Adaptive Artificial Datasets Through Learning Classifier Systems for Classification Tasks

Syahaneim Marzukhi
Faculty of Engineering
Victoria University of Wellington (VUW), Wellington, New Zealand
marzuksyah@myvuw.ac.nz

Will N. Browne
Faculty of Engineering
Victoria University of Wellington (VUW), Wellington, New Zealand
will.browne@ecs.vuw.ac.nz

Mengjie Zhang
Faculty of Engineering
Victoria University of Wellington (VUW), Wellington, New Zealand
mengjie.zhang@ecs.vuw.ac.nz

ABSTRACT

In existing artificial classification systems, the problem domain is created and controlled by humans. Humans set up and tune the problem domain, such as determining the problem's complexity. If humans can set up the problem appropriately then the machines can extract beneficial knowledge to solve classification task. This paper introduces an autonomous classification-problem generation approach. The classification problem's difficulty is adapted based on the classification agent's performance within the defined attributes. An automated problem generator has been created to evolve the simulated datasets whilst the classification agent, in this case a learning classifier system, attempts to learn the evolving problem. The idea here is to tune the datasets autonomously such that the problem characteristics may be determined efficiently to empirically test the learning bounds of the classification agent by lowering human involvement. In this way, the effect of the problem's characteristics, which alter the classification agent's performance, becomes human readable. Tabu Search has been applied in the problem generator to discover the best combination of domain features in order to adjust the problem's complexity. Experiments confirm that the problem generator was able to tune the problem's complexity either to make the problem 'harder' or 'easier' so that it can either 'increase' or 'decrease' the classification agent's performance.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics-Based Machine Learning, Learning Classifier Systems

General Terms

Algorithm, Performance

Keywords

Pattern classification, Learning Classifier Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'13 Companion, July 6-10, 2013, Amsterdam, The Netherlands.
Copyright 2013 ACM 978-1-4503-1964-5/13/07 ...$10.00.

1. INTRODUCTION

The challenge of an automated classification system is to develop a computer based program that learns to identify whether an object belongs to a specific class. Usually a classification system's performance is assessed on different sets of data, commonly from public repositories. There is another approach to investigating a classification system's capability for a classification task, which uses a synthetic dataset [15]. This approach helps researchers to analyze the system (algorithm) under a controlled scenario as the datasets can be generated according to a particular property.

In existing artificial classification systems, the problem domain is created and controlled by humans. Humans set-up and tune the problem domain, such as determining the problem's complexity. If the problem is too complex the system does not learn; conversely, if the problem is too simple the system does not reach its full potential to be able to classify environmental examples. If humans can set-up the problem appropriately then the machines can extract beneficial knowledge to solve the classification task. Automatic classification-problem generation is an important problem that could help in empirically testing the learning bounds of machine learning systems in general and learning classifier systems specifically.

In [9], a novel framework has been implemented for *Two-Cornered Learning Classifier Systems (LCSs)* for creating various pattern classification problems where it can be used to generate artificial datasets (e.g. a library of problems). Both of the problem domain and the solution evolved autonomously (i.e. the pattern generation agent created the problems and the associated sets of patterns, while the pattern classification agent learnt each set of the patterns). The pattern generation agent evolved autonomously to create various complex problems with different levels of difficulty for pattern classification based on its prediction of the pattern classification agent's learning ability.

The ultimate aim of the previous work [9] was to enable the pattern generation agent to autonomously tune (i.e. increase or decrease the problem's difficulty) based on the classifier's learning performance. The system has been tested on non-sparse problems with a low number of conditions and sparse problems with a higher number of conditions. Findings showed that the generated image-based patterns were not well separable; one pattern may be labeled to more than one class, which leads to data ambiguity and class imbalance. It has been discovered that there was no underlying relation between the resulting pattern and the features in the prob-

lem in order to distinguish the class clearly. This was due to the problem formulation not being intrinsically separable and the set of features not being sufficient to describe the understanding concept.

The original focus of that work [9] was on generating varied data, but the underlying feature relationships were not easily separated. The pattern generation agent randomly generated image-based patterns without having any mechanism that could control certain features in the problem, such as data sparsity, noise and class balance. Therefore, the interpretation of the pattern created for each problem and mapped to corresponding difficulty levels was not clear. Moreover, a small number of certain patterns were rare, which introduced sparsity to the problems. This caused the system to wrongly generalize all such 'problems' as 'difficult'. Much more samples would have resulted in the patterns being trivial to learn but this would have been time consuming. Instead it is now considered better to generate the feature relations directly rather than at a level of abstraction.

The main goal of the work is to design a new classification problem generation methodology that utilizes *Tabu Search* technique to autonomously generate synthetic datasets for classification with different levels of complexity based on the classification agent's ability to learn. More specifically, we are concerned with the autonomous problem domain creation by the *problem generator* to generate datasets (i.e. continuous attributes) for data mining tasks. The first objective is for the problem generator to determine the problem's characteristics (i.e. class balance, noise, decision boundary, number of instances), in terms of either 'increasing' or 'decreasing' the problem's complexity. The second objective is for the problem generator to autonomously make the problem either 'harder' or 'easier' for the classification agent to learn.

The problem of defining a problem with suitable complexity for the classification agent is considered a *meta-problem*. The meta-problem is difficult in practice due to the time taken for any individual problem. Starting from the beginning for each problem or repeating a given problem is time consuming. Thus a local search method is required to adjust the problem domain so the classification agent can commence from the previous learnt solutions, without requiring to repeat the same problem domain.

