An Investigation of Genetics-Based Machine Learning as Applied to Global Crop Yields

Will Gantt

April 25, 2017

1 Introduction

As Earth's population continues to grow and as the effects of climate change intensify, it is vital to global food security and market stability that we better understand the environmental and economic factors that most affect agricultural output around the world. While rising temperatures may benefit certain crops in the short term, extreme weather and higher levels of atmospheric CO_2 will, if left unchecked, have disastrous consequences for water supply, soil fertility, and, consequently, for yields. [1]. Furthermore, the United Nations Department of Economic and Social Affairs predicts that the world population will reach 9.7 billion by 2050 and 11.2 billion by the end of the century [2].

These forecasts demand prudent planning, and such planning requires that we discern trends in agricultural data. To that end, I have designed and written a learning classifier system (LCS)—a powerful and versatile tool for data mining—and applied it to data collected by Erik Nelson in order to identify correlations between various agricultural inputs and changes in crop yields. This paper presents the preliminary results of those investigations.

The following section gives a brief overview of genetics-based machine learning (GBML) and LCSs, and of the research done by Nelson and Congdon that gave rise to this project [3]. Section 3 provides a thorough description of the system design of my own LCS. Sections 4 and 5 detail the experiments conducted and the results, respectively. Finally, section 6 offers some concluding remarks and suggests directions for future work.

2 Background

In this section, I provide some context for my project. First, I introduce the GBML paradigm and consider the place of LCSs within it. Next, I give a general overview of LCSs and their basic structure. I then discuss a few important developments in the history of LCSs for supervised learning. Lastly, I conclude by talking specifically about the earlier work of Nelson and Congdon [3].

2.1 Genetics-Based Machine Learning

GBML is a machine learning approach based on the use of evolutionary computation. Briefly, evolutionary computation comprises a set of optimization algorithms that evolve populations of candidate solutions to a particular problem through a process inspired by biological evolution. Many of the most interesting and important problems in machine learning have large and noisy search spaces, and it is in such contexts that techniques from evolutionary computation have proved particularly effective.

The precise relationships between various GBML-related terms and concepts have changed over the years, and usage in the community is often loose. GBML, once taken to refer exclusively to LCSs, is now considered a broader term that encompasses a number of other algorithms, including genetic programming, evolutionary ensembles, evolutionary neural networks, and genetic fuzzy systems [4]. For the purposes of this paper, it is important only to recognize that LCSs exist within a larger machine learning problem-solving framework.

2.2 What is a Learning Classifier System?

2.2.1 Structure

Introduced by John Holland in 1975, LCSs are GBML algorithms that evolve a population of rules[1] [5]. Each rule consists of a condition that indicates when the rule applies, as well as an action to take if the condition's criteria are met. As LCSs have applications to a variety of learning problems, the function of the rules depends largely on the task. With supervised learning problems, rules attempt to categorize a set of training examples by mapping the features of those examples to particular classes. In these cases, a rule's

[1]Rules are also frequently referred to as "classifiers." To maintain a clear distinction between individual *classifiers* and the *classifier systems* that comprise them, I use the term "rule" exclusively throughout to denote the former, and "LCS" to denote the latter.

condition describes the features of the examples to which it applies, and the action specifies the class to be assigned to examples matching the condition. Problems in reinforcement learning, by contrast, often involve determining the best action for an agent to take in response to certain inputs or stimuli from the environment. Here, a rule's condition describes a set of inputs and its action determines how the agent should react given those inputs.

Regardless of the types of problems to which they are applied, all LCSs share certain basic structures and mechanisms. Holmes et al. have identified four such components [6]. However, their model is more characteristic of LCSs used in reinforcement learning than those used in supervised learning. I suggest a slight modification of their model that covers both:

1. *Population.* All LCSs have a population of rules. Although some systems allow for the growth or reduction of the population, there is typically a limit on the maximum number of rules it may contain.

2. *Learning.* An LCS must have a means of coercing its population toward rules that generate good actions or accurate classifications. Numerous methods have been used for this purpose and they tend to vary based on the kind of learning. Many early LCSs designed for reinforcement learning used the bucket brigade algorithm [7], while more contemporary systems often rely on Q-learning [8, 9] or Q-learning-inspired algorithms. For supervised learning tasks, other systems have used genetic algorithms exclusively [10]. Still others have used different approaches, including ensemble learning [11] and bayesian methods [12].

3. *Discovery.* An LCS must also have mechanisms for generating new rules. For that purpose, a vast majority of systems employ genetic algorithms (GAs), which provide two useful operations: crossover and mutation. Crossover produces two "offspring" rules from two existing "parent" rules, and mutation alters some of the attribute values of the offspring. However, many LCSs make use of additional operators. The *covering* operator, for example, which creates a new rule when there are none that match the current input, has been widely used [9, 13, 14]. *Specify*, a second popular operator, proposed by Pier Lanzi, attempts to counteract the over-generalization of rules by specifying the attribute values of a rule according to a particular input [15].

