THE TSC TIME SERIES CLASSIFIER

by

CHRISTOPHER MARK GORE

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2008

Approved by

_____   _____
Daniel R. Tauritz, Advisor   Ralph W. Wilkerson

_____
Ray Luechtefeld

ABSTRACT

A time series is a sequence of data measured at successive time intervals. Time series analysis refers to all of the methods employed to understand such data, either with the purpose of explaining the underlying system producing the data or to try to predict future data points in the time series. Time series analysis is applicable to many problems since there are so many areas that require a more thorough understanding of a time series or the prediction of future values of the time series. The most typical historical examples of time series would be the weather and the financial markets but there are many more real-world time series problems.

An evolutionary algorithm is a non-deterministic method of searching a solution space, and modelled after biological evolutionary processes. A learning classifier system (LCS) is a form of evolutionary algorithm that operates on a population of mapping rules. We introduce the time series classifier *TSC*, a new type of LCS that allows for the modeling and prediction of time series data, derived from Wilson's XCSR, an LCS designed for use with real-valued inputs. Our method works by modifying the makeup of the rules in the LCS so that they are suitable for use on a time series. All of the operations (mutation, crossover, etc.) applied to the rules also were changed from their traditional forms.

We tested TSC on real-world historical stock data. The system would always return a profit, but not as much as the stock market itself is capable of returning by the utilization of an indexing fund. The stock market is a notoriously difficult system to model effectively and therefore any positive results at all are notable, and never losing money in the long-term is impressive in itself, often a difficult task for unskilled human traders.

Although this initial system appears incapable of producing monetary returns better than that of the stock market itself and may not be the eventual solution, it does perform well enough to demonstrate that the system is capable of learning in a very complex environment. The inherent complexity of the market makes the system unusable for automated trading, but this approach should prove to be useful in other less challenging real-world time series problems.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ALGORITHMS

1. INTRODUCTION

This thesis considers applying learning classifier systems (LCS's) to the prediction of time series data. A time series as used here is a sequence of data successively measured through time. Time series analysis encompasses many methods that attempt to understand such time series, aimed at either understanding the underlying theory present in the data points or to make real-world predictions. Time series prediction is the use of a model to predict future events based on known past events: to predict future data points before they are measured. One standard example is the opening price of a share of stock based on its past performance.

1.1. MOTIVATION

No LCS to date has been designed for time series data but instead they were generally limited to Markov systems lacking any memory, which we viewed as a major limitation of LCS's. LCS's are designed specifically with the concept of evolving an effective rule set for a specified problem, which is specifically the sort of capability that would be desirable for time series analysis and prediction: generating useful rule sets.

An LCS is an evolutionary algorithm that operates on a population comprised of rules referred to as the rule set: this rule set is used to attempt to classify a situation. The first LCS was created by Holland [1] shortly after he created genetic algorithms (GA's) [2], one of the classical types of evolutionary algorithms. Holland's first LCS originally used a GA as the evolutionary device. Our system as described here also uses a GA for evolution, although it has been modified from the original form.

Holland's original LCS was quite complicated and failed to produce quality results for most real-world problems. Because of this, the study of LCS's was somewhat inactive until Wilson introduced ZCS [3], a re-imagining of Holland's original LCS distilled to its most basic elements. Wilson's ZCS was capable of producing acceptable results on certain problems and was simple enough to easily understand, reinvigorating LCS research.

A few years after introducing ZCS, Wilson modified it introducing XCS [4], which is

currently one of the best performing and most popular LCS types. Wilson's XCS was based on ZCS but with several important modifications mostly aimed at improving the accuracy of the rules produced and also for a more full coverage of the problem space by the rules. A significant portion of the LCS's being worked on today are modifications or enhancements of Wilson's XCS.

One such enhancement of XCS is known as XCSR [5], which was also developed by Wilson. XCSR improves upon XCS by allowing it to operate with real-valued ranges for input instead of on the traditional ternary alphabet so common to LCS's, consisting of *true*, *false*, and a covering symbol (usually represented as # or ∗).

1.2. BACKGROUND

The system presented here is derived from Wilson's XCSR, which is an extension of Wilson's XCS, which in turn was derived from Wilson's ZCS. ZCS, XCS, XCSR, and this system are all learning classifier systems (LCS's), a crossover of the fields of evolutionary computation (EC) and reinforcement learning (RL), both of which are quite large fields on their own. We will describe in this section the previous works this system was built upon.

1.3. REINFORCEMENT LEARNING

Reinforcement learning [6] is the process of learning how to map situations to actions to maximize a numerical reward. The learning system is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by exploration. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. The two primary distinguishing characteristics of reinforcement learning are:

1. trial-and-error search and

2. delayed reward.

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem. We consider any method that is well suited to solving that problem to be a reinforcement learning method. The idea is to capture the most important aspects of the problem facing the learning agent interacting with its environment to achieve its goal. Such an agent must be able to:

1. perceive the state of the environment,

2. act on the environment, and

3. have a goal or goals relating to the state of the environment.

Tersely put: *sensation*, *action*, and *goal*.

   1.3.1. Exploration versus Exploitation.   A primary challenge is the trade-off between exploration and exploitation. A reinforcement learning agent will prefer actions that it has tried in the past and found to be effective in producing reward in order to reliably obtain more reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to *exploit* existing knowledge to obtain reward, but it also must *explore* to make better action selections in the future. Neither exploration nor exploitation can be pursued exclusively without failure. The agent must try a variety of actions and progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward.

   1.3.2. The Whole Problem.   Reinforcement learning explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment, starting with a complete, interactive, goal-seeking agent, instead of considering subproblems without addressing how they might fit into a larger picture. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. From the beginning, the agent operates with significant uncertainty about its environment. For learning research to make progress, important subproblems have to be isolated and studied, but they should be subproblems that play clear roles in complete, interactive, goal-seeking agents, even if all the details of the complete agent cannot yet be filled in.

1.4. EVOLUTIONARY ALGORITHMS

In artificial intelligence (AI), evolutionary algorithms (EA's) are a style of generic population-based meta-heuristic optimization algorithms whose processes are inspired by those of natural biological evolution. The primary mechanisms employed in EA's to evolve a population of possible solutions towards an optimal one are:

1. parent selection based on fitness,

2. recombination,

3. mutation, and

4. and survivor selection based on fitness.

Evolution serves as a powerful metaphor and demonstrates great creativity in both the natural world and in the world of computer science.

In normal biological evolution the environment that the population exists in exerts various pressures on the individuals in the population that determines the likelihood that any particular individual will manage to survive long enough to reproduce, and it is through this process that the fitness of an individual in the population must be determined: relative to its environment. In an EA, the environment relates to the problem we wish to solve, the individuals in the population encode potential solutions to that problem, and their fitness is their quality as a solution to the problem. By mimicking the methods of natural evolution in this manner we can often arrive at good solutions. The basic evolutionary process is outlined in Algorithm 1.1.

---

1. Initialize the population, either with randomly-generated or seeded candidate solutions or both.
2. Evaluate the fitness of each member of the population.
3. **repeat**
4.   Select members of the population to act as parents. This is typically related to the relative fitness of the parents in some way.
5.   Recombine the genetic material of the parents, producing offspring to be added to the population.
6.   Mutate some or all of the newly-created offspring.
7.   Evaluate the fitness of the offspring.
8.   Select survivors from the current population or a subset thereof, often only the newly-created offspring, to survive to the next generation.
9. **until** some specified termination condition is satisfied.

Algorithm 1.1. The evolutionary process.

1.4.1. Learning Classifier Systems. A learning classifier system is a type of EA in which a description of a current situation is used in an attempt to map that description to

some classification or action. This is achieved through simulated evolutionary processes, where the population being evolved consists of various rules; our entire population forms a rule set, and we apply concepts from Darwinism to our individual rules. This is known in learning classifiers as the *Michigan approach* [7]. The other primary method employed, where each individual is an entire solution, and therefore a whole rule set, is known as the *Pittsburgh approach*. We use a modification of XCSR here, which uses the Michigan approach, and therefore so do we.

1.4.2. ZCS. ZCS is a *zeroth level classifier system* originally proposed in [3]. ZCS preserves most of the functionality of traditional LCS's, but it is a very simplified version, which aids in the understanding of the classifier and its actions. This was a very useful contribution, because many of the problems with LCS's before then were their overly complex and detailed nature. A good short summary of ZCS can be found in [7], and this summary is based primarily on that. The basic structure of ZCS is graphically illustrated in Figure 1.1.

In ZCS there is no message list, a much-welcomed simplification of the traditional LCS. This comes with a cost: there is no explicit method for transmitting information between cycles without the message list. This makes the interface entirely dependent on the interface of the system with its environment, and thus assumes a Markov process. This is most definitely an invalid assumption for real-world traded markets and for other time-series data.

Each rule r is of the form $r = (c, a, s)$ where:

- c is the condition matched by the rule r and is comprised of elements from some alphabet, typically $\{0, 1, \#\}$, where # is the matching symbol, matching both 0 and 1;

- a is the action that the rule r recommends;

- and s is the real-valued strength measurement of the rule r, $s \in \mathbb{R}$, which determines how much of a vote rule r has in selecting the action to pursue.

In each time cycle t the match set M_t is found, a subset of the population, $M_t \subseteq P$, with P being the entire population of rules, the rule set. The members at time cycle t of the match set M_t can be divided into disjoint subsets based on the action they recommend.
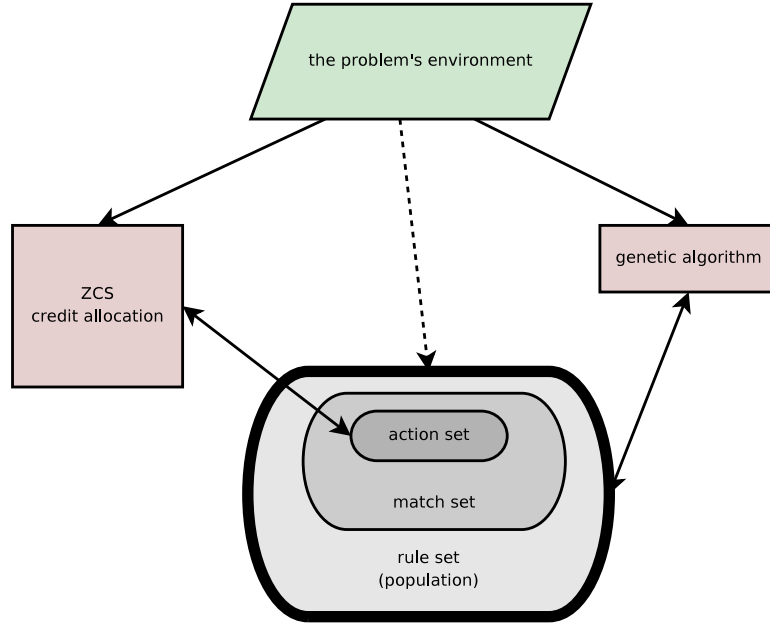
Figure 1.1. ZCS's basic structure

With a finite set of possible actions

$$\mathcal{A} = \{a_0, a_1, \ldots, a_{|\mathcal{A}|}\} \tag{1}$$

and $\mathcal{A}' \subseteq \mathcal{A}$ where

$$\mathcal{A}' = \{a'_0, a'_1, \ldots, a'_{|\mathcal{A}'|}\} \tag{2}$$

comprises all of the actions represented in the match set M_t. For any specific action a'_i represented in the match set M_t we can form the set of all members of the match set that recommend action a'_i, represented as $M_{t,a'_i} \subseteq M_t$ with

$$M_{t,a'_i} = \left\{ r \colon r \in M_t \wedge a_r = a'_i \right\}. \tag{3}$$

The fitness of an action $a'_i \in \mathcal{A}'$ is then

$$\text{fitness}(a'_i) = \sum_{\forall r \in M_{t,a'_i}} s_r, \tag{4}$$

the sum of the fitness of all of the rules r that recommend that particular action present in M_{t,a'_i}. The action to take is selected in a fitness-proportionate method, choosing the action a' with the greatest fitness. If $M_t = \emptyset$ then covering must take place; a random rule that matches the current situation is created by initially setting c to exactly the current situation and then replacing a few elements of c at random with the # symbol, and that suggests a randomly-selected action.