Tabu Search (TS) is a neighborhood search method that has a *memory* (recency-based memory) that allows escaping from sub-optimal solutions by improving the efficiency of the exploration process. The memory forces TS to explore a new area of the search space in order to overcome local-optimal problem. Whenever a new candidate solution is introduced (i.e. a new best solution is found), it goes in the memory and it is made forbidden for a certain number of iterations. TS is used to search the best combination of features in the problem (i.e. with the objective either to 'increase' or 'decrease' the classification agent's performance) by varying the difficulty level (i.e. either to make the problem 'easier' or 'harder' to learn).

Relative change in performance is observed, whereas 'easy' and 'hard' measures of the problem's difficulty required arbitrary judgment [9]. It is hypothesized, that this method will help the problem generator to : 1) define the problems with the appropriate levels of complexity, 2) find types of problems that are commensurate in the domain of competence of the classification agent, and 3) identify adversarial problems to which the pattern classification agent's performance can still be improved.

The remainder of this paper is organized as follows. Section 2 presents the necessary background of pattern classification, problem complexity, LCSs and Tabu Search (TS) technique. Section 3 describes the framework design, the synthetic datasets generation, the knowledge representation and the classification process. Sections 4 and 5 present results of the experiments and discuss the outcomes. Section 6 provides the conclusions and the direction for future work.

2. BACKGROUND

2.1 Pattern Classification

The field of pattern recognition is concerned with the automatic discovery of regularities in data through the use of computer algorithms for further actions [1]. An example of pattern recognition is *classification*, which is the task of assigning one of several predefined categories to each object.

Given a *pattern* as a pair of variables: $[x, \omega]$ [2], where x is a vector of observation, and ω is a label that represents a meaningful concept for the problem domain. Classification is the task of learning a target function $f = [x, \omega]$ that maps each attribute set x to one of the predefined class labels ω. The target function can serve as an explanatory tool to distinguish between objects of different classes (*descriptive modeling*), or can serve as a predictive tool to predict the class label of unknown records (*predictive modeling*).

2.2 Problem Complexity

A *problem* can be difficult for different reasons [3], which can affect the performance of classifiers. In [3], Ho and Basu identified that problems can be difficult because of a mixture of three effects: (1) class ambiguity, (2) boundary complexity, and (3) sample sparsity and feature space dimensionality. In [15], Macia defined these three effects as follows.

Ambiguity is referred to a situation when there are examples in which their features do not allow to distinguish the class. Usually, this ambiguity is due to the problem formulation in which the concepts are intrinsically inseparable or the set of attributes is not sufficient to describe the concepts. Class separability and problem linearity are based on the geometrical complexity of data structure. The classes are ambiguous regardless of sample size or feature space dimensionality.

Boundary complexity is related to the description of the class boundary. Complex decision boundaries and the subclass structure can be categorized by the minimum length of a computer program needed to reproduce it. Boundary complexity is also due to the nature of the problem regardless of sample size or feature space dimensionality.

Sample sparsity and feature space dimensionality are concerned with the difficulty when the sampled instances of a problem do not contain all of the necessary patterns. Moreover, small sample size and high dimensionality are likely to increase this difficulty, making the solution more complex to discover. Therefore, the rules may overgeneralize if they do not encounter examples near the decision boundary. In contrast, simple problems are normally linearly separable with wide margins between decision boundaries.

This set of descriptors that characterize different aspects of complexity are useful to estimate the classification sys-

tem's performance, as well as to investigate the classification system's domains of competence.

2.3 Learning Classifier Systems (LCSs)

LCSs [17] are a subset of Genetics-based Machine Learning (GBML) systems, which are machine learning techniques that incorporate reinforcement learning (RL) and evolutionary computation (eg. Genetic Algorithm (GA)) in its main component. The RL component works on the classifier's prediction to evaluate the classifiers for the identification of best rules, while the GA component is responsible for discovering potentially better rules[5]. The desired outcome of running an LCS is for the classifiers to collectively model an intelligent decision maker [10].

In LCSs, knowledge is commonly represented as a set of *condition-action* rules called classifiers, where the rules are usually in the form of "IF <condition> THEN <action>". The *condition* specifies the state which the rule matches, and the *action* specifies the selected action to be sent to the environment when the condition is satisfied. The *condition* is commonly fixed length *ternary* strings from $\{0, 1, \#\}$, while the *action* and *environmental state* are fixed length *binary* strings from $\{0, 1\}$ [8]. Typical LCSs learn by interacting with the environment.

Accuracy-based LCSs was introduced by Stewart Wilson [12] in 1995, called XCS. The success factors of XCS are: 1)the fitness is based on the accuracy of the prediction and 2)the niche-based nature of the GA. The classifier's fitness is based on the accuracy with which it predicts the reward received from its interaction with the environment [8], rather than the prediction (reward) itself. The GA acts in environmental niches instead of on the whole population to maintain the parallel sustenance of equally important sub-solutions. This means that the GA searches for rules (classifiers) that are accurate in their prediction, independently from their prediction value for selecting rules [11].

