with each other *e.g.*, both 1###111 and #1#1#11 can match the input 1111111. The classifiers in [O] cover the complete input/action space, so the classifiers in Table 4 are not necessary. In contrast, the classifiers in Table 4 do not cover the complete input space, that is, they do not form a complete map [8, 12]. For these reasons, XCS research has focused on learning [O] and not the classifiers in Table 4. See ([8], chapter 4) for a discussion of the value of overlapping rules and a discussion of how to evaluate the success of genetic search in learning the multiplexer and other problems.

**Table 3: Performance in first learning by XCS**

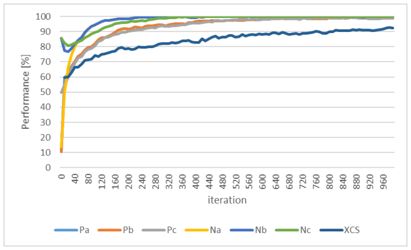
| | Performance [%] |
|---|---|
| Na | 16.1 |
| Nb | 83.1 |
| Nc | 83.1 |

**Table 4: Consistently correct classifiers in the optimal set [O] for the 6-multiplexer**

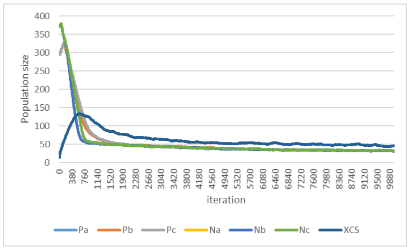
| condition | | | | | | action |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | # | # | # | 1 |
| 0 | 0 | 0 | # | # | # | 0 |
| 0 | 1 | # | 1 | # | # | 1 |
| 0 | 1 | # | 0 | # | # | 0 |
| 1 | 0 | # | # | 1 | # | 1 |
| 1 | 0 | # | # | 0 | # | 0 |
| 1 | 1 | # | # | # | 1 | 1 |
| 1 | 1 | # | # | # | 0 | 0 |

**Table 5: Consistently correct and optimally general classifiers for the 6-multiplexer that are not in the optimal set [O]**

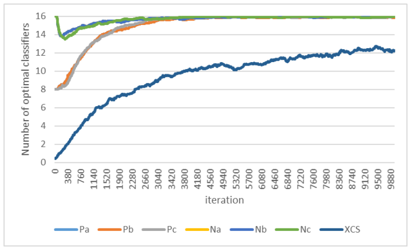
| condition | | | | | | action |
|---|---|---|---|---|---|---|
| 1 | # | # | # | 1 | 1 | 1 |
| 1 | # | # | # | 0 | 0 | 0 |
| 0 | # | 1 | 1 | # | # | 1 |
| 0 | # | 0 | 0 | # | # | 0 |
| # | 1 | # | 1 | # | 1 | 1 |
| # | 1 | # | 0 | # | 0 | 0 |
| # | 0 | 1 | # | 1 | # | 1 |
| # | 0 | 0 | # | 0 | # | 0 |



**(a) Performance**

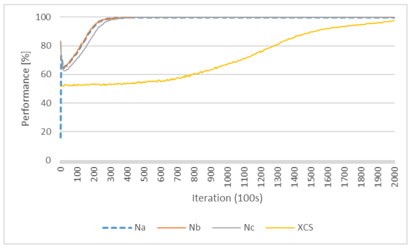


**(b) Population size**

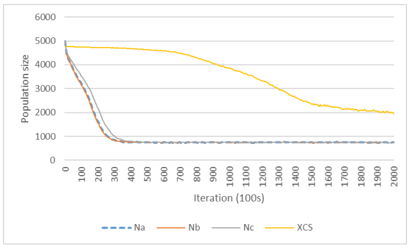


**(c) Number of optimal classifiers**

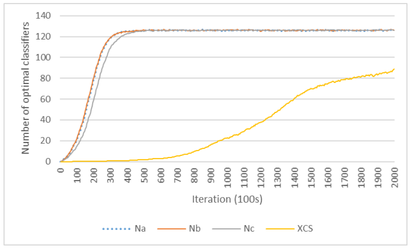
**Figure 7: Result in 6-multiplexer problem**