The credit assignment scheme used by ZCS is somewhat involved, and is referred to as an *implicit bucket brigade*. It attempts to reward sequences which lead to reward from the environment and which are short. First, the rules in the population P but excluded from the match set M_t are originally unchanged:

$$s'_r = s_r \forall r \notin M_t. \tag{5}$$

Next, the rules in the match set M_t but excluded from the action set A_t (those advocating weaker actions than the one chosen) have their strengths reduced by a factor $\tau \in [0,1)$:

$$s'_r = \tau s_r \forall r \in M_t \setminus A_t. \tag{6}$$

Then the strength of the rules in the current action set A_t have a fraction $\beta \in [0,1)$ of their strengths transferred to the members of the previous action set A_{t-1}, reduced by a factor $\gamma \in [0,1)$:

$$s'_r = (1 - \beta)s_r \forall r \in A_t, \tag{7}$$

$$s''_r = s'_r + \frac{\gamma \sum_{\forall r \in A_t} \beta s_r}{|A_{t-1}|} \forall r \in A_{t-1}. \tag{8}$$

Finally, any feedback P_t from the environment is reduced by β and distributed to the rules in the current action set A_t:

$$s'''_r = s''_r + \frac{\beta P_t}{|A_t|} \forall r \in A_t. \tag{9}$$

A mostly standard GA is run on the population (the rule set) periodically, with parent selection directly related to s and death selection inversely related s. The new rules are usually assigned the mean of their parents' strength initially.

1.4.3. XCS. ZCS has many positive features, especially its simplicity and the benefits derived from its cooperative fitness sharing, but there are some notable drawbacks,

primarily that it usually will not evolve a complete mapping of the environmental states and allowable actions to the possible rewards, often quickly selects local optima, and breeds across niches, as noted in [4]. These drawbacks led Wilson to heavily modify ZCS into what is called XCS [4]. In XCS, several of the deficiencies in ZCS are addressed. The basic structure of XCS is graphically illustrated in Figure 1.2.

In ZCS, the GA is run on the entire population, a *panmictic* approach [7, p. 155]. This is ineffective for most problems, so in XCS the GA was run only in the current match set at the time step that the GA is run in the initial version of XCS, and only in the current action set at the time step that the GA is run in the later variants of XCS. We run the GA on the current action set in this work. This allows for a more accurate rule set to be evolved, since each niche is best viewed as its own sub-problem.

In ZCS, a rule is allowed to survive by the GA on the basis of its payoff. This is problematic, since it biases against rules early in a chain of events that are eventually profitable, and because rules that may be the most appropriate for an event might have a relatively low payoff. This caused ZCS to often fail to create a complete mapping and fail to evolve accurate generalizations. This is remedied in XCS by creating a fitness measure for the rules, separate from the predicted payoff, used by the GA.

Each rule r is now of the more complex form

$$r = (c, a, p, \varepsilon, F, exp, ts, as, n), \tag{10}$$

where:

- c is the condition matched by the rule r, comprised of elements from some alphabet such as $\{0, 1, \#\}$, where # is the matching symbol, matching both 0 and 1.

- a is the action that the rule r recommends.

- p is the predicted payoff.

- ε is an estimate of the prediction error.

- F is the fitness used by the GA. It is vital that the fitness used by the GA is a measure of the *accuracy* of the rule, and not a measure of the *magnitude* of the rule, where the magnitude of a rule is how active that rule is in relation to the rest of the rules in the
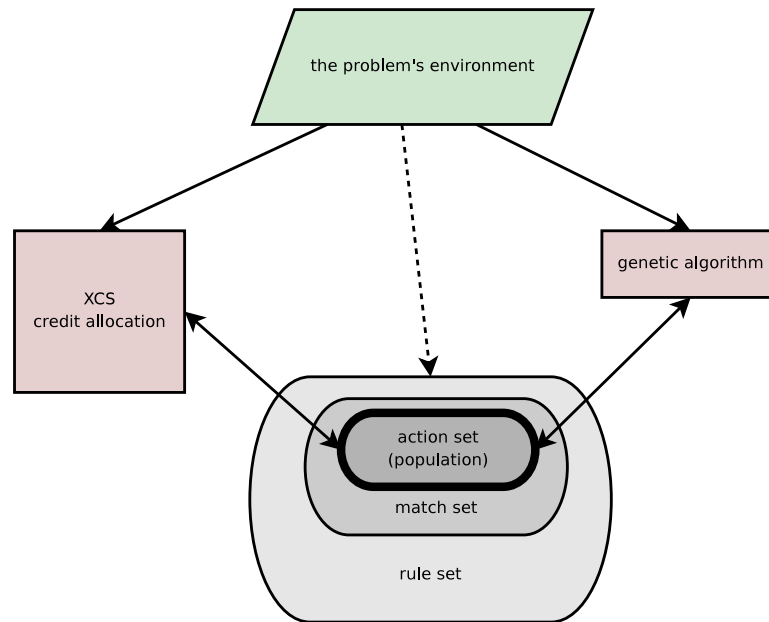
Figure 1.2. XCS's basic structure

rule set, since a rule with greater magnitude but lower accuracy can be a detriment to the system. For example, a rule that always matched every situation (all #'s in the condition) but only accurately predicted 51% of the situations would have high magnitude but low accuracy.

- *exp* is the experience of the rule, a count of the number of times since this classifier's creation that it has belonged to the action set.

- *ts* is a time stamp of the last occurrence of a call to the GA in an action set that this classifier was a part of, as the generational number.

- *as* is an estimate of the average action set size this classifier has belonged to.

- *n* is the numerosity of this macro-classifier. This is how many traditional micro-classifiers this macro-classifier represents. Groups of entirely identical normal classifiers (the micro-classifiers) are subsumed into macro-classifiers instead of being allowed to exist separately within the rule set; this serves solely as a computational

time-saver. Therefore the only difference between a normal classifier (a micro-classifier) and a macro-classifier is the presence of the numerosity, which is a count of how many micro-classifiers that specific macro-classifier represents.

1.4.4. XCSR. Wilson extended his concept of XCS with XCSR in [5]. Classifier systems had typically taken strings from some small alphabet, often binary, as input until then even though many real-world problems have input from the environment of the form $\mathbb{R}^n$ for some order $n \in \mathbb{Z}, n > 0$. Wilson's XCSR allows XCS to operate on just such an input. XCSR is identical to normal XCS with the exception of the input interface, the nature of the predicates, the mutation operator, and the details of covering. The basic structure of an XCSR rule is graphically illustrated in Figure 1.3.

Originally the predicates in XCSR were intervals of the form

$$interval_i = \{center_i, spread_i\}, \tag{11}$$

such that an environmental input x_i was matched by $interval_i$ if and only if

$$center_i - spread_i \leq x_i \leq center_i + spread_i, \tag{12}$$

but this was discovered to induce a bias [8], so the representation was eventually changed to be

$$interval_i = \{lower_i, upper_i\}, \tag{13}$$

where now x_i is matched by $interval_i$ if and only if

$$lower_i \leq x_i \leq upper_i. \tag{14}$$

We use the $\{lower, upper\}$ form in this work.

Crossover is simple two-point crossover, but on the sequence

$$\{center_0, spread_0, \ldots\} \tag{15}$$

or

$$\{lower_0, upper_0, \ldots\} \tag{16}$$
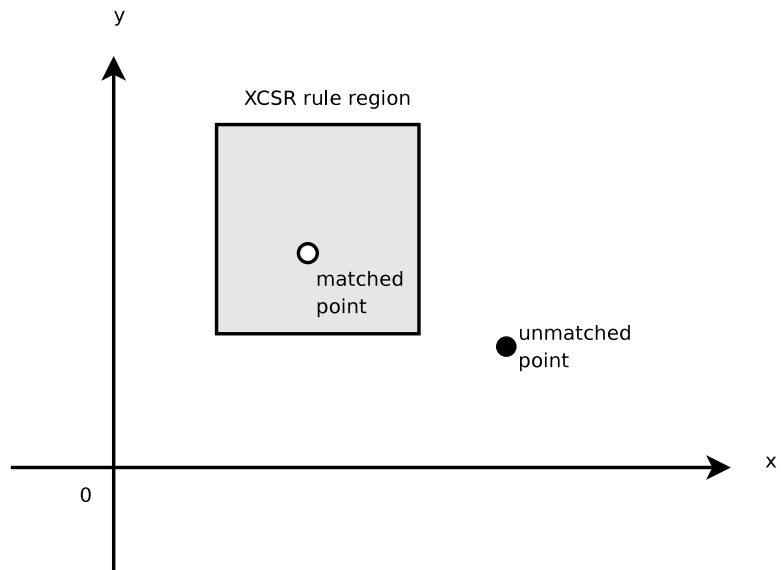
Figure 1.3. XCSR's interval rules

depending on the predicate type, in both cases therefore allowing the crossover points to fall within a single allele.

In the original XCSR, mutation was performed by adding a small random quantity from the range $[-0.1, 0.1]$ to each allele, and all problems were to have their input scaled to $[0, 1]$. The variation of XCSR used here is capable of scaling outside of $[0.0, 1.0]$, so instead mutation is performed as the addition or subtraction of a small percentage of the overall range as seen so far.

2. TIME SERIES PREDICTION

2.1. ARIMA AND OTHER STATISTICAL METHODS

ARIMA, the *autoregressive integrated moving average*, is a common and very powerful statistical method often used in econometric models that can help forecast and estimate what is going to happen in the future. The ARIMA time series analysis uses lags and shifts in the historical data to uncover patterns (e.g., moving averages, seasonality) and predict the future [9]. The ARIMA model was first developed in the late 1960s but was not systemized until the work of Box and Jenkins in 1976 [10]. ARIMA can be more complex to use than other statistical forecasting techniques, although when implemented properly can be quite powerful and flexible. ARIMA is a method for determining two things:

1. how much of the past should be used to predict the next observation (length of weights) and

2. the values of the weights.