4. *Classification or Action Selection.* Finally, an LCS must have some procedure for deciding what action to take or how to classify a particular example. In the typical case, the action or classification is selected

based on the actions recommended by the rules in the *match set*—the set of rules matching a particular input. One possibility is simply to choose the action of the fittest rule in the match set. Another is to conduct a weighted vote of all the rules in the set, where each rule casts a "vote" for its action, weighted by its fitness. The selected action, then, is the one receiving the greatest number of points. Some systems have also incorporated random action selection alongside one of these first two methods [13].

These are, at the broadest level, the modules that feature in all LCSs. There is, of course, substantial variation between implementations, which is to be expected of a paradigm with applications as diverse as those of the LCS algorithm.

2.2.2 Genetic Algorithms

As GAs constitute such a critical component of LCSs, they merit a quick and very general overview. GAs evolve populations of candidate solutions to a particular problem by a process inspired by biological evolution. In a traditional implementation, individuals are represented as bit strings, where each index in the strings corresponds to a particular property of the solution and a 1 or 0 at that index indicates the presence or absence of the property, respectively. In LCSs, a third character (a "don't care" value, often denoted by "#") is often used to indicate that the property is not relevant for a given rule. In principle, solutions may take any form, and GA variants have been developed to accommodate structures of different kinds, including whole objects [16]. The canonical GA works as follows[2]:

1. *Initialization.* Generate a random population of candidate solutions.

2. *Evaluation.* Evaluate the fitness of each individual according to some fitness function.

3. *Selection.* Select n individuals from the population to reproduce. The probability that a given individual will be selected is usually proportional to its fitness.

4. *Crossover.* Randomly pair the n selected individuals. For each pair, combine the bits of one member of the pair with those of the other (e.g. via one-point crossover) to produce two new solutions ("offspring").

[2]For ease of illustration, I have used bit strings as the solution type in this example.

5. *Mutation.* Flip some number of the bits of each of the offspring with probability p.
6. *Replacement.* Replace the n least-fit members of the original population with the offspring created in (5).
7. Repeat (2)-(6) until the stopping condition (e.g. a target average fitness, a specific number of generations, or a runtime limit) is met.

The structure of the GA is just one vector along which different implementations of LCSs can differ. The next section outlines a second.

2.2.3 Pittsburgh and Michigan Styles

One of the most common and useful categorizations of LCSs distinguishes between *Michigan-* and *Pittsburgh-* or *Pitt-style* systems. The difference pertains to what counts as an "individual" in the population. In Michigan systems, an individual is a single rule. In Pitt systems, however, an individual is an entire rule *set.* Correspondingly, a solution in a Michigan system often comprises the entire population, while a solution in a Pitt system consists of a single individual (a rule set). This difference has important implications for many aspects of LCS design, including the learning algorithm, credit assignment, rule syntax, and genetic operators. Perhaps unsurprisingly, each style has notable advantages and disadvantages with respect to the other. Michigan systems, for example, typically require less memory and are less computationally intensive, while Pitt systems avoid the difficulties involved in distributing credit across individual rules. Neither type has emerged as obviously superior to the other, but Michigan systems do appear to predominate in the literature [17].

2.3 Developments in LCS Research

Numerous thorough histories and comparisons of LCS research have been written over the years [17, 18, 19, 20], and it is beyond the scope of this paper to undertake another one. Instead, I wish to highlight just a few key milestones in the particular domain of LCS research with which my project is concerned—namely, supervised learning and data mining. Each of the systems listed below introduced at least one important concept to that domain.

2.3.1 LS-2

Schaffer and Grefenstette's LS-2 [21] was the first attempt to use an LCS for a genuine supervised learning classification task. Specifically, they applied their system to the multi-class problem of classifying five human gait types based on EMG signals from the leg muscles. A total of 11 training examples were used, and rules were represented as 12-bit strings from the ternary alphabet ($\{0, 1, \#\}$), with each index representing a particular signal.

LS-2 evolved a population of rule *sets* and may thus anachronistically be categorized as a Pitt-style LCS, though the distinction had yet to be formulated when the article was published. Of particular note in this system is the fitness scheme. LS-2 represented fitnesses, not as scalar quantities, but as vectors whose elements corresponded to the different possible classes (gait types). For each rule set, a fitness vector was computed as a function of the number of correctly and incorrectly classified examples in each class. When constructing the population for the next generation, a portion of the rules were selected on the basis of each element in the fitness vector. In allowing fitness to be relativized to individual classes, Schaffer and Grefenstette introduced the notion of *niching*—the specialization of rules to particular regions of the solution space—which featured heavily in later systems, and which remains a critical topic in contemporary research.

2.3.2 FCS

Many kinds of data in the real world have continuous-valued attributes. Given the traditional bit string rule representation of an LCS, determining the best way to handle data of this sort is a challenge. In 1991, Manuel Valenzuela-Rendón proposed a solution to this problem using fuzzy sets [22]. In fuzzy set theory, the membership of an element in a set is determined by a membership function. Defining membership in this way allows for elements to be *partial* members of sets.