There are three important classifier groupings in the XCS: the population set [P], the match set [M] and the action set [A]. The *population* of rules consists a number of rules after the input domain has been covered. The *match set* [M] is formed from the current population set [P] such that it includes all classifiers that match the current input. The *action set* [A] is formed from the current match set [M] such that it includes all classifiers from match set [M] that propose the executed action to the environment. The interaction process in the XCS's performance components is illustrated in Figure 1, followed by the basic LCSs algorithm shown in Algorithm 1.

Accuracy-based Learning Classifier Systems with real-value (XCSR), which was introduced later by Wilson [13], enhanced XCS to real inputs (i.e. integer and real-valued problem domains). The changes from XCS to XCSR for integer and real-valued inputs were as follows [13]. The classifier condition C that specifies a problem domain is encoded within the interval $C = (l_1, u_1, l_2, u_2, ..., l_n, u_n)$, where l_n is a *lower bound* and u_n is an *upper bound* of a real-values. The action part A specifies an available action, that is, $A \in a$, where $a = (a_1, ..., a_m)$ is the set of all possible actions in the problem [4]. A classifier matches an input message x if each element x_n belongs to the corresponding interval in C within the lower bound and upper bound range, i.e. $l_n \leq x_n \leq u_n$. When a new covered classifier is created, each interval $int_n = (l_n, u_n)$ is generated as $l_n = x_n - rand(r_0)$ and $u_n = x_n + rand(r_0)$, where $rand(r_0)$ is a value uniform randomly from $[0, r_0]$ and r_0 is a real constant [6]. GA component and the rest of the system works as described in [6, 13].

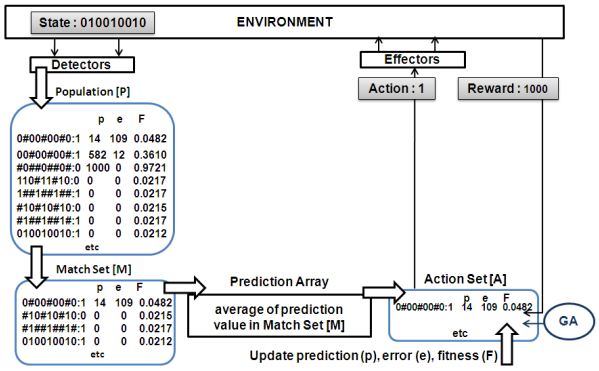

Figure 1: Schematic XCS, adapted from [12].

Algorithm 1: Algorithmic description of XCS's (Performance Component), adapted from [8]

```
1  begin
2      Perceive a single input string (e.g. current state of the
       problem) from the environment.
3      Generate a random population of classifiers [P]. Build a
       match set [M] containing all the classifiers in the population
       [P], where the condition matches the input string.
4      if ([M] is empty or some of the classes are not predicted
       in [M]) then
5          Covering process is activated, a new classifier is created
           (a condition is a generalized version of the input
           example, an action is the class that not covered in [M].
           Add this classifier in the population.
6      end
7      Calculate the prediction values for each action in the match
       set [M] based on the p and f values of each classifier in [M].
8      Evaluate each action in the match set [M] based on the
       calculated prediction value.
9      Form the action set [A] containing the classifiers in match
       set [M] that will advocate action to the environment.
10     Send the selected action to the environment and receive a
       reward.
11     Activate the credit assignment algorithm (i.e.
       Q-Learning-like for classifiers updation).
12 end
```

2.4 Two-Cornered LCSs

In the *Two-Cornered LCSs* [9], the system consists of two main agents, the pattern generation agent (i.e. the Sender(S)) and the pattern classification agent (i.e. the Receiver (R)) (Figure 2). The pattern classification agent is developed based upon *Accuracy-based LCSs with real-value encoding (XCSR)*, an applicable agent model that provides a framework for investigating learning in agents and practical applications [4, 13].

In this setup, the goal is that S generates synthetic datasets from various problems for classification that will need to be solved by R. Therefore, S activates a program for synthetic datasets generation, while R activates a program for data classification. R maintains its own population of candidate classifiers and evolves using evolutionary computation. R learns the classification problems and adapts to different feedback returns from S. Based on R's performance, S will identify the affect of varying feature values in the problem

and generate a new problem for classification at the appropriate levels of difficulty for R to learn. If S's objective is to increase R's performance, S will tune and make the problem 'easier' to learn. If S's objective is to decrease R's performance, S will make the problem 'harder' to learn. Each new problem for classification will be generated to maximize (or minimize) R's performance.

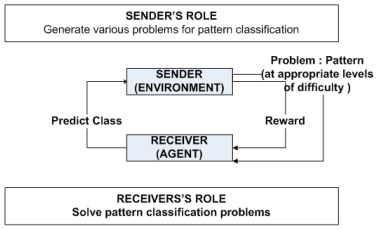

Figure 2: Problem generation and classification process.

2.5 Tabu Search

Tabu Search (TS) is a neighborhood search method proposed by Glover in 1986 as an improvement over basic local search techniques in order to overcome local search problems by not being stuck in a local optimum [18]. A *memory* forces TS to explore a new area of the search space. A known number of solutions that have been examined recently become *tabu* (forbidden) points that cannot be selected when searching for the next solution, they are stored in the memory called the *tabu list* (recency-based memory) [16].