Three common models of time series data are *autoregressive* (AR) models, the *integrated* (I) models, and the *moving average* (MA) models. These three classes depend linearly on previous data points and are combined in the autoregressive integrated moving average (ARIMA) model. A model of this form is referred to as an $ARIMA(p,d,q)$ model where $p,d,q \in \mathbb{N}^*$. The order of the autoregressive part is p, the order of the integrated part is d, and the order of the moving average part is q. Given a time series of data X_t (where t is integer valued and the X_t are real numbers) then an $ARIMA(p,d,q)$ model is given by

$$\left(1 - \sum_{i=1}^{p} \phi_i L^i\right)(1-L)^d X_t = \left(1 + \sum_{i=1}^{q} \theta_i L^i\right)\varepsilon_t \tag{17}$$

where L is the lag operator, ϕ are the parameters of the autoregressive part of the model, θ are the parameters of the moving average part, $d \in \mathbb{N}^*$ (if instead we have $d = 0$ then this model is equivalent to an ARMA model), and the ε_t are error terms. The error terms ε_t are generally assumed to be independent and identically distributed variables sampled from a

normal distribution with zero mean: $\varepsilon_t \sim N(0, \sigma^2)$ where σ^2 is the variance. ARIMA models are commonly used for predicting and analyzing simpler time series. They have been used on the stock market, but are generally viewed only as an indicator, not a predictive tool, due to the complexity of the market and because of their need for accurate knowledge about the time series itself. It is for similar reasons that most traditional statistical methods fail to be of any real use in this task.

For example

$$y(t) = \frac{y(t-3)}{3} + \frac{y(t-2)}{3} + \frac{y(t-1)}{3} \tag{18}$$

is a potential ARIMA model; another potential ARIMA model is

$$y(t) = \frac{y(t-3)}{6} + \frac{4y(t-2)}{6} + \frac{y(t-1)}{6}. \tag{19}$$

The correct ARIMA model requires identification of the right number of lags and the coefficients that should be used. ARIMA model identification uses autoregressions to identify the underling model. Care must be taken to robustly identify and estimate parameters as outliers (pulses, level shifts, local time trends) can wreak havoc.

2.2. ARTIFICIAL NEURAL NETWORKS

An artificial neural network is a graph of connected processing elements called neurons which can exhibit complex global behavior as determined by the connections between the neurons and their parameters. This technique was originally inspired by the examination of the central nervous systems of living creatures, most notably that of humans, the most significant information processing system found in nature. While a neural network is not adaptive itself, most practical examples use algorithms designed to alter the weights of the connections in the network to produce a desired signal flow. These networks are also similar to their biological counterparts in that their functions are performed collectively in parallel by the entire network, with no clear delineation of sub-tasks to which various units are assigned. Modern artificial neural networks often abandon much of this for a more practical approach based on statistics and signal processing [11]. There have been many attempts to predict financial time series with artificial neural networks [12, 13], and there have even been somewhat successful results using genetic algorithms to evolve the weights for neural networks [14, 15]. However, there is one main drawback that comes with the

use of artificial neural networks. There is no easy way to translate the neural network that has been produced into an understandable set of rules describing its innate knowledge: the information is effectively trapped in the weights on the neurons. Extracting useful rules from ANN's is a challenging field unto itself [16].

2.3. NON-LCS EVOLUTIONARY APPROACHES

There have been attempts at using evolutionary approaches other than LCS's to predict and analyze markets and other time series, ranging from the simplistic to the very complex. In [17], traditional genetic algorithms were used to optimize the exact numbers to be used in traditional technical analysis. In [18], traditional genetic algorithms were again used, but this time in optimizing the rule sets for candlestick-style analysis; this outperformed a random trader. In [19], a simplified variant on the concept of genetic programming, coded in C++, was used to develop trading rules for six stocks, and they managed to return better results than both the market and a naive trader. However, the innate challenges of the real market have lead many researchers to resort to simulated markets, whose simplicity can make fundamental discoveries about economic theory sometimes less challenging to achieve; a small survey of these sorts of markets can be found in [20].

2.4. LCS-BASED APPROACHES

There have been a few attempts at using LCS's to analyze and predict financial markets. We will highlight a few derived from XCS here, since the system presented here is also derived from XCS.

2.4.1. XCS. A predictive system lacking a memory component is almost completely useless in attempting to model a highly interdependent nonlinear multivariate time series such as the stock market with any hope of utility; none the less, it has been attempted. One of the more notable attempts at this is described in [21], in which an XCS was used to predict the correct trading action for a stock on consecutive trading days. Later work by Schulenburg and Ross in [22] does show some promise: they utilize the opinions of a large host of simulated traders in order to make a decision. This would yield in the general vicinity of 9%p.a. returns: not spectacular or applicable to real-world trading, but respectable.

2.4.2. XCSF. In [23] Wilson outlined an extension to XCS for the approximation

of functions, called XCSF, which attempts to learn a function of the form $y = f(x)$, where $y \in \mathbb{R}$, $|x| = n$, and $x_i \in \mathbb{Z} \forall x_i \in x$. A classifier consists of n interval predicates of the form $int_i = (l_i, u_i)$ and matches an input x if and only if $l_i \leq x_i \leq u_i \forall i \in \mathbb{N}$. Classical two-point crossover is employed, but where crossover may occur in-between the alleles or at the ends of the prediction, although the action is not involved in the crossover process. A covering classifier is generated for a situation x by forming the l_i through subtracting from x_i some random integer from $[0, r_0]$, and forming u_i by adding some other random integer from $[0, r_0]$ to x_i, both limited to a maximum range of possible input, where r_0 is a parameter. A rule r^1 can subsume a rule r^2 if and only if $l_i^1 \leq l_i^2 \wedge u_i^2 \leq u_i^1 \forall i$. While this could possibly be used to predict some very simplistic time series data, function approximation often does not perform very well in real-world problems, as is well-known in reinforcement learning literature [24, 25], and this drawback of XCSF (and similar approaches) is explicitly acknowledged in [26]. This would be most definitely true of a system as complex as the stock market, which cannot be easily and usefully mapped to any polynomial.

3. APPROACH AND DESIGN OF THE TIME SERIES CLASSIFIER

3.1. FUNDAMENTAL OPERATIONS

Our representation of a time series and our approach to their evolutionary methods requires us to be capable of generating multi-dimensional raster paths, where a raster path is a one-dimensional path through a raster space. This is so that we can run raster paths through a raster space of data, a discrete sampling of data. A raster space is one that is representable by $\mathbb{Z}_a \times \mathbb{Z}_b \times \cdots \times \mathbb{Z}_z$. In other words, all of the dimensions are along sets of finite integers instead of the real numbers. A common example is raster imagery: a two-dimensional bitmap of size $m \times n$ can be viewed as a complete representation of the two-dimensional raster space of $\mathbb{Z}_m \times \mathbb{Z}_n$. A multidimensional matrix can therefore fully represent these spaces, instead of merely being samplings of the real space, although we are using these raster spaces for sampling of real data in our approach. We form a useful sample of the data for further analysis and classification by TSC by generating paths through the data and it is these raster paths that the TSC actually classifies the situations with, not with the entire data set which is generally very large. We will now outline the basic operations we use to generate raster lines.

3.1.1. The *Sort On* Algorithm. This algorithm sorts a sequence s according to the ordering of another sequence t, and is outlined in Algorithm 3.1.

Input: A sequence s to be sorted.
Input: A sequence t upon which to sort s with.
Input: A comparator c to sort with, typically $>$ or $<$.
Require: $|s| = n \leq |t|$.
 1. Construct a sequence u containing pairs of the form $u_i = (s_i, t_i)$ as elements, $|u| = n$,

$$u = (u_0, \ldots, u_{n-1}) = ((s_0, t_0), \ldots, (s_{n-1}, t_{n-1})). \tag{20}$$

 2. Sort u using the second elements as the key, using any normal sorting algorithm, giving

$$u' = ((s'_0, t'_0), \ldots, (s'_{n-1}, t'_{n-1})) \tag{21}$$

where $t'_0 \leq \ldots \leq t'_{n-1}$ if we are sorting in ascending order (with the $<$ comparator).
 3. **return** $s' = (s'_0, \ldots, s'_{n-1})$.

Algorithm 3.1. Sort on.

3.1.2. The *Sort Order* Algorithm.

This algorithm returns the re-ordered indices of a sorted sequence, and is outlined in Algorithm 3.2. For example, if $t = \{4, 5, 3, 9\}$ then the sorted ordering of t would be $\{3, 1, 0, 2\}$ since $t_3 \geq t_1 \geq t_0 \geq t_2$.

Input: A sequence t.
Input: A comparator c, usually $<$ or $>$.
 1. **let** $n \leftarrow |t|$.
 2. Generate $\mathbb{Z}_n = (0, \ldots, n-1)$.
 3. **return** The result of the *sort on* algorithm from §3.1 on $s = \mathbb{Z}_n$ with t and the comparator c.

Algorithm 3.2. Sort order.

3.1.3. Rasterized Linear Paths Through Arrays.

Given an array A of rank r and dimensions $d_0 \times \cdots \times d_{r-1}$, we wish to pull a one-dimensional list or vector v of values from the array, starting at position $A_{s_0 \ldots s_{r-1}}$ and finishing at position $A_{f_0 \ldots f_{r-1}}$, following a linear path through the array.

As an example consider the 4×6 array:

$$A = \begin{pmatrix} a & b & c & d & e & f \\ g & h & i & j & k & l \\ m & n & o & p & q & r \\ s & t & u & v & w & x \end{pmatrix}.$$

3.1.3.1. A purely horizontal path. The linear path from A_{00} to A_{05} would be composed of the values

$$\langle A_{00}, A_{01}, A_{02}, A_{03}, A_{04}, A_{05} \rangle$$

and would be

$$\langle a, b, c, d, e, f \rangle$$

as illustrated by

$$A = \begin{pmatrix} a^\star & b^\star & c^\star & d^\star & e^\star & f^\star \\ g & h & i & j & k & l \\ m & n & o & p & q & r \\ s & t & u & v & w & x \end{pmatrix}.$$

3.1.3.2. A purely vertical path. The linear path from A_{00} to A_{30} would be composed of the values

$$\langle A_{00}, A_{10}, A_{20}, A_{30} \rangle$$

and would be

$$\langle a, g, m, s \rangle$$

as illustrated by

$$A = \begin{pmatrix} a^\star & b & c & d & e & f \\ g^\star & h & i & j & k & l \\ m^\star & n & o & p & q & r \\ s^\star & t & u & v & w & x \end{pmatrix}.$$

3.1.3.3. A traditional diagonal path. The linear path from A_{00} to A_{33} would be composed of the values

$$\langle A_{00}, A_{11}, A_{22}, A_{33} \rangle$$

and would be

$$\langle a, h, o, v \rangle$$

as illustrated by

$$A = \begin{pmatrix} a^\star & b & c & d & e & f \\ g & h^\star & i & j & k & l \\ m & n & o^\star & p & q & r \\ s & t & u & v^\star & w & x \end{pmatrix}.$$

3.1.3.4. Non-equal diagonal paths. The confusing part arises when we are dealing with diagonal paths with unequal steps. Consider the linear path from A_{00} to A_{35}. We end up with a stair-stepping path through the array:

$$(A_{00}, A_{11}, A_{21}, A_{32}, A_{42}, A_{53})$$

and would be

$$(a, h, i, p, q, x)$$

as illustrated by

$$A = \begin{pmatrix} a^\star & b & c & d & e & f \\ g & h^\star & i^\star & j & k & l \\ m & n & o & p^\star & q^\star & r \\ s & t & u & v & w & x^\star \end{pmatrix}.$$

3.1.3.5. The Raster Line Algorithm. This is the algorithm used to determine a linear raster path, and is outlined in Algorithm 3.3. It returns a list of points that follow the linear path from the starting point p to the ending point q. This is derived from the algorithm for raster conversion of a 3D line as described in [27]. This should work for any dimensionality.

