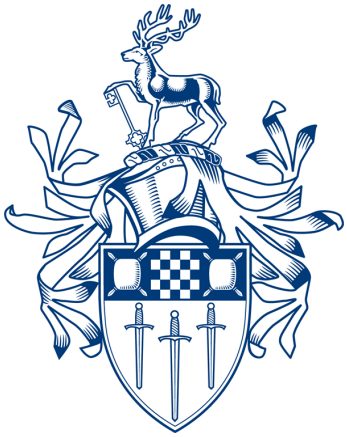
University of Surrey

Department of Computer Science

Optimal Run Parameter Values in Learning Classifier Systems for Single-Step Classification Problems

Author:
Vasily Shcherbinin

Supervisor:
Dr. Sotiris Moschoyiannis



A thesis submitted for the degree of

BSc Computer Science

May 21, 2019

Abstract

This research project aims to narrow down the scope of possible values for major Learning Classifier Systems run parameters and provide concrete guidelines to determine what the parameters should be given different problem scenarios, explaining the relationship between the run parameters and the data being learned. This will be achieved by implementing and running two Michigan-style Learning Classifier System variant implementations - XCS and UCS - on single-step benchmark problems and determining the effect of various run parameter values on the model prediction accuracy, computational time, and the resulting model size.

Declaration of Originality

I confirm that the submitted work is my own work. No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have also clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook.

I understand that the University may submit my work as a means of checking this, to the plagiarism detection service Turnitin® UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

If in completing this work I have been assisted with its presentation by another person, I will state their name and contact details of the assistant in the 'Comments' text box below. In addition, if requested, I agree to submit the draft material that was completed solely by me prior to its presentational improvement.

Acknowledgements

I would like to thank my supervisor, Dr. Sotiris Moschoyiannis, for their expertise, support and guidance throughout this project.

I would also like to thank my family: my parents, Gennady and Tatiana, for giving me an opportunity to receive higher education in the United Kingdom and for their support during my studies; and my sisters, Maria and Ekaterina, for their endless support and love.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This section will provide an introduction to the project, as well as provide an overview of what this project aims to accomplish and how this will be done. The section will begin with a description of a hypothetical problem scenario, and will use it to introduce the motivation of the project. The introduction will then provide a brief overview of the algorithms and their corresponding run parameters to be used in this project. The chapter will finish with clear project objectives outlined and the structure of the report presented.

1.1 Problem Scenario

Assume there is a task to create an application that would be able to provide smart suggestions to travellers regarding the fastest way to travel from point A to point B. The application would take in as input information regarding the weather and future weather broadcasts (will it rain or will there be no rain?), information regarding the time of day (rush hour?), day of week (work day?), season, traffic congestion, etc, and then would provide a user with a recommendation based on the input from the environment. Such scenario is possible to implement using Learning Classifier Systems, a rule-based machine learning paradigm that evolves a population of rules that is able to extract patterns and relationships from large amounts of data. The population of rules then serves as a model that can be used to determine the best action to take given the state of the environment, or predict which action would be best to take if the state of the environment was different to what has been seen before.

Learning Classifier Systems (LCS) have a wide range of applications, but due to their complexity require prior tuning before being applied to solve a particular problem. One way to execute such tuning is via run parameters - parameter constants set at the beginning of the algorithm run that can have a significant impact on the ability of the LCS algorithm to successfully create a good model for the problem being solved.

1.2 Project Motivation

Interestingly, despite the known importance of run parameters on the system operation, no extensive studies have been conducted previously to determine the optimal parameter values for a given problem. Of course, run parameter values would be greatly dependent on the problem being solved and the data that is used: its amount and size, presence of noise, data complexity and presence of patterns (generalisation) in the studied domain, but very scarce information can be found in regards to concrete values that should be used in various problem scenarios. Majority of scientific papers do not explain their choice of run parameters, simply stating what they are or providing very high level suggestions that can not be pinpointed to particular values (e.g. "must be set high"). Since there is no one-fits-all solution, "sweet-spot" suggestions without value justification are insufficient. **Taking all of this into account, this research project aims to narrow down the scope of possible values for the major LCS run parameters and provide concrete values for what the parameters should be given the different problem scenarios, explaining the relationship between the run parameters and the data being learned**.