**(a) Performance**



**(b) Population size**



**(c) Number of optimal classifiers**

**Figure 8: Result in 37-multiplexer problem**

## 6.6 Result of 135-multiplexer problem

Figure 9 shows the performance of RFXCS in 135-multiplexer problem. Vertical and horizontal axes indicate the performance and iteration, respectively. From this figure, though the performance becomes low along to the iterations, the performance is 92% in 53000 iteration, and 80% averagely. The reason for the performance becoming low is that XCS deletes the better classifiers which might become optimal solutions. Since the classifier whose address bit has # must not optimal solutions in multiplexer problem, XCS cannot utilize this classifier and has to delete this. The multiplexer problem with many bits becomes very difficult because XCS has to change almost all bits while must not change address bits to #. Generally, XCS has many classifiers to decrease the probability the better classifiers are deleted in 135-multiplexer problem. However, if there are many classifiers, each classifier is not utilized more times, and cannot be evaluated accurately. At the result, the better classifier might be deleted. This suggests that the performance of Figure 9 might become high. Table 6 shows the detail of the classifiers in 0, 53000, and 519000 iterations. Each column shows the iteration number, max number of # in each classifier (right number is the number of # in the optimal rule), and the number of suboptimal classifiers from left in order. In this problem, RFXCS cannot acquire any optimal classifiers, but the suboptimal classifiers. The suboptimal classifiers are that if several numbers of reference bits become #, the classifier becomes optimal classifiers. There are 512 kinds of optimal rules in this problem, RFXCS has the suboptimal rules corresponding to all optimal rules. From this table, RFXCS can acquire the many suboptimal rules and the more optimally suboptimal rules than before as the iteration goes on. Table 7 shows the classifiers including the suboptimal rules in 0, 53000, and 519000 iterations. Each column indicates the address bits, the reference bit indicated by the address bits, the action, the number of #, the prediction $p$, the fitness $f$, the numerosity *num*, the experience *exp* from left in order. The upper, middle, and lower sides indicate the classifiers in 519000, 53000, and 0 iterations, respectively. In this table, if the output is equally to action, the prediction of the classifier is 1000 as the reward, otherwise, that is 0. From this result, RFXCS has the appropriate classifiers in each iteration.

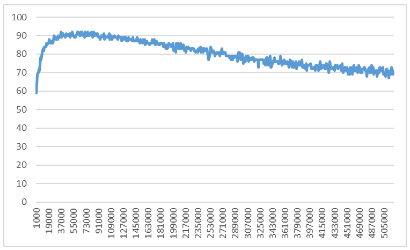


**Figure 9: Performance in 135-multiplexer problem**

**Table 6: Detail of classifiers in 0, 53000, and 519000 iterations**

| iteration | max number of # | number of suboptimal solutions |
|---|---|---|
| 519000 | 126/127 | 13340 |
| 53000 | 125/127 | 12271 |
| 0 | 125/127 | 6254 |

If the numerosity and the experience is high, the fitness of the classifier is high. In addition, the numerosity and the experience become large, and the fitness becomes large as the iteration goes on. This suggests that the classifiers including many other classifiers and evaluated many times have accurate fitness and do not become deleted. From this result, RFXCS can evaluate the classifiers appropriately in 135-multiplexer problem. Therefore, RFXCS might have all optimal classifiers in more iterations.

**Table 7: Suboptimal classifiers in 0, 53000, and 519000 iterations**

| 519000 iteration | | | | | | | |
|---|---|---|---|---|---|---|---|
| address | output | action | # | *p* | *f* | *num* | *exp* |
| 1010101 | 0 | 0 | 125 | 1000 | 0.62 | 67 | 243 |
| 1110011 | 0 | 0 | 125 | 1000 | 0.98 | 117 | 276 |
| 0101010 | 0 | 0 | 125 | 1000 | 0.85 | 81 | 249 |
| 1010101 | 0 | 1 | 125 | 0 | 0.99 | 109 | 238 |
| 1110011 | 0 | 1 | 125 | 0 | 0.94 | 100 | 265 |
| 0101010 | 0 | 1 | 125 | 0 | 0.94 | 78 | 276 |
| 0001100 | 1 | 0 | 125 | 0 | 0.88 | 93 | 212 |
| 1010101 | 0 | 0 | 125 | 1000 | 0.64 | 71 | 181 |
| 0110110 | 0 | 1 | 125 | 0 | 0.99 | 117 | 193 |
| 0110011 | 0 | 0 | 126 | 1000 | 0.93 | 135 | 262 |
| 1011010 | 1 | 1 | 125 | 1000 | 0.78 | 73 | 145 |
| 0011001 | 1 | 1 | 125 | 1000 | 0.98 | 109 | 148 |
| 0000000 | 0 | 1 | 125 | 0 | 0.86 | 93 | 113 |
| 0110011 | 0 | 1 | 125 | 0 | 0.74 | 90 | 89 |
| 0001100 | 1 | 0 | 125 | 0 | 0.71 | 50 | 65 |
| 1011010 | 1 | 1 | 125 | 1000 | 0.29 | 20 | 60 |
| 1011010 | 1 | 1 | 125 | 1000 | 0.52 | 32 | 61 |
| 0110011 | 0 | 0 | 126 | 1000 | 0.87 | 75 | 94 |
| 0010011 | 1 | 1 | 125 | 1000 | 0.78 | 143 | 30 |
| 0000000 | 0 | 1 | 125 | 0 | 0.38 | 19 | 40 |
| 0000000 | 0 | 1 | 125 | 0 | 0.22 | 6 | 27 |
| 0110011 | 0 | 1 | 125 | 0 | 0.17 | 3 | 25 |