Input: a starting point p and a final point q, both represented as lists.
Require: $|p| = |q| \wedge p_i \in \mathbb{N} \forall p_i \in p \wedge q_i \in \mathbb{N} \forall q_i \in q$.
 1. **if** $p = q$ **then** *// This is a simple degenerate case.*
 2.   **return** $\{p\}$, a list containing only one element, p.
 3. **let** $n \leftarrow |p| = |q|$ be the dimensionality.
 4. **let** $\delta \leftarrow \{|p_0 - q_0|, \ldots, |p_{n-1} - q_{n-1}|\}$, $|\delta| = n$.
 5. **let** o be the sorted ordering of δ by $>$ from the *sort order* algorithm in §3.2.
 6. **let** p' and q' be p and q respectively, sorted according to o.
 7. **if** $p'_0 \leq q'_0$ **then** *// We want the starting point to have the lower initial dimension.*
 8.   Swap p' with q'.
 9. **let** $\delta' \leftarrow \{|p'_0 - q'_0|, \ldots, |p'_{n-1} - q'_{n-1}|\}$.
10. **let** $s \leftarrow \left(\text{sgn}\left(p'_0 - q'_0\right), \ldots, \text{sgn}\left(p'_{n-1} - q'_{n-1}\right)\right)$, where sgn is the signum function.
11. **let** $d \leftarrow \{d_1, \ldots, d_{n-1}\}$, $|d| = n - 1$, the deciders, where $d_i \leftarrow 2\delta'_i - \delta'_0 \forall d_i \in d$.
12. **let** $a \leftarrow \{a_1, \ldots, a_{n-1}\}$, $|a| = n - 1$, the if-increments, $a_i \leftarrow 2\delta'_i \forall a_i \in a$.
13. **let** $b \leftarrow \{b_1, \ldots, b_{n-1}\}$, $|b| = n - 1$, the else-increments, $b_i \leftarrow 2\left(\delta'_i - \delta'_0\right) \forall b_i \in b$.
14. **let** $r \leftarrow \{p'\}$, initializing the result of the algorithm, an ordered list of points.
15. **let** $z \leftarrow p'$, initializing the current point.
16. **while** $z_0 < q'_0$ **do** *// After this, we have $r = \{p', \ldots, q'\}$.*
17.   Increment z_0 by 1.
18.   **for all** $d_i \in d$ **do**
19.   **if** $d_i < 0$ **then**
20.   increment d_i by a_i.
21.   **else** *// In this case we have $d_i \geq 0$.*
22.   increment d_i by b_i and z_i by s_i.
23.   Push a duplicate of z to the back of r, so that now $r = \{p', \ldots, z\}$.
24. Reorder the coordinate of the points in r according to the original coordinate ordering forming r' by applying the inverse of o, which is o.
25. **if** we originally swapped the start and end points **then**
26.   **return** the reverse of r'.
27. **else**
28.   **return** r'.

Algorithm 3.3. Raster line.

3.1.4. List Slices. This function returns a slice from a one-dimensional list; that is, a modular subset of the list, and is outlined in Algorithm 3.4. For example, a 2-slice of the list $\{1,2,3,4,5,6,7,8,9\}$ would be the list $\{1,3,5,7,9\}$.

Input: A list of elements $l = \{l_0, l_1, \ldots, l_{|l|}\}$.
Input: A positive rational slice size s.
  1. Initialize the resulting list $r \leftarrow nil = \{\}$, initially empty.
  2. Initialize the moving index $i \leftarrow 0$.
  3. **while** $i < |l|$ **do**
  4.   **if** $i \in \mathbb{Z}$ **then**
  5.   Append l_i to the end of r.
  6. $i \leftarrow i + s$.
  7. **return** r.

Algorithm 3.4. List slice.

3.2. DATA REPRESENTATION

This LCS is intended to operate on a multivariate time series. The data consists of a single temporal dimension, several positional dimensions, and a single dimension of type. This is represented as a linked list consisting of multidimensional arrays, where each element in the matrices is a structure. Each array of structures represents a single time step; the position in the list is the position in time. The fields of the structures are independent data. Thus, any specific value in the multivariate time series could be uniquely referenced in the form:

$$\{t, x_0, \ldots, x_{n-1}, \phi\} \tag{22}$$

where t is the temporal position, $x_0, \ldots, x_{n-1}$ are the dimensional positions (for n dimensions), and ϕ is the field selector. It must hold that $\forall x_i \in \mathbb{N}^*$. The temporal position t specifies a time $t_{current} - t$, and it must also hold that $t \in \mathbb{N}_0$.

This representation can be simplified: the entries can be single elements instead of full structures, and the arrays themselves can even be reduced to single elements, reducing to a traditional one-dimensional time series, all using the same code. This is what is done in the examples here, and all tests were performed on one-dimensional time series, although each entry was a structure containing multiple related data. For our example of market analysis, t is the number of days from present time, and the fields are the opening price, closing price, high price, low price, adjusted closing price, and the volume of the trades for

that particular stock at that particular time.

3.3. RULE REPRESENTATION

The representation of a single rule is a collection of predicates; each predicate must match the current situation for the rule to match the situation. A single predicate consists of an initial and a final position, each of the form

$$\{t, x_0, \ldots, x_{n-1}\}, \tag{23}$$

a field selector ϕ, an operator ω, and a range pair consisting of a lower and upper bound $[l, u]$. The field selector ϕ is to be a lexical closure taking only one argument, which is the structure at the position $\{t, x_0, \ldots, x_{n-1}\}$. If the structure is not a structure, but rather a single element, the only value that would usually make sense for ϕ would be an identity function: simple transformative functions would be acceptable otherwise. Any function that operates in a uniform manner, applied to a single entry, would be an acceptable ϕ. The operator ω is also a lexical closure, and is intended for classification purposes; all ω's must operate over a one-dimensional vector of data.

If we take the data along the straight line segment from the initial point A to the final point B, forming a vector d, we can then form d' by applying ϕ to each element in d:

$$d'_i = \phi(d_i) \forall d_i \in d. \tag{24}$$

The predicate is said to match the data if and only if

$$l \leq \omega(d') \leq u. \tag{25}$$

When all of the predicates of the rule match, then the rule matches; the rule then recommends a particular classification or action.

3.4. MUTATION

The approach to mutation of the paths is to restrict the mutation of the line segment to the same line, only allowing the end points to move up or down along that line. In this method, the alteration of the line segment is minor, and therefore there is very little change in the

actual information held by the path. This is exactly the sort of effect we wish in mutation: small changes. By only allowing for smaller mutations we do not have the information stored in the rule itself destroyed completely, but instead it is just slightly modified.

The lower and upper values of the range are altered, but limited by a maximum mutation parameter, and also limited to ensure that the current situation maintains its current classification under the classifier rule.

3.5. CROSSOVER

We use a marginally-modified form of one-point crossover. Consider viewing the environment condition of a rule as consisting of several predicates, each possessing an initial point A, a final point B, a lower bound l, an upper bound u, a field ϕ and an operation ω. We could choose to view this as a list of the form

$$\left\{A_0, B_0, l_0, u_0, \phi_0, \omega_0, \ldots, A_{p-1}, B_{p-1}, l_{p-1}, u_{p-1}, \phi_{p-1}, \omega_{p-1}\right\} \tag{26}$$

where p is the number of predicates contained in the rule. Apply one-point crossover on two lists of this form, but insure that both lists break the predicates in the same way.

3.6. LEARNING PARAMETERS

There are numerous parameters used in XCS, a few added by XCSR, and a few more still added here. Choosing their values wisely can be very important in some problem domains unfortunately. This subsection gives brief descriptions of the important parameters and specifies sensible default values for typical problems. It is important that any results described should also list the parameter settings used.

3.6.1. From XCS.

These are the parameters that are present in XCS. As such, they are also present in XCSR and TSC.

3.6.1.1. General Parameters
These are parameters related to the general operation of XCS.

Maximum total numerosity. This is N in [28]. It specifies the maximum size of the population in micro-classifiers, that is, the maximum sum of the numerosities of the classifiers. This should be a positive integer, normally in the hundreds or at most the thousands.

Learning rate. This is β in [28]. It is used as the learning rate for the predicted payoff, prediction error estimate, GA fitness, and action set size estimate for the classifiers. This should be in the range $[0.1, 0.2]$ for most problems, and always in the range $[0, 1)$.

Possible actions. This is $\mathcal{A}$, the set of all of the possible actions that the classifier rules may take for values of a.

3.6.1.2. Recalculating Fitness   These are parameters used in XCS while recalculating the fitness of the rules in the population.

Multiplier parameter. This is α in [28]. This is the multiplier used in recalculating the fitness of the classifiers in the *update fitness* algorithm from §3.7. It is usually around 0.1.

Equal error threshold. This is ε_0 in [28]. This is the threshold used in recalculating the fitness of the classifiers in the *update fitness* algorithm from §3.7 to decide if the errors are essentially the same. It is usually around 1% of the ρ, the reward.

Power parameter. This is ν in [28]. This is the exponent used in recalculating the fitness of the classifiers in the *update fitness* algorithm from §3.7. It is typically set to 5.

3.6.1.3. Multi-Step Specific   These are parameters that are only used in multi-step problems.

Discount factor. This is γ in [28]. It is the discount factor used in multi-step problems when updating the classifier predictions. It is typically around 0.71.

3.6.1.4. GA Specific   These are parameters that are only used by the GA within XCS.

GA Threshold. This is θ_{GA} in [28]. The GA is run whenever the average number of generations since the last time the GA was run is greater than this threshold. It is typically in the range $[25, 50]$, and should always be in $\mathbb{N}^*$.

Crossover probability. This is χ in [28]. It is the probability of applying the crossover operator while executing the GA. It is typically in the range $[0.5, 1.0]$.

Mutation probability. This is μ in [28]. It is the probability of applying the mutation operator while executing the GA. It is typically in the range $[0.01, 0.05]$.

Deletion threshold. This is θ_{del} in [28]. It is the threshold for classifier deletion. If a classifier's experience is greater than this parameter then it may be considered for deletion. It is typically 20.