Valenzuela-Rendón's "fuzzy classifier system" (FCS) applied this logic to rule representation. Suppose the condition of a rule R contains a "temperature" attribute, whose possible values are "cold," "mild," and "hot." Instead of assigning R one of those values, FCS would define a membership function for each value, and R would be assigned some combination of those functions. If, for example, R was assigned the functions for "cold" and "mild," the result would be a curve for the temperature attribute that, given a real-valued temperature, indicated the *degree* to which that temperature should be considered "cold" or "mild."

In short, structuring rules in this way introduced two valuable concepts to the LCS community: (1) the capability of LCSs of handling real-valued inputs, and (2) the possibility of eliminating hard boundaries between classes.

2.3.3 XCS

Although designed explicitly for reinforcement learning, the "eXtended Classifier System" (XCS), proposed by Wilson in 1995, exerted such a profound influence on LCS research of all kinds that it deserves mention here [13]. Nearly all of the key features of XCS had been implemented in earlier systems, but its impressive accuracy and generalization capabilities made clear for the first time their potential effectiveness.

XCS pioneered a slew of techniques that have since become staples of LCS design, particularly in reinforcement learning, including the use of Watkins's Q-learning algorithm for credit assignment and the use of a niche GA [17]. However the technique of greatest relevance to supervised learning was the use of accuracy-based fitness. Wilson was not the first to use accuracy-based fitness (credit for that goes to Holland's first LCS, CS-1 [23]), but a substantial majority of previous systems used a different parameter known as *strength*. Strength was decidedly a reinforcement learning mechanism, and measured the expected reward (or "reward prediction") from the environment if the action of a particular rule was taken. The problem with strength-based fitness was that it tended to eliminate rules that predicted lesser rewards, but that nonetheless reliably led to the best action in specific circumstances. Even though accuracy had been used before, XCS radically increased its popularity and is now considered the primogenitor of all contemporary accuracy-based systems [17].

2.3.4 UCS

In 2003, Bernadó-Mansilla and Garrell-Guiu presented a supervised learning variation on XCS [14]. Dubbed "UCS," their system adapted the reinforcement learning model of XCS to better suit multi-objective classification tasks. UCS retained all of the main features of XCS, including its accuracy-based fitness scheme, a GA that encouraged niching, and an online learning style, but it changed the manner in which accuracy was computed. The details of the differences are not salient; rather, the significance of UCS consists in its demonstration that ideas from reinforcement learning within LCS research could, with some effort, be refashioned to yield good solutions to problems in supervised learning.

2.4 Global Crop Yields

This project examines data on crop yields collected by Erik Nelson and his students in the economics department. The data relate the various environmental and economic variables to the agricultural output of countries, both in units of mass (Mg/ha) and in units of energy (Mkcal/ha), from 1975 to 2007. The variables considered cover such categories as growing season weather; crop choice; investment in irrigation capability, land, and machinery; agricultural technology; fertilizer use; and cropped footprint.

Using these data, Nelson and Congdon evaluated the relative impact of the year-to-year changes in the different variables on the change in global and regional crop yields [3]. The authors applied two analytical methods: fixed-effects econometric modeling and decision trees, each of which is explained in greater depth below. For both, annual changes in yield were discretized to the categories "high" (H), "medium" (M), and "low" (L).

2.4.1 Fixed Effects Modeling

Fixed-effects models offer a means of accounting for any unobserved, time-independent factors that affect a dependent variable in the analysis of panel data—that is, multi-dimensional data containing measurements across time. In Nelson and Congdon's work, the dependent variables considered were the per-hectare crop yield in metric tons (Mg) and in millions of kilocalories (Mkcal). Let y_{ct} be the per-hectare crop yield (in whichever units) for a country c in a year t. The fixed effects model, then, has the form:

$$y_{ct} = \beta_1 x_{ct1} + \beta_2 x_{ct2} + \ldots + \beta_N x_{ctN} + \mu_{ct} + a_c$$

Where x_{cti} is the value of the independent variable x_i for country c in year t; β_i is the coefficient for the independent variable x_i; μ_{ct} (the "idiosyncratic error") covers all unobserved, time-*variant* effects on y_{ct}; and a_c (the "fixed effect") captures all unobserved and time-*invariant* effects. The specific variables used are shown in table 1 (as above, the subscript "$_{ct}$" indicates that the variable is for a country c in a year t).

Thus, their fixed effects model, computed using the method of least squares, has the form

$$Y_{ct} = \beta_0 + \boldsymbol{\beta_1 X_{ct}} + \boldsymbol{\beta_2 K_{ct}} + \beta_3 A_{ct} + \beta_4 S_{ct} + \beta_5 Ict + \beta_6 \boldsymbol{Z_{ct}} + \beta_7 t + \beta_8 F_{ct}$$

Using this model, the authors constructed expected yield curves for each country between 1975 and 2007 (in both Mg/ha and Mkcal/ha), as well as

Variable	Description
$\boldsymbol{X_{ct}}$	a vector of harvested hectare percentages across different crops.
$\boldsymbol{K_{ct}}$	a vector of variables measuring investment in agricultural machinery and equipment per harvested hectare.
A_{ct}	the total harvested hectarage.
S_{ct}	the soil quality.
I_{ct}	the percentage of harvested land equipped for irrigation.
$\boldsymbol{Z_{ct}}$	a vector of various annual weather statistics.
F_{ct}	the total amount of fertilizer used.