| 53000 iteration | | | | | | | |
|---|---|---|---|---|---|---|---|
| address | output | action | # | *p* | *f* | *num* | *exp* |
| 1010101 | 0 | 0 | 125 | 1000 | 0.47 | 24 | 24 |
| 1110011 | 0 | 0 | 125 | 1000 | 0.55 | 33 | 26 |
| 0101010 | 0 | 0 | 125 | 1000 | 0.72 | 37 | 29 |
| 1010101 | 0 | 1 | 125 | 0 | 0.35 | 4 | 21 |
| 1110011 | 0 | 1 | 125 | 0 | 0.96 | 103 | 46 |
| 0101010 | 0 | 1 | 125 | 0 | 0.46 | 25 | 24 |

| 0 iteration | | | | | | | |
|---|---|---|---|---|---|---|---|
| address | output | action | # | *p* | *f* | *num* | *exp* |
| 1010101 | 0 | 0 | 125 | 0 | 0.01 | 1 | 0 |
| 1110011 | 0 | 0 | 125 | 0 | 0.01 | 1 | 0 |
| 0101010 | 0 | 0 | 125 | 0 | 0.01 | 1 | 0 |
| 1010101 | 0 | 1 | 125 | 0 | 0.01 | 1 | 0 |
| 1110011 | 0 | 1 | 125 | 0 | 0.01 | 1 | 0 |
| 0101010 | 0 | 1 | 125 | 0 | 0.01 | 1 | 0 |

## 7 CONCLUSIONS

This paper proposes a data mining technique called RFXCS for high-dimensional data mining. Concretely, RFXCS integrates RF and XCS: RF generates several numbers of decision trees, and XCS generalizes the classifiers converted from the decision trees. The convert manner is as follows: (1) the branch node of the decision tree becomes the attribute; (2) if the branch node does not exist, the attribute of that becomes #; and (3) One decision tree becomes at least one rule. There are 6 versions of RFXCS. The versions of RFXCS have the different type of classifiers with each other: the versions Pa, Pb, and Pc have positive examples with ($p = 0, F = 0$), ($p = 1000, F = 0$), and ($p = 1000, F = 1$), respectively. The versions Na, Nb, and Nc have positive examples with ($p = 0, F = 0$) and negative examples with ($p = 1000, F = 0$), ($p = 0, F = 0$), and ($p = 0, F = 1$), respectively. From the experiments of the 6 and 37-multiplexer problems, we derives (i) the faster performance of RFXCS than XCS; and (ii) RFXCS is good at high-dimensional data mining because XCS helps RFXCS to optimize the generalized knowledge obtained by RF from high-dimensional data.

In the future, we are going to cope the following things in order to improve RFXCS. One is that we utilize the shortest route of a DT to generate the classifier in this paper. Since the classifier might be overgeneralized by this way, we have to determine the length of a route of a DT to generate that classifier. The other is that RFXCS converts DTs to classifiers in terms of condition and action in the classifiers. Since the performance of XCS changes along parameters of initialized classifiers, we have to derive the technique to extract the parameters from the DTs.

## REFERENCES

[1] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[2] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. 2004. Toward a theory of generalization and learning in XCS. *IEEE Transactions on Evolutionary Computation* 8, 1 (Feb 2004), 28–46. https://doi.org/10.1109/TEVC.2003.818194

[3] M. V. Butz and S. W. Wilson. 2002. An algorithmic description of XCS. *Soft Computing* 6, 3 (01 Jun 2002), 144–153. https://doi.org/10.1007/s005000100111

[4] Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36, 4 (01 Apr 1980), 193–202. https://doi.org/10.1007/BF00344251

[5] Paul Munro Garrison W. Cottrell. 1988. Principal Components Analysis Of Images Via Back Propagation. (1988), 1001 - 1001 - 8 pages. https://doi.org/10.1117/12.969060

[6] H. Habe. 2012. *Random Forests.* Technical Report 31. Department of Informatics, School of Science and Engineering, Kinki University.

[7] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.* 18, 7 (July 2006), 1527–1554. https://doi.org/10.1162/neco.2006.18.7.1527

[8] Tim Kovacs. 2004. *Strength or Accuracy: Credit Assignment in Learning Classifier Systems.*

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[10] Kazuma Matsumoto, Takato Tatsumi, Hiroyuki Sato, Tim Kovacs, and Keiki Takadama. 2017. XCSR Learning from Compressed Data Acquired by Deep Neural Network. 21 (09 2017), 856–867.

[11] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[12] Stewart W Wilson. 1995. Classifier fitness based on accuracy. *Evolutionary computation* 3, 2 (1995), 149–175.