Fitness fraction threshold. This is δ in [28]. It is the fraction of the mean fitness of the population below which the fitness of a classifier may be considered in its probability of deletion. It is typically around 0.1.

Initial fitness. This is F_I in [28]. It is used as the initial value of the fitness used by the GA for the newly-created classifiers. It is typically only slightly more than zero.

3.6.1.5. Rule Set Specific These are parameters that deal with the rule set as a whole.

Minimum subsumption experience. This is θ_{sub} in [28]. The experience of a classifier must be greater than this threshold for it to subsume another classifier. It must hold that $\theta_{sub} \in \mathbb{N}^*$, and typically we have $\theta_{sub} \geq 20$.

Covering probability. This is $P_\#$ in [28]. It is the probability of using the covering element in a single attribute. It is typically around 0.33.

Initial prediction. This is p_I in [28]. It is used as the initial value of the predicted payoff for the newly-created classifiers. This is typically slightly more than zero.

Initial prediction error. This is ε_I in [28]. It is used as the initial value of the estimated prediction error for the newly-created classifiers. It is typically only slightly more than zero.

Exploration probability. This is P_{explr} in [28]. It specifies the probability of exploration during the action selection phase. It is typically around 0.5.

Minimal number of actions. This is θ_{mna} in [28]. This should be in $\mathbb{N}$, and is typically equal to the number of possible actions, so that complete covering will take place.

Maximum number of steps. This is the maximum number of steps that a multistep problem can spend in one trial. This variable is not mentioned in [28], but it is present in Butz's XCS code written in the C programming language.

GA subsumption? This is *doGASubsumption* in [28]. It is a boolean parameter specifying if the offspring are to be tested for possible logical subsumption by the parents. It is usually best to set this to *true*.

Action set subsumption? This is *doActionSetSubsumption* in [28]. It is a boolean parameter specifying if action sets are to be tested for subsuming classifiers. It is usually best to set this to *true*.

3.6.2. From XCSR.

These are the learning parameters that are added to an XCS system by XCSR. Since our system derives from XCSR, we use these as well. The variables used here are slightly different from those in a traditional XCSR.

Problem range. This is a two-element list of the lower and upper values that the input is expected to lie within. As the input violates this, this range is expanded automatically. As an example, if it is known for a specific problem that the input should always lie within the real-valued range $[0,1]$, then this should be set to the list $\{0.0, 1.0\}$.

Covering maximum. This is how large of a fraction of the range can be added to both the lower and upper bounds combined in the covering. The current default value we are using is 0.1. Thus, if we wish to cover $[0.3, 0.5]$, which has a spread of $0.5 - 0.3 = 0.2$, the largest allowable spread would be $(1 + covering_{maximum})spread = (1 + 0.1)0.2 = 0.22$.

Mutation maximum. This is how large of a fraction of the range may be added or subtracted from the lower and upper bounds in the mutation method. The current default

value we are using is 0.1. For example, if we are mutating a rule which matches the bounds $[0.3, 0.72]$, which has a spread of $0.72 - 0.3 = 0.42$, we would have a maximum change of 0.042, so our mutated rule would now match bounds determined randomly from $[0.3 \pm 0.042, 0.72 \pm 0.042]$, but enforced to be within the problem bounds.

Initial spread limit. This is s_0 in [5]. It is the maximum initial spread when a new predicate is created through the covering operator.

3.6.3. New in TSC.

These are the parameters introduced specifically for use in TSC.

Maximum environment condition length. This is how many predicates we may have at the maximum in any individual classifier. It should always be a positive integer.

Maximum temporal mutation. This is the most that the temporal element of the position may be randomly perturbed during the mutation process. It should always be a positive integer.

Maximum position mutation. This is the most any dimensional element of a position may be randomly perturbed during the mutation process. It should always be a positive integer.

Valid operations. This is a list of all the valid operations for the classifier, the ω's, a list of first-order lexical closures. A first-order lexical closure is, roughly speaking, a function and its associated scope. These ω's each must be capable of operating on any arbitrary list of data extracted from the data set, and these lists of data are extracted by following the raster paths through the data.

Valid fields. This is the list of valid fields for the classifier, the ϕ's, a list of first-order lexical closures. These ϕ's must be capable of operating on a single time instance of the data.

Visible time range. This is the range in time that is visible to the classifiers. None of the classifiers are allowed to look beyond this window. This also is generally how much of a history should be generated before the classifier system is allowed to start. This is a set interval.

3.7. TRIVIALLY MODIFIED ALGORITHMS

There are several algorithms from XCS and XCSR that are only slightly modified for our purposes from their original forms.

The *Generate Match Set* Algorithm. This is the *GENERATE MATCH SET* function in [28]. The match set M contains all of the classifiers in the population P which match the current situation. After filling the match set with all pre-existing matching classifiers, it repeatedly generates new covering classifiers until the minimum number of actions is satisfied.

The *Select Action* Algorithm. This is the same as in traditional XCS. There are two methods for selecting an action used here: either randomly, or the best action.

The *Generate Action Set* Algorithm. This is the *GENERATE ACTION SET* function in [28]. It forms the action set A out of the match set M, all of the classifiers that match the selected action.

The *Update Set* Algorithm. This is the *UPDATE SET* function in [28]. It updates the parameters for classifiers in the action set.

The *Update Fitness* Algorithm. This is the *UPDATE FITNESS* function in [28]. The fitness of all of the classifiers in the action set are updated in a normalized manner.

The *Run GA* Algorithm. This is the *RUN GA* function in [28]. It runs a simple genetic algorithm, not on the full population P, but instead only on the action set A, in order to induce niching.

The *Select Offspring* Algorithm. This is the *SELECT OFFSPRING* function in [28]. It uses a roulette-wheel method of selection.

The *Insert into the Population* Algorithm. This is the *INSERT IN POPULATION* algorithm in [28]. It is slightly more complex than just pushing the new classifier into the population list: we need to check to see if it is already present in the population. If it is, we must increment that classifier's numerosity instead. For a new classifier r, find

an $r' \in P$, with P being the entire population, such that r and r' are identical. If such an r' exists, increment r'_n; otherwise insert r into P.

The *Delete from Population* Algorithm. This is the same as the *DELETE FROM POPU-LATION* function in [28]. It decides which members of the population are suitable for deletion, allowing for niching, and then removes low-fitness individuals.

The *Deletion Vote* Algorithm. This is the same as the *DELETION VOTE* algorithm in [28]. The deletion vote for a classifier r is dependent upon its action set size estimate. Let $F_{average}$ be the average fitness in the entire population. We want classifiers with sufficient experience and a significantly lower than average fitness than the rest of the population to be deleted before others. Expressed in terms of the TSC parameters as outlined in §3.6:

$$r_{exp} > \theta_{del} \bigwedge \frac{r_F}{r_n} < \delta F_{average}. \tag{27}$$

This then returns

$$\frac{r_{as}r_n F_{average}}{\left(\frac{r_F}{r_n}\right)} = \frac{r_{as}r_n^2 F_{average}}{r_F} \tag{28}$$

as the deletion vote for this classifier r; otherwise it returns $r_{as}r_n$ as the deletion vote for this classifier r.

The *Do Action Set Subsumption* Algorithm. This is the *DO ACTION SET SUBSUMP-TION* function in [28]. The function chooses the subsumer from the most general classifiers capable of subsumption and then subsumes all possible classifiers in to the subsumer.

The *Could Subsume?* Predicate. We say that a specific classifier r is capable of subsuming others if it has both sufficient accuracy and sufficient experience. That is, if the experience of the classifier is greater than the minimal subsumption experience threshold, and the prediction error of the classifier is less than the equal error threshold. In symbols:

$$r_{exp} > \theta_{sub} \bigwedge r_\varepsilon < \varepsilon_0. \tag{29}$$

The *Subsume?* Predicate. This is called *DOES SUBSUME* in [28]. A classifier r^1 subsumes another classifier r^2 if the following conditions are all met:

1. Their actions are identical: $r_a^1 = r_a^2$.

2. The classifier r^1 is capable of subsumption, as decided by the *could subsume?* predicate described in §3.7.

3. The classifier r^1 is more general than the classifier r^2, as decided by the *more general?* predicate described in §3.10.

3.8. THE *MATCH?* PREDICATE

This is based upon the algorithm called *DOES MATCH* in [28], but it has been generalized in order to suit our needs here. Assume a classifier r and a situation σ. In traditional learning classifiers, $\sigma \in \{false, true\}$ which is usually represented $\{0, 1\}$, and therefore it is only necessary to see if every element in the condition part of the classifier r, that is r_c, is either equal to each other or a covering symbol in r:

$$\left(r_{c_i} = \sigma_i \bigvee r_{c_i} = \# \right) \forall i \in \mathbb{Z}_{|r_c|=|\sigma|}. \tag{30}$$

For us, it is slightly more involved due to the more complex nature of the conditions used in the construction of the classifiers.

The *match?* predicate for ternary values. For ternary values as used in traditional learning classifiers, a ternary predicate t matches a situation element x when either $t = x$ or $t = \#$, the covering symbol. Similarly, a ternary predicate t matches a second ternary predicate u when t matches all of the situations matched by u; that is, when $t = u \bigvee t = \#$.

The *match?* predicate for ranges. For ranges as used in Wilson's XCSR [5], a range predicate r matches a situation x when that situation x lies within the lower and upper bounds specified by the range predicate, $l \leq x \leq u$.

The *match?* predicate for a time-series. If we take the data along the straight line segment from the initial point A to the final point B, forming a vector d, we can then form d' by applying ϕ to each element in d:

$$d_i' = \phi(d_i) \forall d_i \in d. \tag{31}$$

The predicate is said to match the data if and only if

$$l \leq \omega \left(d' \right) \leq u. \tag{32}$$

When all of the predicates of the rule match, then the rule matches; the rule then recommends a particular classification or action.

Two situations σ_1 and σ_2 match if every one of their elements match element-wise:

$$\text{match?} \left(\sigma_{1_i}, \sigma_{2_i} \right) = true \ \forall i \in \mathbb{Z}_{|\sigma_1| = |\sigma_2|}. \tag{33}$$

The *match?* predicate for classifiers and situations. A classifier r matches a situation σ if r^1 and r^2 match, as decided by the *match?* predicate described in §3.8, and at least one of the elements of the classifier is more general in r^1 than in r^2.

The *match?* predicate for classifiers. A classifier r^1 matches another classifier r^2 if the environment condition of r^1 matches the environment condition of the classifier r^2.

3.9. THE *GENERATE COVERING CLASSIFIER* ALGORITHM

This is derived from the *GENERATE COVERING CLASSIFIER* function in [28]. It creates a classifier which matches the current situation. This is handled somewhat differently in TSC than in XCS or in XCSR, and the method operates as described in Algorithm 3.5.

Input: a TSC instance.
1. **let** l be randomly chosen, $1 \leq l \leq$ the maximum environment condition length.
2. **let** c, the condition $\leftarrow$ *nil* $= \{\}$, an empty list.
3. **let** a, the action $\leftarrow$ a random element from the set of all possible actions that are not in the match set.
4. **for** l times **do**
5.   **push** a covering predicate **onto** c
6. **return** a new classifier instance with environment condition c, action a, time stamp set to the current number of situations, and the rest of the slots set to their defaults.