This will be achieved by running two Michigan-style LCS variant implementations - XCS and UCS - on single-step benchmark problems and determining the effect of various run parameter values on the model prediction accuracy, computational time, and the resulting model size.

1.3 Project Objectives

The objectives of this project are to:

- Provide a comprehensive overview of how Michigan-style Learning Classifier Systems operate.
- Provide overview of the importance of setting the correct LCS run parameters in accordance with the problem being solved.
- Provide a software implementation of XCS and UCS algorithms in Python that can be used to investigate run parameter impact, replicate results and continue research in this direction.
- Provide comprehensive analysis of acquired results through experimentation to determine the optimal critical run parameter values for single-step problems of various complexity and nature. **The key goal is to determine the optimal run parameter values to maximise the resulting model prediction accuracy and minimise the algorithm run time and model size**.

1.4 Structure of Report

The report has been divided into seven chapters:

- *Chapter 2: Statement of Ethics* - will provide a Statement of Ethics regarding the project, demonstrating considerations for Legal, Social, Ethical and Professional aspects of the project.

- *Chapter 3: LCS Overview* - will provide an in-depth overview and literature review of what Learning Classifier Systems are, their historical evolution, overview of the LCS Functional Cycle and how it can be adapted to suit different problems, overview of the XCS and UCS LCS variants, an example of an LCS run and, finally, some interesting and important applications of LCS in various industries.

- *Chapter 4: LCS Run Parameters Overview* - will lay out the foundations for this project, providing an overview and literature review of the existing run parameters and their importance in the LCS functional cycle. Furthermore, the chapter will provide a list of run parameters to be investigated in this project, as well as a review of the existing literature and background research on the topic.

- *Chapter 5: LCS Implementation Design and Overview* - will provide an in-depth look into the software implementation used for this project. The chapter will begin with a literature review of existing LCS implementations. Implementation decisions used in this project will be justified, and an overview of the Benchmark problems used will be provided. Justification of methods utilised for result validation will also be covered. The chapter will then demonstrate a single iteration of the LCS algorithm, provide implementation code snippets for different LCS components, as well as demonstrate the final result.

- *Chapter 6: Analysis of Acquired Results* - will provide an in-depth analysis for all parameters that have been investigated within the project, demonstrating data gathered and drawing conclusions. Justification and explanation of results acquired will also be provided.

- *Chapter 7: Conclusion and Future Work* - will provide a final conclusion to the project, as well as lay out the direction for future work. The chapter will conclude with an in-depth Evaluation of the project, as well as an evaluation of the created implementation.

Chapter 2

Statement of Ethics

This section will provide insight into the relevant Legal, Social, Ethical and Professional (LSEP) aspects relating to the project. Even though the project does not rely on any externally gathered data, LSEP aspects must be considered to ensure that the project is ethical and legal.

2.1 Legal Considerations

Legal considerations included consideration for legislation that must be followed and respected during the execution of the project and presentation of project results.

Since this research project does not require participation of any external subjects or use of any externally gathered sensitive data, legislation regarding informed consent and data confidentiality is not applicable nor relevant to this project. This project involved utilising and modifying existing code written by a third-party, unaffiliated to the project, individual, so special care was given to respect the legislation and licenses by which the used code was protected. Specifically, the code used (details in Chapter 5) is an open-source implementation of an educational learning classifier system, licensed under GNU General Public License v3.0[1]. This license gives the following permissions:

- Commercial use (software and derivative may be used for commercial purposes)
- Modification (software may be modified)
- Distribution (software may be distributed)
- Patent use (license provides an express grant of patent rights from contributors)
- Private use (software may be used or modified in private)

[1] `https://www.gnu.org/licenses/gpl-3.0.en.html`

This project respects and does not break any of the permissions of the license above when utilising the third-party code. Where third-party code was used, the original author was referenced and credit to the original author was given. The resulting software that was used for this project has been made open-source under the GNU General Public License in order to encourage further research into the results and to encourage other researchers to replicate the experiments described in this report.