Table 1: Variables used in Nelson and Congdon's analysis of crop yield data [3]

for the temperate and tropical regions, and for the world. They also constructed counterfactual yield curves for each country and for each variable, by holding that variable constant at 1975 levels. By comparing the integral of an expected yield curve with the integral of a corresponding counterfactual curve for a variable v, one can evaluate the impact of v on yield relative to other variables.

2.4.2 Decision Trees

Decision trees are machine learning tools for classification that, much like LCSs, attempt to group the outcomes of a process based on the inputs to that process. In this case, the outcomes are simply the changes in country-level crop yields and the inputs are the year-to-year changes in the values of the variables listed above. Each node in the tree corresponds to an attribute of the input (e.g. change in soil quality) and specifies a value of that attribute according to which the training examples may be partitioned into two groups: examples whose value for that attribute falls above the specified value and those whose value is at or below the specified value. Beginning at the root node and partitioning in this way generates some number of leaves, each containing a subset of the examples. The objective is to construct a tree such that the examples in each leaf are as homogeneous as possible with respect to their outcomes. A path from the root node to one of the leaves thus characterizes the attribute values of the examples in that leaf. The closer to the root an attribute node appears, the more it may be said to "explain" the differences in outcomes among the examples.

2.4.3 Results

Both methods found that changes in crop mix accounted for more of the growth in yield between 1975 and 2007 than any other single variable. In the tropics, an increase in the average daytime growing season temperature correlated with a noticeable decrease in yields. Perhaps surprisingly, investment in irrigation, land, and machinery and equipment, as well as the quality of the cropped soil, had a negligible impact. The methods disagreed, however, on the importance of fertilizer use: Where the econometric model showed it to have a significant positive influence on yields, the decision trees revealed no such relationship.

With all of this analysis already done, one may reasonably question the utility of re-evaluating the same data with an LCS. There are at least two advantages:

1. LCSs have proved themselves effective in supervised learning tasks of precisely this sort and have the potential to reveal important patterns in the data not made evident by either decision tree or regression analysis.

2. An LCS may help to resolve the conflicting aspects of the results of these first two methods. Specifically, the discrepancy in the reported impact of fertilizer use on yield gains.

3 System Design

In this section, I present my design for a Michigan-style LCS for the classification of examples with real-valued attribute vectors. The explanation consists of four parts: The first part explains the representation of rules, including a short description of their attributes, and a longer discussion about the choice of representation of the condition in particular. The second part describes the learning component and offers a general overview of the GA, as well as more detailed analysis of its constituent modules. The third part covers the testing component—the algorithm for the classification of new examples once a rule set has been created. Finally, the fourth part enumerates the parameters of the LCS.

3.1 Rule Representation

Rules are represented as objects with the following member variables:

id: A unique identifier for the rule in the population.

classification: The class, given by an integer value, that an example matching the rule will be assigned.

true_positives: The number of examples that the rule both matches and correctly classifies.

true_negatives: The number of examples that are neither matched by the rule nor of the same class.

false_positives: The number of examples that the rule matches, but incorrectly classifies.

false_negatives: The number of examples that have the same class as the rule but that the rule does not match.

num_dont_care: The number of Attribute objects in the rule's condition (represented as a vector of Attribute objects) whose *dont_care* variable is currently set to true.

fitness_1: An initial fitness value used to determine a preliminary ranking of the rules.

fitness_2: A fitness value based on Fitness 1, but that takes into account the fact that there may be multiple rules that cover the same examples.

condition: A vector of Attribute objects that describes the attribute values an example must have to be matched by the rule.

As mentioned in section 2.2.2, much of LCS research has concerned itself with the classification of examples with binary-valued attributes that simply indicate the presence or absence of some feature in the example. The conditions of rules used in LCSs of this sort may therefore be represented as strings of characters from the ternary alphabet $\{0, 1, \#\}$. When dealing with real values, however, determining how best to represent the condition is more difficult. With real values, not only does the search space expand to a theoretically infinite size, but the genetic operators (crossover and mutation) have to be adapted as well. Thus, the condition must be constructed in a manner that (a) reduces the search space to a reasonable size, and (b) does not hinder or over-complicate the functioning of the genetic operators.

The success of fuzzy LCSs like FCS [22] and Fuzzy-UCS [9] (a fuzzy update of the original UCS) suggests one promising approach. Given the complexity of these sytems, though, and time limitations, I chose the simpler, but nonetheless well-supported method of discretizing all attributes into some number of quantiles. Each attribute in the rule's condition not marked as irrelevant covers a range of values specified by one of the quantiles. Note that a condition constructed in this way meets both of the criteria given in the preceding paragraph: It radically restricts the search space (since each attribute will have a small number of quantiles) and it does not inhibit or fundamentally alter the behavior of the genetic operators (as will be shown in more detail further on in this section). The number of quantiles used is the same across all condition attributes and is specified by the user. The condition is represented as a vector of Attribute objects, each of which has a name, a quantile number, an upper and lower bound describing the range of that quantile, and a boolean "dont care" value indicating whether the attribute is relevant for the rule to which it belongs.