Algorithm 3.5. Generating covering classifiers.

3.10. THE *MORE GENERAL?* PREDICATE

This is derived from the *IS MORE GENERAL* function in [28].

The *more general?* predicate for a TSC predicate. This returns true only if the predicate p matches predicate q and if it is more general than it as well. Predicate p is more general than predicate q if and only if:

$$p \text{ matches } q \wedge$$

$$\left(l_p < l_q \vee u_q < u_p \vee \left(path_q \text{ lies completely along } path_p \wedge path_p \neq path_q \right) \right).$$

The *more general?* predicate for classifiers. This is based upon the algorithm called *IS MORE GENERAL* in [28], but it has been generalized in order to suit our needs here. In traditional learning classifiers, it is only necessary to count the occurrences of the covering symbol, #, in order to determine which of two classifiers is more general: the one with the greater number of occurrences of it. For us it is slightly more involved due to the more complex nature of the conditions used in the construction of the classifiers. A classifier r^1 is more general than another classifier r^2 if r^1 and r^2 match, as decided by the *match?* predicate described in §3.8, and at least one of the elements of the classifier is more general in r^1 than in r^2.

4. EXPERIMENTAL RESULTS

4.1. THE NATURE OF A REALISTIC TIME SERIES

The primary difficulty experienced in testing was an unknown aspect of time series themselves. Originally the test problem was a very simple one-dimensional sine wave, with only a simple slope function for an operator, and with the classification task of deciding if the next point will be up or down from the current point. This appears as if it were a trivial problem, and indeed a high degree of accuracy can be achieved with only two very simple rules: if the previous point is below the current one then the next point will be above; otherwise the next point will be below the current point.

This approach will not work in general. There are several distinct types of time series, such as: up-trending, down-trending, steady, periodic, up-step, down-step, hills, and valleys. Real-world time series are comprised of several of the characteristics from each type, and any system that would be capable of operating on a real-world time series would need to be able to handle all of the different types simultaneously. The problem is that a simple slope operator is only capable of learning time series that are primarily linear, and a periodic time series such as the sine wave requires entirely different operators.

4.2. THE SIMPLISTIC INCREASING/DECREASING TESTS

The original test time series was a sine wave, which is a perfect example of a periodic function, but the simple slope operator is only capable of learning linear time series data. The new tests were designed with this in mind, and is actually a closer match to the appearance of real market data.

The first new test was simply a randomly chosen slope for a line, either upward or downward; the classification question is still whether or not the next point will be above or below the current one; this was very quickly optimally learned by the system.

In the second simple test, the series is randomly selected to be either upward or downward for a random number of time steps, with a randomly chosen slope, over and over again with completely different random elements each time. This was also very quickly
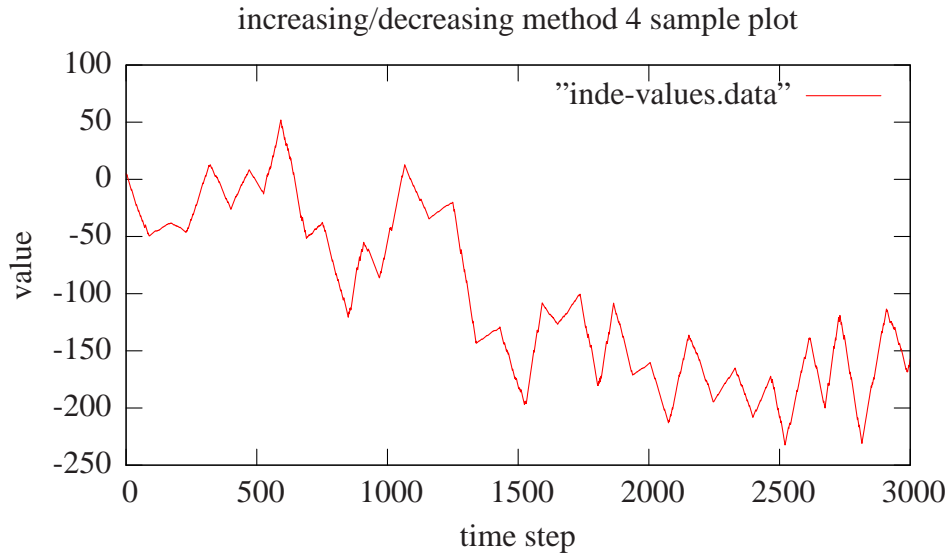
Figure 4.1. Increasing/decreasing method 4 sample plot.

optimally learned by the system.

The third simple test added random noise to the second test; TSC would typically optimally learn this problem within 1,000 to 2,000 time steps.

The fourth simple test randomly switched the direction of the time step with a certain probability. This, as well, was optimally learned within 1,000 to 2,000 time steps. This test would superficially resemble a traded entity, so it is of particular interest. What is shown in Figures 4.1 and 4.2 is a typical run under this test, with a probability of exploration of 0.35 and a probability of random misdirection of 0.1; this would imply a best-case eventual accuracy of:

$$1 - \frac{0.35}{2} - 0.1 = 0.725$$

or 72.5%, which eventually appears.

4.3. THE STOCK MARKET

We experimentally determined many of the parameters that are best for use on the stock market. We used actual historical data of the Dow Jones Industrial Average (^DJI), with daily trading data starting on August 20, 1990, with data ending on August 18, 2006. The data was gathered from Yahoo! Finance. The first 100 data points of the time series were

increasing/decreasing method 4 sample performance
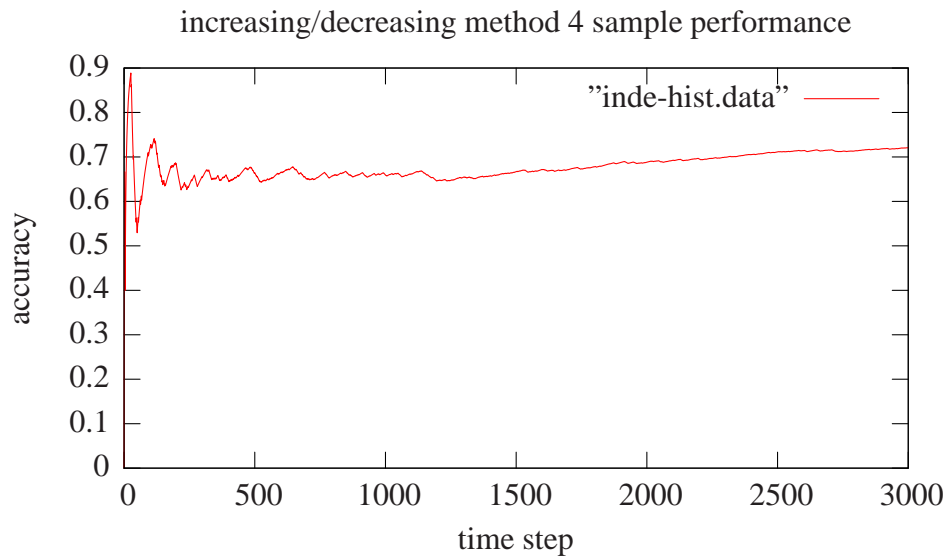
Figure 4.2. Increasing/decreasing method 4 sample performance.

skipped, allowing for historical data that far back even in the very first day of simulated data, causing an actual start of analysis of January 11, 1991. In each of these experiments, a statistical sample of at least 30 runs was gathered, each run going on for 1,500 simulated trading days (2,167 actual days, 5.93 years), for an end of December 16, 1996. At each trading day the stock was given the option to either put all of its resources into the ^DJI or into a bank account yielding roughly 4% per annum. The system initially had $1,000,000.00.

In these trials we report:

1. the trial number,

2. the number of correct actions,

3. the percentage of correct actions,

4. the final financial return,

5. the ratio of the final financial return to that of the buy-and-hold strategy,

6. and the percentage returned per annum.

parameter	value
max environment condition length	10
valid operations	simple slope
valid fields	closing price, opening price, and trading volume
max total numerosity, N	400
learning rate, β	0.2
discount factor, γ	0.71
GA threshold, θ_{GA}	25
equal error threshold, ε_0	20.0
multiplier parameter, α	0.1
crossover probability, χ	0.8
mutation probability, μ	0.04
exploration probability, P_{explr}	0.2
fitness fraction threshold, δ	0.1
covering probability, $P_{\#}$	0.33
initial prediction, p_I	10.0
initial prediction error, ε_I	0.0
initial fitness, F_I	0.01

Table 4.1. Initial parameters for the TSC.

We will use the buy-and-hold (B&H) strategy as our primary performance benchmark. In this strategy, the stock is purchased outright, and then the money is just left in the stock for the entire duration of the experiment.

Our initial parameters are listed in Table 4.1, and were chosen by general trial and error throughout the software development process.

4.3.1. Reward Methods. Several different possible reward methods for use in the stock market were considered, and we analyzed their relative performance. We refer to these different reward methods as a_1, a_2, b, c, d_{opt}, and d_{pess}.

Reward method a_1 is very simple:

1. **if** the correct action is taken **then**
2.   **return** a reward of 1,000.
3. **else**
4.   **return** a reward of 0,

It had the results as described in Table 4.2 over 36 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	754	50.256%	$1,853,080.30	0.67500	10.96%pa
std dev	22.3	1.489%	$333,964.25	0.12165	1.98%pa
max	797	53.133%	$2,527,462.80	0.92065	16.91%pa
min	697	46.467%	$1,117,451.00	0.40704	1.89%pa

Table 4.2. TSC results for reward method a_1.

Reward method a_2 is almost identical to a_1:

1. **if** the correct action is taken **then**

2.   **return** a reward of 1,000.

3. **else**

4.   **return** a reward of -200.

It had the results as described in Table 4.3 over 44 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	748	49.867%	$1,863,365.60	0.67875	11.06%pa
std dev	23.4	1.557%	$294,466.10	0.10726	1.75%pa
max	790	52.667%	$2,571,187.50	0.93657	17.25%pa
min	693	46.2%	$1,358,889.10	0.49499	5.31%pa

Table 4.3. TSC results for reward method a_2.

Reward method b offers slightly more incentive for good-performing rules:

1. **let** $\$_{ratio}$, the money ratio $\leftarrow \frac{\$_{t+1}}{\$_t}$, the ratio of the money the classifier has immediately one time-step in the future to the money it currently has.

2. **if** $\$_{ratio} > 1.005$ **then**

3.   **return** a reward of 1,000.

4. **else**

5.   **return** a reward of 0.

It had the results as described in Table 4.4 over 57 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	750	49.992%	$1,792,041.90	0.65276	10.33%pa
std dev	27.9%	1.8631344	$378,179.50	0.13775	2.18%pa
max	815	54.333%	$2,820,059.80	1.02723	19.09%pa
min	692	46.133%	$1,219,942.60	0.44437	3.41%pa

Table 4.4. TSC results for reward method b.

Reward method c tries to scale the reward:

1. **let** $\$_{ratio}$ be the money ratio as previously defined.