The basic principle of TS is to pursue local search whenever it encounters a local optimum by allowing non-improving moves as cycling back to previously visited better solutions in the tabu list is forbidden [19]. The *Tabu list* records the recent history of the search, essentially the value of the objective function $f(i^*)$ of the best solution i^* and also keeps information on the itinerary through the last solutions visited [19]. Thus, the use of the tabu list allows escaping from sub-optimal solutions by improving the efficiency of the exploration process. Whenever a new candidate solution is introduced (i.e. a new best solution is found), it goes in the tabu list and it is made tabu for a certain number of iterations. For each iteration any non-zero values in the tabu list are decremented by one. It can be revisited again only when value of the candidate solution in the tabu list is 0.

In this work, instead of maintaining the whole solutions, TS may consider individual features to be searched, i.e. combination of features that can cause the agent's classification performance either to increase or decrease. The simplest approach for TS maintaining the tabu list is by remembering an *index* of features that have been swapped and the *time* when it was swapped [16]. Figure 3 illustrates the implementation of tabu list with length five. In Figure 3 (top) the swap at *index 3* gave the best result, thus the feature at *index 3* is forbidden to be used and not available for the next five iterations. In Figure 3 (bottom), *index 2*, *5* and *8* are available to be swapped any time, however *index 1* is not available to be swapped for the next three iterations, and so on. The most recent swap is at *index 4*. TS performs as follows (Algorithm 2).

Index	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
counter	0	0	5	0	0	0	0	0

The contents of tabu list after iteration 1.

Index	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
counter	3	0	1	5	0	4	2	0

The contents of tabu list after iteration 5.

Figure 3: Example of performing Tabu List (recency-based memory), adapted from [16].

Algorithm 2: Algorithmic description of Tabu Search adapted from [19].

```
1  begin
2      s ← s0 : create an initial solution.
3      sBest ← s.
4      TabuList ← null.
5      while (not stopping condition) do
6          Find the best neighbor of the current solution by
           applying an allowed move (non-tabu move).
7          if (a given criteria is meet) then
8              sCandidate ← : accept as the new current solution.
9          end
10         else
11             sCandidate ← : find another neighbor (best
               non-tabu neighbor).
12         end
13         if (fitness(sCandidate)>fitness(sBest)) then
14             sBest ← sCandidate.
15             TabuList ← featureDifferences(sCandidate,
               sBest).
16             while (size(TabuList)> maximum TabuList size)
               do
17                 ExpireFeatures(TabuList).
18             end
19         end
20     end
21     return sBest : globally best solution found.
22 end
```

3. METHODS

A method to generate different types of problem with different features in order to create various synthetic datasets for classification will be presented. The use of TS in the pattern generation agent to search for the best combination of features in the problem to prevent many repeated datasets being generated is described.

3.1 Synthetic Datasets Generation

The pattern generation agent will generate datasets from various problems with different sets of features. The problem of defining an appropriate problem domain is considered a *meta-problem*. The *meta-problem* can be described by a list of the features containing `<Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd>`, where each feature can take a number of values as described in Table 1. Each *instance* in the datasets is created *on-the-fly* based on the specified problem (see example in Table 2). `F` is used to distinguish a feature of the meta-problem and `f` relates to a feature in the dataset of an individual problem. Note that `Fn` determines the number of features `f` in the dataset. Once these parameters are set, the dataset containing a number of instances will be generated and the *class* of each instance is labeled according to the specified meta-problem that has been defined. In addition, we implement *Tabu Search* technique in S to find an optimal combination of `F` in the meta-problem that can either maximize or minimize R's performance. An algorithmic description of this method is shown in Algorithm 3, which de-

scribes the main task of S and R for problem generation and classification task (restricted to two class problems here).

Algorithm 3: Algorithmic description for problem generation and classification, Subscript R Receiver.

```
1  begin
2      problem ← Sender : generate initial problem to Receiver.
3      while (problem less than maximum problems) do
4          while (instance less than maximum instance in
           dataset) do
5              instance ← Sender : generate instance based on
               problem.
6              Receiver ← pattern : perceive instance from
               Sender.
7              GENERATE MATCH SET [M]_R out of [P]_R using
               instance.
8              GENERATE PREDICTION ARRAY PA_R out of
               [M]_R.
9              class ← SELECT ACTION according to PA_R .
10             GENERATE ACTION SET [A]_R out of [M]_R
               according to class.
11             Receiver : execute action class.
12             reward ← Sender : Sender check class and send
               reward back to Receiver.
13             prediction ← reward : update prediction with
               current reward.
14             UPDATE SET [A]_R using prediction possibly
               deletion in[P]_R.
15             RUN GA in [A]_R considering instance insertion in
               [P]_R.
16             if instance equal to maximum instance in dataset
               then
17                 classificationPerformance : calculate
                   Receiver classification performance.
18             end
19         end
20         Sender : read classificationPerformance
21         problem ← Sender : APPLY TS on problem based on
           classificationPerformance.
22     end
23 end
```

3.2 Knowledge Representation

S initializes a random meta-problem containing a list of parameters (i.e `<Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd>`) for a synthetic dataset generation. The dataset consists of a set of i instances, where each instance is defined by `Fn` features as in Table 1. Each feature f in i is created *on-the-fly* based on the specified problem within the interval of $[0, 1]$ and is labeled accordingly. Using Tabu Search technique, S searches for the best combination of features `F` for the set task. Based on R's classification performance, S changes the values of features `F` (i.e. finds the best combination of features that can either maximize or minimize R's classification performance) for generating the next problem for R to learn (refer to Table 3).