3.2 Rule Discovery and Learning

3.2.1 Overview

As with virtually all other LCSs, this system uses a GA in its discovery component. One complete run of the LCS attempts to characterize a single "target" class from the data. The population is initialized with a user-specified number of rules whose class attribute (action) values are set to that of the target class, and whose condition attributes are randomly either marked as irrelevant (i.e. with their *dont_care* variables set to true) or else assigned a random quantile. For each rule in the population, the system tallies the number of true positives, true negatives, false positives, and false negatives among the examples in the training set. The rule is then assigned a preliminary fitness by computing the odds ratio and adding a small boost for each of its attributes that is marked as a "don't care." Once this has been done for the entire population, the rules are ranked by fitness.

Next, a second fitness is calculated using the same procedure, except that an example may count as a true positive for at most one rule. In computing the original fitness, it frequently happens that a single example will be counted as a true positive for multiple rules. In order to avoid such redundancy and to encourage diversity in the population, the second fitness counts an example as a true positive only for the fittest rule (based on the first fitness) that covers it.

The N best rules as evaluated by this second fitness are then immediately added to the next generation. To fill the remainder of the new population,

	$T+$	$T-$
$R+$	a	b
$R-$	c	d

Figure 1: The matrix of true positives, true negatives, false positives, and false negatives. The column indicates whether an example is a positive or negative instance of the target class. The row indicates whether the rule classifies the example as a positive or negative example.

$pop\_size - N$ rules are selected for reproduction from the current one using stochastic universal sampling. The selected rules are paired and crossed over to produce two offspring, which are mutated and added to the next generation. The process then repeats for a user-specified number of iterations.

3.2.2 Fitness Update

Both the first and second fitnesses are computed using the odds ratio, in conjunction with a slight bonus for each of its "don't care" attributes. A given example is either a positive instance ($T+$) of the target class or a negative instance ($T-$). Similarly, a given rule will *classify* the example either as a positive instance of the target class ($R+$) or a negative instance ($R-$). The four possible combinations of the example's actual class ($T+/T-$) and the class that a given rule selects for that example ($R+/R-$) yield the following matrix:

Clockwise from the top left, the elements of the matrix correspond to true positives, false positives, true negatives, and false negatives (for a given rule and example pair). As its name suggests, the odds ratio is a ratio of two odds:

1. The odds that a rule classifies an actual positive example of the target class as a true positive (a/c).

2. The odds that a rule classifies a negative example of the target class as a true positive (b/d).

Dividing (1) by (2) gives:

$$\frac{\left(\frac{a}{c}\right)}{\left(\frac{b}{d}\right)} = \frac{ad}{bc} = \frac{|TP| \cdot |TN|}{|FP| \cdot |FN|}$$

There is no difference in the calculation of the two fitnesses—only in the tallying of true and false positives and negatives. In the first fitness, an example e counts as a true positive for all the rules that match it. In the second fitness, e counts as a true positive only for the fittest rule that matches it, as determined by the first fitness. For a rule R that matches e, but that is not the fittest that does so, e is simply not included in any of the tallies for R. Because the second fitness is the one that drives the genetic algorithm, those rules that are redundant—that cover examples already covered by fitter rules—will over time be weeded from the population. Observe that this schema also rewards those rules that cover just a few examples, so long as those examples are not covered by fitter rules.

Note that when either $|FP|$ or $|FN|$ is 0, the odds ratio is undefined. In order to avoid this problem, $|TP|$, $|TN|$, $|FP|$, and $|FN|$ are all initialized to 0.5.

3.2.3 Selection

Selection is done based only on the second fitness using stochastic universal sampling (SUS), a superior alternative to traditional roulette wheel selection developed by James Baker [24]. In roulette wheel (or fitness-proportionate) selection, members of a population are selected randomly, where the probability that a member k is chosen is given by:

$$\frac{\text{Fitness}(k)}{\sum_{i=1}^{N} \text{Fitness}(i)}$$

The basic algorithm is shown below. First, the sum of the fitnesses of all members of the population is computed. Next, a number r in the range $[0, fitness\_sum]$ is determined by randomly selecting a value from the interval $[0, 1]$ and multiplying that value by the sum of the fitnesses. The algorithm then iterates over the members of the population. At each iteration, the fitness of the current member is subtracted from r. The member whose fitness, when subtracted from r, causes r to fall to or below 0 is the one selected. It is clear, therefore, that the larger the the fitness of a member, the more likely it is to be chosen.

The primary shortcoming of roulette wheel selection is the extent to which it is biased against rules of lesser fitness. Generally speaking, we want the rules selected to be among the fitter ones, but in order to avoid premature convergence, some low-fitness rules must participate regularly in the crossover.