2. **let** $m \leftarrow 1000$, a multiplier.

3. **let** $e \leftarrow 2$, an exponent.

4. **let** $s \leftarrow 1.015$, a threshold term.

5. **return** $m \cdot (\$_{ratio} - s)^e$

It had the results as described in Table 4.5 over 30 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	747	49.791%	$1,795,971.30	0.65420	10.37%pa
std dev	20.2	1.345%	$300,842.88	0.10958	1.74%pa
max	790	52.667%	$2,407,121.50	0.87681	15.95%pa
min	702	46.8%	$1,340,345.30	0.48823	5.07%pa

Table 4.5. TSC results for reward method c.

Reward method d is slightly more complex than the rest:

Input: cu, the amount of reward if the classifier is correct on an up day.

Input: cd, the amount of reward if the classifier is correct on an down day.

Input: iu, the amount of reward if the classifier is incorrect on an up day.

Input: id, the amount of reward if the classifier is incorrect on an down day. *// Days that are not up are viewed as down days here.*

1. **if** the classifier has chosen the correct action $\wedge$ it is an up day **then**
2.   **return** cu.
3. **else if** the classifier has chosen the correct action $\wedge$ it is a down day **then**
4.   **return** cd.
5. **else if** the classifier has chosen the incorrect action $\wedge$ it is an up day **then**
6.   **return** iu.
7. **else if** the classifier has chosen the incorrect action $\wedge$ it is a down day **then**
8.   **return** id.

From this we have the two reward methods d_{opt}, which is optimistic, and d_{pess}, which is pessimistic.

Reward method d_{opt} calls d with the values of $cu = 1000, cd = 750, iu = 0, id = 200$. It had the results as described in Table 4.6 over 45 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	728	48.526%	$1,624,189.40	0.59162	8.52%pa
std dev	22.2	1.477%	$223,009.56	0.08123	1.17%pa
max	786	52.4%	$2,122,616.30	0.77318	13.52%pa
min	689	45.933%	$1,163,151.90	0.42369	2.58%pa

Table 4.6. TSC results for reward method d_{opt}.

Reward method d_{pess} calls d with the values of $cu = 750, cd = 1000, iu = 200, id = 0$.

It had the results as described in Table 4.7 over 45 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	728	48.526%	$1,624,189.40	0.59162	8.52%pa
std dev	22.2%	1.477	$223,009.56	0.08123	1.17%pa
max	786	52.4%	$2,122,616.30	0.77318	13.52%pa
min	689	45.933%	$1,163,151.90	0.42369	2.58%pa

Table 4.7. TSC results for reward method d_{pess}.

From these experiments we see that the a methods are the best performing, although there is no effective difference between the performance of a_1 and a_2: this is because the scaling of the reward should not effect the outcome of the reward method at all. We arbitrarily choose of the two to employ a_2 for the remaining experiments.

4.3.2. GA Thresholds. After deciding on a_2 as the best reward method and keeping it for the rest of these tests, we turn our attention to optimizing the GA threshold θ_{GA}, which is described earlier in §3.6.1.4. We chose to look at the possible values for this parameter of 25, 35, 45, and 50.

A GA threshold of 25 was used in the previous situation, so we can borrow the results from that a_2 run; refer to Table 4.3.

For a GA threshold value of 35, we observed the results as described in Table 4.8 over 30 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	759	50.571%	$1,874,746.50	0.68289	11.17%pa
std dev	22.7	1.513%	$315,092.60	0.11477	1.88%pa
max	806	53.733%	$2,627,517.80	0.95709	17.68%pa
min	719	47.933%	$1,346,346.10	0.49042	5.14%pa

Table 4.8. TSC results for a GA threshold of 35.

For a GA threshold value of 45, we observed the results as described in Table 4.9 over 31 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	760	50.688%	$1,881,119.10	0.68521	11.24%pa
std dev	25.6	1.706%	$217,843.06	0.07935	1.30%pa
max	816	54.4%	$2,297,796.00	0.83699	15.05%pa
min	699	46.6%	$1,250,916.30	0.45566	3.85%pa

Table 4.9. TSC results for a GA threshold of 45.

For a GA threshold value of 50, we observed the results as described in Table 4.10 over 30 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	763	50.891%	$1,885,079.90	0.68665	11.28%pa
std dev	21.1	1.405%	$293,885.56	0.10705	1.76%pa
max	808	53.867%	$2,425,741.30	0.88359	16.10%pa
min	713	47.533%	$1,329,746.00	0.48437	4.91%pa

Table 4.10. TSC results for a GA threshold of 50.

There was no significant effect on the results of the algorithm based on the GA threshold: all of the other means fall well within $\frac{1}{4}$ of a standard deviation relative to the initial value of $\theta_{GA} = 25$, so we will employ that value for all remaining experiments.

4.3.3. Crossover Probabilities.   After deciding on the correct reward method and the correct GA threshold, using those results, we investigated the crossover probability, which is described earlier in §3.6.1.4. We chose to look at 0.5, 0.6, 0.7, 0.8, and 0.9.

For a crossover probability of $\chi = 0.3$, we obtained the results as described in Table 4.11 over 33 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	755	50.358%	$1,862,015.40	0.67825	11.05%pa
std dev	24.3	1.621%	$213,367.14	0.07772	1.27%pa
max	829	55.267%	$2,354,066.50	0.85749	15.52%pa
min	712	47.467%	$1,313,611.90	0.47849	4.71%pa

Table 4.11. TSC results for $\chi = 0.3$.

For a crossover probability of $\chi = 0.5$, we obtained the results as described in Table 4.12 over 31 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	754	50.275%	$1,882,082.30	0.68556	11.25%pa
std dev	24.9	1.662%	$265,281.13	0.09663	1.59%pa
max	799	53.267%	$2,426,894.30	0.88401	16.11%pa
min	717	47.8	$1,354,985.40	0.49356	5.25%pa

Table 4.12. TSC results for $\chi = 0.5$.

For a crossover probability of $\chi = 0.7$, we obtained the results as described in Table 4.13 over 34 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	754	50.275%	$1,884,173.30	0.68632	11.27%pa
std dev	28.9	1.930%	$258,017.90	0.09399	1.54%pa
max	809	53.933%	$2,526,742.80	0.92039	16.90%pa
min	688	45.867%	$1,264,236.40	0.46051	4.03%pa

Table 4.13. TSC results for $\chi = 0.7$.

A crossover probability of 0.8 was used in the previous situation, so we can borrow the results from the $\theta_{GA} = 25$ run; refer to Table 4.3.

For a crossover probability of $\chi = 0.9$, we obtained the results as described in Table 4.14 over 39 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	759	50.626%	$1,943,606.10	0.70797	11.85%
std dev	21.9	1.462%	$277,516.22	0.10109	1.69%
max	801	53.400%	$2,399,683.00	0.87410	15.89%
min	707	47.133%	$1,419,889.40	0.51721	6.09%

Table 4.14. TSC results for $\chi = 0.9$.

We can now easily observe that a crossover probability of $\chi = 0.9$ offers the best results with an arithmetic mean of 11.85%pa, and we employ it for all of the remaining experiments.

4.3.4. Mutation Probabilities.   Using all of our previous results, we then looked into the mutation probability, described earlier in §3.6.1.4. We looked at values of 0.04, 0.06, 0.08, 0.10, 0.15, and 0.20.

A mutation probability $\mu = 0.04$ was used in the previous situation, so we can borrow the results from the $\chi = 0.9$ run; refer to Table 4.14.

For a mutation probability $\mu = 0.06$, we observed the results as described in Table 4.15 over 34 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	762	50.831%	$1,972,095.00	0.71835	12.13%pa
std dev	21.5	1.433%	$255,299.13	0.09299	1.57%pa
max	792	52.800%	$2,734,496.80	0.99606	18.47%pa
min	704	46.933%	$1,579,600.50	0.57538	8.01%pa

Table 4.15. TSC results for $\mu = 0.06$.

For a mutation probability $\mu = 0.08$, we observed the results as described in Table 4.16 over 39 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	754	50.285%	$1,905,925.10	0.69425	11.48%pa
std dev	29.79326	1.986%	$285,127.30	0.10386	1.72%pa
max	806	53.733%	$2,421,790.30	0.88216	16.07%pa
min	668	44.533%	$1,230,840.30	0.44834	3.56%pa

Table 4.16. TSC results for $\mu = 0.08$.

For a mutation probability $\mu = 0.10$, we observed the results as described in Table 4.17 over 36 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	761	50.744%	$1,950,889.10	0.71063	11.92%pa
std dev	22.6	1.506%	$299,845.56	0.10922	1.83%pa
max	796	53.067%	$2,891,320.00	1.05319	19.59%pa
min	709	47.267%	$1,250,508.00	0.45551	3.84%pa

Table 4.17. TSC results for $\mu = 0.10$.

For a mutation probability $\mu = 0.15$, we observed the results as described in Table 4.18 over 32 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	763	50.908%	$2,037,007.90	0.74200	12.74%pa
std dev	22.299%	1.487%	$320,506.80	0.11675	2.00%pa
max	804	53.600%	$2,975,396.80	1.08381	20.17%pa
min	719	47.933%	$1,406,036.50	0.51216	5.91%pa

Table 4.18. TSC results for $\mu = 0.15$.

For a mutation probability $\mu = 0.20$, we observed the results as described in Table 4.19 over 36 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	762	50.800%	$1,889,297.80	0.68819	11.42%pa
std dev	24.369835	1.625%	$232,916.92	0.08484	1.40%pa
max	803	53.533%	$2,708,086.00	0.98644	18.28%pa
min	697	46.467%	$1,502,196.10	0.54719	7.10%pa

Table 4.19. TSC results for $\mu = 0.20$.

We can now easily observe that a mutation probability of $\mu = 0.15$ offers the best results with a arithmetic mean of 12.74%pa, and we therefore use that value for all remaining experiments.

4.3.5. Exploration Probabilities.   After this we looked at the exploration probability, which we describe in §3.6.1.5. We investigated the possible values of 0.1, 0.2, 0.3, and 0.4, using our previous results for the rest of the parameters.

For an exploration probability of $P_{explr} = 0.1$, we observed the results as described in Table 4.20 over 42 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	762	50.795%	$1,849,187.80	0.67358	10.92%pa
std dev	28.9	1.925%	$233,230.42	0.08496	1.38%pa
max	810	54.000%	$2,446,262.50	0.89107	16.27%pa
min	691	46.067%	$1,210,933.40	0.44109	3.28%pa

Table 4.20. TSC results for $P_{explr} = 0.1$.

An exploration probability of $P_{explr} = 0.2$ was used in the previous situation, so we can borrow the results from the $\mu = 0.15$ run; refer to Table 4.18.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	763	50.908%	$2,037,007.90	0.74200	12.74%pa
std dev	22.3	1.487%	$320,506.80	0.11675	2.00%pa
max	804	53.600%	$2,975,396.80	1.08381	20.17%pa
min	719	47.933%	$1,406,036.50	0.51216	5.91%pa

Table 4.21. TSC results for $P_{explr} = 0.15$.