Table 4 illustrates the R's *condition-action* rule. The *condition* was encoded to real value of $real_n = (l_n, u_n)$, where l_n is the lower bound and u_n is the upper bound within the interval $[0, 1]$. The *action* can be either '1' for 'Class 1', otherwise '0' for 'Class 0'. R will receive a reward of '1000' for correct classification or '0' for incorrect classification.

3.3 Pattern Generation and Classification

A detailed example of process of pattern generation and classification between S and R follows. First, S creates variants of the meta-problem with different features `F` and generates various synthetic datasets for classification that will

Table 1: Description of features in the problem.

Description.	Value.
Fn: number of data features in each instance (i.e. IF Fn=2, THEN instance f1, f2:class).	**Fn**: from 2 to 5
Fc: number of data features that apply conjunction.	**Fc**: at least 1/2 of Fn
Fd: number of data features that apply disjunction.	$Fn \geq$ **Fd** ≥ 0
Fi: number of irrelevant data features in each instance.	$Fn \geq$ **Fi** ≥ 0 and **Fi** $\neq Fc \neq Fd$
Fr: number of redundant data features in each instance.	$Fn \geq$ **Fr** ≥ 0
Fan: percentage of noise level that apply to class.	**Fan**: from 0% to 50%
Fcn: percentage of noise level that apply to data features.	**Fcn**: from 0% to 50%
Fcbl: percentage of class balance within Class 1 and Class 0 to dataset (i.e. percentage of Class 1, percentage of Class 0).	**Fcbl**: from 0% to 100%
Fcbd: percentage of decision boundary to class (i.e. wide or small decision boundary of the class).	**Fcbd**: from 0% to 50%

Table 2: Sample of the generated instances.

Problem.	Instance and class.
Example 1: conjunction Problem: <Fn=2,**Fc=1**,Fd=0,Fi=0,Fr=1> Class 1: f1<=0.4 AND f2<=0.8	**f1 f2 : class** 0.1 0.1 : 1 0.5 0.5 : 0
Example 2: disjunction Problem: <Fn=5, Fc=0, **Fd=2**, Fi=0, Fr=1> Class 1: f1<=0.5 OR f2<=0.2	**f1 f2 f3 f4 f5 : class** 0.1 0.1 0.3 0.4 0.5 : 1 0.6 0.6 0.3 0.3 0.1 : 0
Example 3: conjunction and disjunction Problem: <Fn=5,**Fc=1**,**Fd=2**,Fi=0,Fr=1> Class 1: f1<=0.5 AND f2<=0.6, f3>=0.5	**f1 f2 f3 f4 f5 : class** 0.6 0.6 0.5 0.3 0.1 : 1 0.1 0.1 0.3 0.4 0.5 : 0
Example 4: noise to class Problem: <Fn=2,Fc=0,Fd=1,Fi=0,Fr=1,**Fan=10**> Class 1: f1<=0.4	**f1 f2:class:random value** 0.1 0.1 : 1 : 0.5 0.5 0.5 : 0 : 0.6 FOR each feature (f1 to f2). IF(random value>noise)THEN remain the class. ELSE flip the class either from 1 to 0 or from 0 to 1.
Example 5: noise to condition Problem: <Fn=2,Fc=0,Fd=1,Fi=0,Fr=1,**Fcn=10**> Class 1: f1<=0.4	**f1 f2 : class** 0.1 0.1 : 1 0.5 0.5 : 0 FOR each feature (f1 to f2) IF(feature's value>noise) THEN remain the value. ELSE the value will be replaced by a new random value.
Example 6: conjunction and class balance Problem: <Fn=4,**Fc=2**,Fd=2,Fi=0,Fr=1,**Fcbl=70**>	**Class1=70%,Class0=30%** Formula: $P(AandB) = P(A)xP(B)$ If P(A and B)=0.7, then P(A),P(B),P(C),P(D)=0.914 Set value f1,f2,f3,f4>=0.1
Example 7: disjunction and class balance Problem: <Fn=2,Fc=0,**Fd=2**,Fi=0,Fr=1,**Fcbl=60**>	**Class1=60%,Class0=40%** Formula: $P(AorB) = P(A) + P(B) - [P(A)xP(B)]$ If P(AorB)=0.6, then P(A)<0.6 If P(A)=0.5, then P(B)=0.2 Set value f1>=0.5, f2>=0.8

Example 8: decision boundary
Problem:
<Fn=2,Fc=1,Fd=1,Fi=0,Fr=1,**Fcbd=20**>
Class 1: f1<=0.5 and f2<=0.5
$DB(DecisionBoundary) = 0.5$
$R(Range) = Fcbd/100$
	FOR each feature (Xn:f1 to f2)
	IF(feature's value>decision boundary)
	$Xn = (DB + (R/2)) + (Xn - DB/1 - DB)x(1 - DB - (R/2))$
	ELSE
	$Xn = (Xn/DB) + (DB - (R/2)$

Table 3: Changes in features F using Tabu Search.

INITIAL SOLUTION: 2, 1, 0, 0, 0, 50, 50, 70, 25
INITIAL PERFORMANCE: 44.0
Search COMPLETE!
BEST PERFORMANCE: 96.0
BEST SOLUTION: 2, 1, 0, 0, 0, 1, 50, 70, 25
NOTE:
R's classification performance increased from 44% to 96% using TS to adjust Fc,Fd,Fi,Fr,Fan,Fcn,Fcbl,Fcbd.

Table 4: Example of Receiver's condition-action rule.