Algorithm 1 : Roulette Wheel Selection

```
function RWS($Pop, N$)
    $fitness\_sum \leftarrow$ sumFitnesses($Pop$)
    $r \leftarrow$ random($0, fitness\_sum$)
    for $i$ in $\{0, \ldots, N-1\}$ do
        $r \leftarrow (r - \text{Fitness}(Pop[i]))$
        if $r \leq 0$ then
            return i
        end if
    end for
    return $N - 1$
end function
```

Algorithm 2 : Stochastic Universal Sampling

```
function SUS($Pop, N$)
    $fitness\_sum \leftarrow$ sumFitnesses($Pop$)
    $pointer\_dist \leftarrow fitness\_sum/N$
    $pointer\_start \leftarrow$ random($0, pointer\_dist$)
    $pointers \leftarrow$ newArray($N$)
    for $i$ in $\{0, \ldots, N-1\}$ do
        $pointers[i] \leftarrow pointer\_start + (i \cdot pointer\_dist)$
    end for
    $selected \leftarrow$ newArray($N$)
    for $p$ in $pointers$ do
        $j \leftarrow 0$
        while sumFitnesses($Pop[0], Pop[j]$) $< p$ do
            $j \leftarrow j + 1$
        end while
        $selected$.Add($Pop[j]$)
    end for
end function
```

SUS reduces this bias. Instead of generating a new random variable for each selection, SUS uses a single random variable to make all of its selections. This turns out to be of great utility in increasing the frequency with which low-fitness rules are selected. Furthermore, it has been shown to have faster average runtime and less variance in the fitnesses of the selected rules than does roulette wheel selection [25].

Algorithm (2) shows SUS in pseudocode. It will help, in understanding the behavior of SUS, to imagine the interval $[0, fitness\_sum]$ on the real number line. Additionally, imagine that each member of the population is given its own subinterval within the larger interval that is equivalent in length to its fitness (see Fig. 2). Finally, suppose that the subintervals are arranged left-to-right in descending order of fitness.

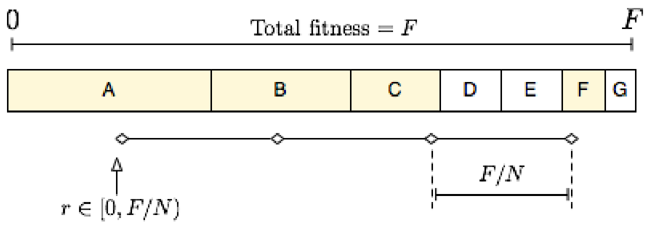

Figure 2: Beginning at a random starting point, stochastic universal sampling (SUS) uses evenly spaced pointers to select rules (Source: [26])

To begin, a value known as the "pointer interval" is computed by dividing the sum of the fitnesses (*fitness_sum*) by the number of rules to be selected (N). Then, a random number r is chosen from the range $[0, pointer\_interval]$. Beginning at r, the algorithm selects N positions along $[0, fitness\_sum]$ by counting out intervals of length $pointer\_interval$. The rules corresponding to the subintervals in which the positions fall are the ones selected.

Genetic Operators and Replacement

In my implementation, I used only the two canonical genetic operators, crossover and mutation.

Crossover. The GA uses single-point crossover. For every set of parents, a random integer in the range $[1, num\_attributes - 1]$ is selected as the crossover point. The condition of each of the offspring rules is created by swapping the attribute values to the right of the crossover point in each of the parents (see Fig.). If an attribute in one of the parents has its *dont_care* variable set to true, then the same attribute in one of the children will be marked the same way.

Mutation. After crossover, the mutation operator iterates over the condition attributes of each offspring. Each attribute has the same probability (*p_mutate*) of being modified. If it has been determined that an attribute is to be mutated, a second random number in the interval $[0, 1]$ is generated. If that number is less than or equal to a probability *p_dont_care*, the attribute's *dont_care* variable is set to true. If the number is greater than *p_dont_care*, the attribute is randomly assigned a new quantile.

After both operators have been applied, all of the offspring replace all of the non-elite ($pop\_size - N$) members of the current generation. The resulting population will be the one used in the next generation.

3.3 Classification of Examples

The classification of examples in the test set is very straightforward. At the end of *num_iters* iterations, the genetic algorithm terminates. The elite rules of the final population are the only ones used for the classification task. The reason for this is two-fold:

1. The elite rules in the final population are (mostly) those that have weathered repeated applications of the selection procedure and have proved themselves effective. Among the non-elites, which will have just been generated through crossover and mutation on the final iteration, there are likely to be many bad rules.

2. We are interested in minimizing the number of rules needed to accurately characterize the target class.

Once these rules have been identified, the LCS iterates over the examples in the test set. If a rule that covers the current example is found among the set of elites, the example is classified as a member of the target class. If no matching rule is found, the example is classified as a member of another, default class. Since the system attempts to characterize just one class per run (the target class), it matters little what is chosen for the default class, so long as it is different from the target class. After a class has been selected for the example, the system compares the selected class with the actual class of the example and categorizes it accordingly as a true positive, true negative, false positive, or false negative. The tallies of all four categories are outputted once all examples have been classified.