For an exploration probability of $P_{explr} = 0.3$, we observed the results as described in Table 4.22 over 40 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	765	50.985%	$2,090,409.40	0.76145	13.23%pa
std dev	22.8	1.521%	$295,592.78	0.10768	1.87%pa
max	817	54.467%	$2,848,646.30	1.03764	19.29%pa
min	725	48.333%	$1,536,175.30	0.55956	7.50%pa

Table 4.22. TSC results for $P_{explr} = 0.3$.

For an exploration probability of $P_{explr} = 0.4$, we observed the results as described in Table 4.23 over 47 trials.

—	correct	% correct	returns	B&H ratio	%pa
B&H	806	53.733%	$2,745,309.50	1.0	18.54%pa
arith mean	767	51.119%	$1,950,627.60	0.71053	11.92%pa
std dev	19.5	1.299%	$196,644.44	0.07163	1.20%pa
max	806	53.733%	$2,395,224.50	0.87248	15.86%pa
min	730	48.667%	$1,637,668.40	0.59653	8.67%pa

Table 4.23. TSC results for $P_{explr} = 0.4$.

We can now easily observe that an exploration probability of $P_{explr} = 0.3$ offers the best results with an arithmetic mean of 13.23%pa, and we therefore use that value.

5. CONCLUSIONS AND FINAL RESULTS

After all of our tests we arrived at the set of parameters in Table 5.1 for the time series classifier. In this table the return is the equivalent percentage per-year (%pa) return provided by the parameters at that setting, and the B&H ratio is the performance relative to a simplistic buy-and-hold stategy, with 1.0 being equal, less than 1.0 implying an underperforming result over the same period, and greater than 1.0 implying a superior result over the same time period. The DJIA returned 18.54%pa over the period investigated here, and we failed to meet that in any of our tests. For example, 11.06%pa implies that with all of the other parameters set to their initial default and the reward method set to a_2 is equivalent to a savings account yielding 11.06%pa returns, but underperforming the DJIA itself if we were to merely buy and hold it for the same period of time. While these results demonstrate the system's ability to learn a complex situation, they are not at a level acceptable for real-world use on the stock market, underperforming the simplistic buy-and-hold strategy. Instead this system in its current form will only truly be applicable to less interesting problem spaces.

TSC would not be a usable real-world system for the stock market unless it were to result in returns in excess of the buy-and-hold strategy, which it did not. If it were capable of outperforming buy-and-hold then we could use it for automated and unsupervised trading. As it is, a more effective real-world approach would be to simply purchase an indexing fund. TSC is no longer useful to us since our interest is specifically automated stock trading,

parameter	value	return	B&H ratio
reward method	a_2	11.06%pa	0.67875
GA threshold, θ_{GA}	25	$\cdots$	$\cdots$
crossover probability, χ	0.9	11.85%pa	0.70797
mutation probability, μ	0.15	12.74%pa	0.74200
exploration probability, P_{explr}	0.3	13.23%pa	0.76145

Table 5.1. TSC Final Parameters

and our research will continue towards other avenues of automated time series analysis and prediction, probably still in the area of evolutionary computation and possibly employing a novel type of LCS.

There are many real-world applications comprising simpler time series than the stock market, and TSC does have a lot of room left to grow still, so continued research by others would be welcomed and potentially fruitful. TSC demonstrates that an LCS can natively represent a time series under analysis and learn in such an environment: that demonstration is the most valuable result of this research, perhaps encouraging more attempts at LCS-based time series analysis methods.

6. FUTURE WORK

There are several opportunities for improvement on TSC. Some of these are obvious and result from known simplifications and limitations of the current TSC system. The most obvious paths for future research with this TSC fall into the following major tasks:

1. using more advanced ϕ's,

2. using more advanced ω's,

3. finishing the implementation of multidimensionality,

4. using more advanced concepts in the GA,

5. represent the rule strengths with polynomials instead of reals,

6. changing from a Michigan to a Pittsburg approach,

7. using a GP instead of a GA,

8. and applying the system to other real-world problems.

Using more advanced ϕ's, is the most straightforward to start on. In the version of TSC as outlined here, and in the associated code, it is entirely possible to use any lexical closure as a ϕ, as long as it is capable of operating on one position of the time series data. In our use we only used ϕ to select the data field, but there is no reason why this should not or could not have vastly more complex operations. Any operations that would be useful in discernment might be useful.

Using more advanced ω's would address what is probably the greatest weakness of the current system. At present we have only used a simple slope function for the ω and have not attempted anything else. There are bound to be many more useful functions available. We specifically expect that the ability to match against polynomials and against periodic functions would be of the most intrinsic value.

Extending TSC so that it is a system fully capable of handling multivariate time series depends on the previous two tasks' completion first. The TSC system as described and the code used were both originally designed to handle multivariate time series, and therefore much of the work is already completed, but exactly what else remains to be finished is not entirely clear. We assert that at least new ω's that are designed with multivariate time series in mind would be required, but there may be other elements of the TSC system that need revision as well.

Using more advanced concepts within TSC's GA would be one of the easiest methods of improvement. The form of crossover we used was simple one-point crossover, and there are several well-known forms of crossover with better performance in general use. Employing a self-adaptive GA to evolve its own parameters encoded in its gene could also provide for some major gains, as this has been the most computationally intensive part of our investigation. Other methods of mutation may be beneficial, although this would require novel work: the non-standard form of the individuals in TSC appears to necessitate non-standard mutation approaches. The easiest modification of the mutation that would possibly be beneficial would be to try a Gaussian form of mutation which would allow for more drastic alteration to the population members on rare occasions. This would allow the system to adapt more fully to notably different environments.

The measures of the strengths here are real numbers currently, but we suspect that they may be better represented by polynomials, especially in the stock market problem since there is a great deal of difference in the value of a rule in differing times for any specific stock.

XCS and company use the so-called Michagan approach, where the entire population is the rule set. We suspect that the Pittsburgh approach, where each individual in the population is a complete rule set, could possibly be a better fit for our stock market problem in specific and possibly time series problems in general. This would be quite involved, and almost a complete redesign of the system.

Replacing the GA with a genetic program (GP), would be quite an undertaking. This would allow for vastly more complex classification rules and could possibly discover new basic metrics for the time series problems presented to the system. This would be of particular value with the stock market even though there are several well-known metrics because they are rarely of any quality. This would even more valuable for less-investigated time

series problems since there might not even be any known metrics as of yet for the problem.

The final task is actually many tasks: TSC should be applied to many more real-world problems, both to better solve those problems and to improve TSC itself. We hope that this work will prove useful in many problems and look forward to its use by others.

BIBLIOGRAPHY

[1] John H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern directed inference systems*, pages 313–329. Academic Press, New York, NY, 1978.

[2] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[3] Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1 – 18, 1994.

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

[5] Stewart W. Wilson. Get real! XCS with continuous-valued inputs. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems: From Foundations to Applications*, volume 1813 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 209 – 219. Springer-Verlag, 2000.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. The MIT Press, Cambridge, Massachusetts, 1998.

[7] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[8] C. Stone and L. Bull. For real! xcs with continuous-valued inputs. *Evolutionary Computation*, 2003.

[9] George Edward Pelham Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[10] George Edward Pelham Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.

[11] Fredic M. Ham and Ivica Kostanic. *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill Higher Education, 2000.

[12] Shaun-Inn Wu and Ruey-Pyng Lu. Combining artificial neural networks and statistics for stock-market forecasting. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 257–264, New York, NY, USA, 1993. ACM Press.

[13] Thomas Kolarik and Gottfried Rudorfer. Time series forecasting using neural networks. In *APL '94: Proceedings of the international conference on APL : the language and its applications*, pages 86–94, New York, NY, USA, 1994. ACM Press.

[14] Andrew Skabar and Ian Cloete. Neural networks, financial trading, and the efficient markets hypothesis. In *ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science*, pages 241–249, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[15] Yung-Keun Kwon, Sung-Soon Choi, and Byung-Ro Moon. Stock prediction based on financial correlation. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 2061–2066, New York, NY, USA, 2005. ACM Press.

[16] Robert Andrews and Shlomo Geva. Rule extraction from local cluster neural nets. *Neurocomputing*, 2000.

[17] David de la Fuente, Alejandro Garrido, Jaime Laviada, and Alberto Gómez. Genetic algorithms to optimise the time to make stock market investment. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1857–1858, New York, NY, USA, 2006. ACM Press.

[18] Peter Belford. Candlestick stock analysis with genetic algorithms. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1851–1852, New York, NY, USA, 2006. ACM Press.

[19] M. A. Kaboudan. Genetic programming prediction of stock prices. *Comput. Econ.*, 16(3):207–236, 2000.

[20] Hakman A. Wan, Andrew Hunter, and Peter Dunne. Autonomous agent models of stock markets. *Artif. Intell. Rev.*, 17(2):87–128, 2002.

[21] Sonia Schulenburg and Peter Ross. Strength and money: An lcs approach to increasing returns. In Lanzi et al. [29], pages 114 – 137.

[22] Sonia Schulenburg and Peter Ross. A learning evolutionary trading system, May 29 2002.

[23] Stewart W. Wilson. Function approximation with a classifier system. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 974–981, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[24] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function, April 25 2004.

[25] Theodore J. Perkins and Doina Precup. A convergent form of approximate policy iteration. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *NIPS*, pages 1595–1602. MIT Press, 2002.

[26] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. XCS with computed prediction in multistep environments. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1859–1866, Washington DC, USA, 25-29 June 2005. ACM Press.

[27] Arie Kaufman and Eyal Shimony. 3d scan-conversion algorithms for voxel-based graphics. In *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics*, pages 45–75, New York, NY, USA, 1987. ACM Press.

[28] Martin V. Butz and Stewart W. Wilson. An algorithmic description of XCS. In Lanzi et al. [29], pages 253 – 272.

[29] Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Advances in Learning Classifier Systems*, volume 1996 of *Lecture Notes in Artificial Intelligence (LNAI)*. Springer-Verlag, 2001.

VITA

Christopher Mark Gore was born in Cleveland, Ohio, on December 8, 1978. He received his High School Diploma from Triad High School in Saint Jacob, Illinois in 1997. Then he received his Associate of Science from Southwestern Illinois College, located in Belleville, Illinois, in 2001. After this he received his Bachelor of Science in Mathematics and Computer Science from Eastern Illinois University, located in Charleston, Illinois, in 2003. This thesis is part of the requirements for the completion of his Master of Science in Computer Science from the University of Missouri – Rolla, located in Rolla, Missouri, in 2007. His computational interests include evolutionary algorithms and other methods of unaided computational learning, financial simulation and analysis, and Lisp. He is currently employed at Astronautics Corporation of America developing software for the Integrated Network Server Unit (INSU) that will fly with the Airbus A400M, a large turboprop-driven military cargo aircraft designed to replace the aging C-130 Hercules throughout the world. He married Monica Louise Gore (née Smith) on May 27, 2006, and they currently reside in Oak Creek, Wisconsin, a suburb of Milwaukee, with their cat Casper.