Receiver's condition-action rule:
IF<condition> THEN<class>
$c_1 : [u_1, l_1], c_2 : [u_2, l_2], c_3 : [u_3, l_3], c_4 : [u_4, l_4], c_5 : [u_5, l_5] : class$
0.6:[0.98,0.5], 0.6:[0.8,0.4], 0.5:[0.6,0.3], 0.3:[0.5,0.2], 0.1:[0.5,0.08]:**0**

need to be solved by R. The meta-problem contains a list of features <Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd>.

For example, given <Fn=5 Fc=1 Fd=0 Fi=0 Fr=1 Fan=5 Fcn=5 Fcbl=50 Fcbd=5>, the dataset will consist of five data features such as f1,f2,f3,f4,f5, where f2 will have the same value with f1 (when number of redundant data features is 1, Fr=1). Therefore, the dataset may contain instances such as <0.6 0.6 0.5 0.3 0.1 :0> and <0.1 0.1 0.3 0.4 0.5 :1>. The data features of f1 and f3 will apply conjunction (when Fc=1), thus the class for each instance is determined as follows, IF (f1<=0.5 AND f3<=0.4) THEN 'Class 1', ELSE 'Class 0'. The noise level to apply to the condition is 5% (when Fcn=5), for any value of the data features f1,f2,f3,f4,f5 that is less than 0.05, it will be recalculated to any random value. When Fan=5, the noise level to apply to the action is 5%, for any generated value of the class that is less than 0.05, the class will be flipped either from '1' to '0' or '0' to '1'. There will be 50% instances of 'Class 1' and 50% instances of 'Class 0' (as Fcbl=50). In order to avoid the classification agent learn the inverse problem, the value of Fan and Fcn is set within 0-50%. If class balance Fcbl=100, all instances will belong to 'Class 1'.

Second, R needs to classify each instance as either belonging to 'Class 1' or 'Class 0' and sends either '1' for 'Class 1' or '0' for 'Class 0' as suggested by its rules (Table 4). In response, S sends a numerical reward of '1000' for correct classification else '0' returned to R.

S's objective can be either to maximize R's performance by decreasing the problem's difficulty or minimize R's performance by increasing the problem's difficulty. Based on R's performance (i.e. problem became 'harder' or 'easier' based on change in R's performance), S uses the effect of varying feature values F using Tabu Search for generating a new meta-problem in the next set of iterations (Table 3).

4. EXPERIMENTAL DESIGN

Three sets of experiments have been performed in order to confirm the suitability of the synthetic datasets as the control environment for the *Two-Cornered Coevolution LCSs Framework* for classification tasks. In *Experiment 1*, different combinations of the problem features within the problem (i.e. increasing and decreasing value of Fan, Fcn and Fcbl) are enumerated to analyze R's performance with respect to those changes. In *Experiment 2*, R was tested on different datasets (i.e. different problems) throughout the experiments. In *Experiment 3*, TS was applied in S to search for the best combination of features in the problem. Based on R's classification performance, S changes the combination of features in the problem for generating the next problem for R to learn.

4.1 Experimental Setup

In our implementation, R was executed following *Wilson's explore/exploit scheme* [12], which has become the standard approach in XCS. The explore and exploit scheme are run alternately with probability 50% Explore and 50% Exploit. The parameter settings used were also similar for the most part in [7], where a few modifications were made to improve efficiency, including the number of classifiers in the population, the tournament size and the number of iterations. The tournament size is a fraction $\tau = (0, 1]$ of the current action set size ($\tau = 0.4$ is the suggested value). Both GA-subsumption and ActionSet-subsumption are activated.

The population size N was limited to 500 classifiers for each problem, R learns 2,000 instances (i.e. R runs for 2,000 iterations). Both values are low for standard LCSs to reduce training times as the overall meta-problem task is time consuming. All experiments were run 30 times with different random seeds for analysing the results. R's performance was calculated from exploit trials (i.e. classification performance). The classification performance is the percentage of correct classifications over all the instances from each of the datasets.

S generates a problem containing a list of parameters (features of the problem) and hence creates a dataset (i.e. set of n instances, n=2,000), where each value of an instance is randomly initialized in the interval of $[0, 1]$. The instance is labeled (i.e. either 'Class 0' or 'Class 1', as a two-class problem) according to the defined problem. S uses TS to search for the best combination of features in the problem (i.e. either to maximize or minimize R's performance) by avoiding becoming trapped in the local optimum.

5. RESULTS

5.1 Experiment 1 Results

In this experiment, different combination of features in the problem (i.e. increasing and decreasing value of certain features such Fan,Fcn and Fcbl) on four problem domains (i.e. Fn=2 to 5) are exhaustively explored. In each problem domain, the value of Fan and Fcn are incremented by 5 within the range of 5 to 50, and value of Fcbl is also incremented by 5 within the range of 50 to 100, while other features value Fc,Fd,Fi,Fr are set where Fc=1,Fd=0,Fi=0,Fr=0. Figure 4 and Figure 5, show the trade-off surface for examples of Fn=2 that alters R's performance. If there is no gradient in difficulty then it would be impossible for S either to make the problem 'harder' or 'easier' for R to learn.