3.4 Parameters

The system takes the following arguments:

pop_size: the number of rules in the population.

num_iters: the number of iterations the LCS will run before terminating.

target_class: the class in the training data that the LCS will attempt to characterize.

elitism_rate: the approximate fraction of the population that is to be retained from one generation to the next.

p_mutate: the probability that a condition attribute will be mutated when the mutation operator is invoked.

p_dont_care: if it has been determined that an attribute is to be mutated, the probability that its *dont_care* variable will be set to true (instead of assigning it a new quantile).

training_set: an object of the Dataset class containing all of the examples in the training set and other relevant information.

test_set: another object of the Dataset class containing all of the examples in the test set and other relevant information.

4 Experiment and Results

4.1 Preliminary Evaluation on Small Data Sets

As a benchmark test of performance, I ran the system on the Iris and Wine data sets from the UC-Irvine Machine Learning Repository [citation]. Details of the data sets are shown in tables 2 and 3. For both sets, I used parameter values of *pop_size* = 40, *num_iters* = 1000, *elitism_rate* = 0.6, *p_mutate* = 0.25, and *p_dont_care* = 0.3. Ten experiments were run per class on each set using a simple holdout method with training and test sets of approximately equal size. Tables [#] and [#] show the mean accuracy, odds ratio, and numbers of true positives, true negatives, false positives, and false negatives for Iris and Wine, respectively.

Each run of the LCS attempts to characterize a single class (the "target class"). Quartiles are computed for each non-class attribute based on the range of values for that attribute across all examples in the target class.

Iris	
Number of Examples:	150
Classes:	Iris-Setosa, Iris-Versicolour, Iris-Virginica
Number of Non-Class Attributes:	4
Non-Class Attribute Value Types:	Real
Non-Class Attributes:	sepal length, sepal width, petal length, petal width

Table 2: The Iris dataset from the UCI Machine Learning Repository.

Wine	
Number of Examples:	178
Classes:	class 1, class 2, class 3
Number of Non-Class Attributes:	13
Non-Class Attribute Value Types:	Integer, Real
Non-Class Attributes:	Alcohol, Malic Acid, Ash Alcalinity of Ash, Magnesium, Total phenols, Flavanoids, Nonflavanoid phenols, Proanthocyanins, Color intensity, Hue, OD280/OD315 of diluted wines, Proline

Table 3: The Wine dataset from the UCI Machine Learning Repository. The names for the class attribute values are not specified.

Set	Class	Class Size	Accuracy	Odds Ratio	TP	TN	FP	FN

4.2 Experiments on Yield Data

For the yield data, I employed two kinds of test: the holdout method (used on the Iris and Wine data sets) and a ten-fold cross validation. In total, I used twelve data sets — the same as were used in Nelson and Congdon's experiments. All twelve are derived from the same original set, but each represents a unique combination of the following variables:

1. *Year.* Indicates whether or not the year is included as an attribute of the examples.

2. *Region.* Indicates whether the data set aggregates country-level data from countries in the tropics, countries in the temperate regions, or both (global).

3. *Yield Units.* Indicates whether the change in yield is expressed as Mg/ha or Mkcal/ha.

Thus, two possibilities each for the *year* and *yield unit* variables, and three possibilities for the *region* variable gives twelve data sets. The change in yield, the class attribute, was discretized to tertiles ("high" (H), "medium" (M), and "low" (L) change) as it was in Nelson and Congdon's work. The non-class attributes, however, were discretized into quartiles, after some preliminary tests showed them to be preferable to tertiles, quintiles, and sextiles.

I determined good parameter values for the LCS both by experimenting and by consulting the literature. I maintained the values of *elitism_rate* (0.6), *p_mutate* (0.25), and *p_dont_care* (0.3) used on the Iris and Wine sets, but increased *pop_size* to 300 and *num_iters* to 2500.

4.2.1 Holdout Method

In the holdout tests, the data sets were split into one training and one test set of approximately equal size. Ten tests were run per data set per class. The mean and variance of the accuracy, the odds ratio, and the number of rules used in the classification, for both the training and the test sets, are recorded in table [#].

4.2.2 Ten-Fold Cross Validation

In a ten-fold cross validation, the data are divided into ten sets. In a single run, the LCS is trained on nine of the sets and then tested on the tenth.

The process is repeated for all ten sets. Tables [#] through [#] show the results of these experiments.

5 Future Work and Conclusion

References

[1] O. US EPA, "Climate Impacts on Agriculture and Food Supply," 2017.

[2] "World population projected to reach 9.7 billion by 2050 | UN DESA | United Nations Department of Economic and Social Affairs," July 2015.

[3] E. Nelson and C. B. Congdon, "Measuring the Relative Importance of Different Agricultural Inputs to Global and Regional Crop Yield Growth Since 1975," *Economics Department Working Paper Series*, Sept. 2016.

[4] T. Kovacs, "Genetics-Based Machine Learning," in *Handbook of Natural Computing* (G. Rozenberg, T. Bck, and J. N. Kok, eds.), pp. 937–986, Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-540-92910-9_30.