R achieves a good performance of 90-95%, when the class balance Fcbl is within 50-100% and the noise level that applies to the action Fan is within 5-20% (Figure 4). R also achieves a good performance of 90-100%, when the class balance Fcbl is in the range of 50-100% and the noise level that applies to the condition Fcn in the range 5-40% Figure 5. Results of R's performance in the other problem domain (i.e. Fn=3,Fn=4 and Fn=5) are varied but show the same pattern (i.e. the affects of changing the value of Fan,Fcn

and Fcbl either will increase or decrease R's performance) so are not included in this paper due to space restriction.

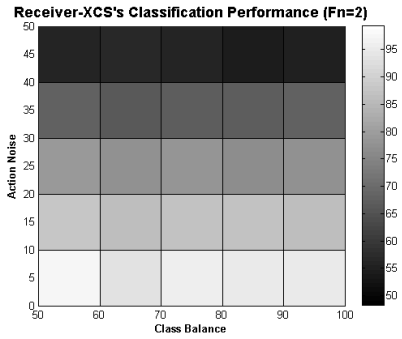

Figure 4: Trade-off surface of R's performance (based on the average of R's classification performance from 30 runs in training mode for learning a two-class classification problem when Fn=2, while value of Fan and Fcbl is incremented by 5).

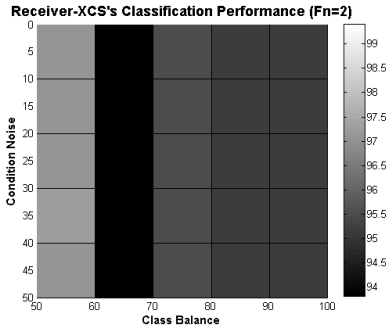

Figure 5: Trade-off surface of R's performance (based on the average of R's classification performance from 30 runs in training mode for learning a two-class classification problem when Fn=2, while value of Fcn and Fcbl is incremented by 5).

Note, 100% performance is not reached due to limiting the number of classifiers and training instances. Also the performance improves as the class imbalance increases beyond 90% as the majority class facilitates general (possibly over-general) classifiers and the crude (deliberately so) fitness function does not take this into account. Results show that a gradient in difficulty exists in relation to features. Therefore, this enumerated-information can be used in later experiments to set up the starting problem for S and determine whether S can vary the difficulty levels appropriately.

5.2 Experiment 2 Results

Figure 6 presents R's classification performance on four problem domains with different combination of features in the problem (i.e<Fc Fd Fi Fr Fan Fcn Fcbl Fcbd>). Result shows that R was able to learn the generated datasets with varied performance. These results suggest that the problem domain of Fn=5 is the hardest problem compared to others, while the problem domains of Fn=2 and Fn=3 are the easiest problem as would be expected. Therefore, R performance can be tested on these four problem domains, whilst the problem's difficulty can be varied within that domain.

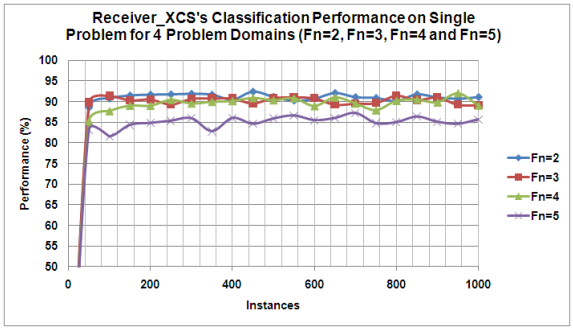

Figure 6: Average of R's classification performance from 30 runs in training mode for learning a two-class classification problem on 4 problem domains. The problem consists of 2,000 instances, result from 1000 Exploit trial.

5.3 Experiment 3 Results

Figure 7 shows R's classification performance when TS is applied in S to search for the best combination of features (i.e. <Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd>) for a two-class classification problem, with the objective to *maximize* R's performance. S was started with a predefined problem, where <Fc=1 Fan=50 Fcn=50 Fcbl=70 Fcbd=25> that was likely to be a 'hard' problem. TS is used to vary the features except for Fn.

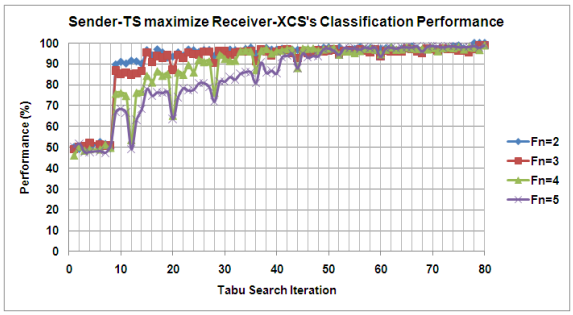

Figure 7: Average of R's classification performance from 30 runs in training mode for learning a two-class classification problem on 4 problem domains, where TS is used in S for adjusting the difficulty levels (i.e. from 'hard' to 'easy').

Figure 8 shows R's classification performance when TS is applied in S to search for the best combination of features (i.e. <Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd>) for a two-class classification problem, with the objective to *minimize* R's performance. S was started with a predefined problem, where <Fc=1 Fan=5 Fcn=5 Fcbl=50 Fcbd=5> that was likely to be an 'easy' problem. TS is used to vary the features except Fn.

The results suggest that applying TS in S is suitable for helping S discover combinations of features in the problem to alter R's performance (i.e. S can autonomously determine the effect of individual problem features towards R's performance). For example, in the problem domain Fn=2, TS changes the initial-predefined problem <2 1 0 0 0 50

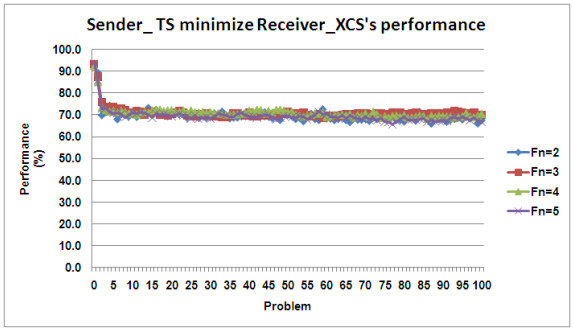

Figure 8: Average of R's classification performance from 30 runs in training mode for learning two-class classification problems on 4 problem domains, where TS is used in S for adjusting the difficulty levels (i.e. from 'easy' to 'hard').

`50 70 25>` to the next problem `<2 1 1 0 0 1 50 70 25>` which increases R's performance from 44% to 96% (Figure 7). In another example, TS changes the initial-predefined problem `<2 1 0 0 0 5 5 50 5>` to the next problem `<2 1 1 0 1 0 3 50 7>`, which decreases R's performance from 93% to 89% (Figure 8). S is able to adjust the difficulty levels by varying the features in the problem either to maximize or minimize R's performance, where S can either make the problems 'harder' or 'easier' for R to learn.

6. CONCLUSIONS AND FUTURE WORK

Generating datasets through specifying features rather than patterns has led to a system that can tune datasets to adjust the performance of a LCS in a desired manner. An enumerative analysis of the potential datasets identified the performance gradients but the on-line learner identified useful gradients more efficiently. Important features, which control the ease of learning within the problem domain for the classification system, were identified. Although the system can be extended to expand the maximum values of features f in the datasets (e.g. `Fn`>5), the work only focuses on identifying features values `F` that affect the classification agent's performance (i.e. controlling for possible confounding variables to the problem's complexity).

Future work will investigate different types of performance measure (i.e. robustness, scalability, and predictive accuracy) that relate to difficulty factors. Furthermore, the developed system will be used to modify the process of generating classification problems for *Three-Cornered LCSs Framework*, where the problem domain will tune autonomously depending on the two different agents' ability to learn (i.e. the Receiver (R) and the Interceptor (I) which used different techniques of learning) [14]. I is required to direct S to change the problem's difficulty when R becomes stagnated.

7. REFERENCES

[1] C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Natural Computing Series. 2006.

[2] R.O. Duda, P.E. Hart and D.G. Stork. *Pattern Classification 2nd Edition.* John Wiley, 2001.

[3] Tin Kam Ho and Mitra Basu. *Complexity Measure of Supervised Classification Problems.* IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 24/2002(3):289-300. 2002.

[4] M.V. Butz. *Kernel-based, Ellipsoidal Conditions in the Real-Valued XCS Classifier System.* Genetic Evolutionary Computational Conference (GECCO 2005). ACM, 2005.

[5] M.V. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design.* Springer. 2006.

[6] Muhammad Iqbal, Will N. Browne and Mengjie Zhang. *XCSR with Computed Continuous Action.* Australisian AI 2012, pages 350-361. Springer, 2012.

[7] F. Kharbat, L. Bull and M. Odeh. *Revisiting Genetic Selection in the XCS Learning Classifier System.* The 2005 IEEE Congress on Evolutionary Computation, pages 2061-2068. 2005.

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

[9] Syahaneim Marzukhi, Will N. Browne and Mengjie Zhang. *Two-Cornered Learning Classifier Systems for Pattern Generation and Classification.* The 12th Genetic and Evolutionary Computation Conference (GECCO 2012), pages 895-902. ACM, 2012.

[10] R.J. Urbanowicz and J.H. Moore. *Review Article Learning Classifier Systems: A Complete Introduction, Review, and Roadmap.* Journal of Artificial Evolution and Applications, Volume 2009:1-25. 2009.

[11] O. Sigaud and S.W.Wilson. *Learning Classifier Systems: A Survey.* Soft Computing - A Fusion of Foundations, Methodologies and Applications, Volume 11(11):1065-1078. 2007.

[12] S.W. Wilson. *Classifier Fitness Based on Accuracy.* Evolutionary Computation, Volume 3 (2):149-175. Massachusetts Institute of Technology, 1995.

[13] S.W. Wilson. *Get Real! XCS with Continuous-Valued Inputs.* Learning Classifier Systems, From Foundations to Applications, LNAI-1813,pages 209-219. Springer, 2000.

[14] S.W. Wilson. *Coevolution of Pattern Generators and Recognizers.* Lecture Notes in Computer Science (LNCS), Volume 6471/2010(1):38-46. 2010.

[15] Nuria Macia, Albert Orriols-Puig and Ester Bernado-Mansilla. *Beyond Homemade Artificial Data Sets.* Hybrid Artificial Intelligence Systems Lecture Notes in Computer Science, Volume 5572/2009:605-612. 2009.

[16] Z. Michalewicz and D.B. Fogel. *How to Solve It: Modern Heuristics.* Springer, 2000.

[17] J.H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, Artificial Intelligence.* University of Michigan Press, pages 313-329. 1975.

[18] Sean Luke. *Essentials of Metaheuristics: A Set of Undergraduate Lecture Notes.* Department of Computer Science, George Mason University. 2010.

[19] Micheal Gendreau. *A Tutorial On the Tabu Search.* Department of Computer Science, de Montreal University, Canada. 2000.