[5] J. H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*, vol. viii. Oxford, England: U Michigan Press, 1975.

[6] J. H. Holmes, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, "Learning classifier systems: New models, successful applications," *Information Processing Letters*, vol. 82, pp. 23–30, Apr. 2002.

[7] J. H. Holland, "Properties of the Bucket Brigade," in *Proceedings of the 1st International Conference on Genetic Algorithms*, (Hillsdale, NJ, USA), pp. 1–7, L. Erlbaum Associates Inc., 1985.

[8] W. C. J. C. H., *Learning from Delayed Rewards.* PhD thesis, 1989.

[9] A. Orriols-Puig, J. Casillas, and E. Bernado-Mansilla, "Fuzzy-UCS: A Michigan-Style Learning Fuzzy-Classifier System for Supervised Learning," *IEEE Transactions on Evolutionary Computation*, vol. 13, pp. 260–283, Apr. 2009.

[10] X. Llor, R. Reddy, B. Matesic, and R. Bhargava, "Towards Better Than Human Capability in Diagnosing Prostate Cancer Using Infrared Spectroscopic Imaging," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, (New York, NY, USA), pp. 2098–2105, ACM, 2007.

[11] Y. Gao, J. Z. Huang, H. Rong, and D. Gu, "Learning Classifier System Ensemble for Data Mining," in *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation*, GECCO '05, (New York, NY, USA), pp. 63–66, ACM, 2005.

[12] Hai H. Dam, Hussien A. Abbass, and Chris Lokan, "BCS: Bayesian Learning Classifier System," Tech. Rep. TR-ALAR-200604005, The Articla and Adaptive Robotics Laboratory, School of Information Technology and Electrical Engineering, University of New South Wales, Cardiff, UK, 2006.

[13] S. W. Wilson, "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, vol. 3, pp. 149–175, June 1995.

[14] E. Bernad-Mansilla and J. M. Garrell-Guiu, "Accuracy-based learning classifier systems: models, analysis and applications to classification tasks," *Evolutionary Computation*, vol. 11, no. 3, pp. 209–238, 2003.

[15] P. L. Lanzi, "A Study of the Generalization Capabilities of XCS," in *In Proceedings of the Seventh International Conference on Genetic Algorithms*, pp. 418–425, Morgan Kaufmann, 1997.

[16] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer, "Evolving Objects: A General Purpose Evolutionary Computation Library," in *Artificial Evolution*, pp. 231–242, Springer Berlin Heidelberg, Oct. 2001. DOI: 10.1007/3-540-46033-0_19.

[17] R. J. Urbanowicz and J. H. Moore, "Learning Classifier Systems: A Complete Introduction, Review, and Roadmap," *J. Artif. Evol. App.*, vol. 2009, pp. 1:1–1:25, Jan. 2009.

[18] P. L. Lanzi and R. L. Riolo, "A Roadmap to the Last Decade of Learning Classifier System Research (From 1989 to 1999)," in *Learning Classifier Systems* (P. L. Lanzi, W. Stolzmann, and S. W. Wilson, eds.), no. 1813 in Lecture Notes in Computer Science, pp. 33–61, Springer Berlin Heidelberg, 2000. DOI: 10.1007/3-540-45027-0_2.

[19] S. W. Wilson and D. E. Goldberg, “A Critical Review of Classifier Systems,” in *Proceedings of the Third International Conference on Genetic Algorithms*, (San Francisco, CA, USA), pp. 244–255, Morgan Kaufmann Publishers Inc., 1989.

[20] S. W. Wilson, “State of XCS Classifier System Research,” in *Learning Classifier Systems* (P. L. Lanzi, W. Stolzmann, and S. W. Wilson, eds.), no. 1813 in Lecture Notes in Computer Science, pp. 63–81, Springer Berlin Heidelberg, 2000. DOI: 10.1007/3-540-45027-0_3.

[21] J. D. Schaffer and J. J. Grefenstette, “Multi-objective Learning via Genetic Algorithms,” in *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI’85, (San Francisco, CA, USA), pp. 593–595, Morgan Kaufmann Publishers Inc., 1985.

[22] Manuel Valenzuela-Rendn, “The Fuzzy Classifier System: A Classifier System for Continuously Varying Variables”,” (San Diego, CA), 1991.

[23] J. H. Holland and J. S. Reitman, “Cognitive Systems Based on Adaptive Algorithms,” *SIGART Bull.*, pp. 49–49, June 1977.

[24] B. J, “Reducing Bias and Inefficiency in the Selection Algorithm,” in *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, (Hillsdale, New Jersey), pp. 14–21, L. Eerlbaum Associates, 1987.

[25] Tobias Blickle and Lothar Thiele, “A Comparison of Selection Schemes used in Genetic Algorithms,” Tech. Rep. 11, Computer Engineering and Communication Networks Lab, Swiss Federal Institute of Technology (ETH), Zurich, Dec. 1995.

[26] S. Hatthon, “A diagram illustrating the Stochastic Universal Sampling operation. In this illustration 4 individuals are being selected (N=4), resulting in the selection of A, B, C and F.,” Jan. 2007.