Special care was given in regards to the tables and figures used in this project report in order not to infringe on intellectual property and copyright of the original creators. Majority of figures and tables adapted from other researchers work were distributed under the Creative Commons Attribution License, which allows material unrestricted use, distribution, and reproduction in any medium, given the original work is cited. In situations where no evidence of a copyright license was present and there was no way to contact the original creator, figures and tables were assumed to fall under the Fair Use policy, which allows material reuse as long as the source is properly indicated. In this project report, use of copyrighted work was not abused and only used where absolutely necessary to convey the message; all images and tables that have been adapted from external sources have been properly cited and referenced. Furthermore, every instance of use of external ideas from other researchers, direct quotes and paraphrases of other researchers' ideas have also been correctly referenced.

2.2 Social Considerations

Social considerations included understanding the impact that the project can have on society and whether it adheres to the commonly-accepted societal norms.

The project aims to bring a positive impact to society, by providing results that can then be utilised to improve the use of learning classifier systems and their application to various domains in the industry. The resulting software accompanying this project, and used within the project to gather data, has been made open-source for other people to use and experiment with; at the same time, social impact was considered by making sure that the learning classifier system implementation can not possibly be used to do any harm. This was ensured by having the LCS solution solely narrowly applicable to the research task at hand, making it fit specifically for this project research purpose and unfit for any other use, including malicious.

2.3 Ethical Considerations

Ethical considerations included understanding and considering the morals/principles of what constitutes right or wrong in relation to the carried out project, and considering whether or not the proposed project and proposed project idea are within the boundaries of accepted conduct in society and profession.

The project adheres to all ethical principles, from the original research idea to the resulting implementation and acquired results. Making sure the project is ethical and corresponds to common expectations of what an ethical project should be was an important project consideration. Ethical standards were followed to make sure that the project idea does not cause harm to others, that the project was carried out ethically and respectfully to other researchers on whose research the project based its ideas - all ideas, code and graphics that were not created originally by the author was correctly referenced and cited, respecting the time and work others. In accordance to the ethical principle of "only assessing relevant components", only the relevant parts of others' research was utilised within the project report, solely to convey the message as correctly and comprehensibly as possible. Similarly, the aforementioned implementation of the LCS system was devised with a narrowly meant purpose, making it unfit to be used maliciously. Finally, ethics dictate that conclusions drawn and reported must be thoroughly checked and based on factual information - every statement made in this report was based on either results achieved directly in this project or on results that have been achieved by others. Best effort was made to carry out the experiments that yielded the results in the most professional, ethical and fair manner, attempting to bring bias to a minimum.

2.4 Professional Considerations

Professional considerations encompass considerations of what behaviour is expected from individuals that consider themselves professionals in a given field or area. As a Computer Science student at the University of Surrey, the author of this work is a member of the British Computer Society (BCS) and must adhere to the BCS Member Code of Conduct[2] that regulates the behaviour of BCS members in professional matters. The author therefore used the BCS Code of Conduct as guidance to the professional behaviour that is expected during the execution of this project. The BCS Code of Conduct contains four points that must be considered and can be summarised as:

1. **Public Interest**
    - *Have high regard for society and environment well-being, respect and promote equality and human rights, respect rights of Third Parties.*

[2] `https://www.bcs.org/upload/pdf/conduct.pdf`

2. **Professional Competence and Integrity**

    - *Do not claim to be someone you are not, always develop and learn, understand the Legislation when carrying out professional responsibilities, respect others and their property, do not indulge in unethical activity.*

3. **Duty to Relevant Authority**

    - *Respect and follow the requirements of the Relevant Authority, avoid conflict of interest and take responsibility for committed work.*

4. **Duty to the Profession**

    - *Uphold the reputation of the profession, seek to improve professional standards, act with integrity and respect in professional relationships, encourage and support others in their development.*

The project described in this report follows and implements all four of the points made above. The project aims to provide knowledge in public interest, providing information that can be used by other researchers to enhance their own and future projects. The rights of Third Parties, as mentioned numerously before, have been noted, respected and documented appropriately. These considerations cover point **(1)** of the BCS Code of Conduct. Within this project, every claim made has been supported either by sources or directly by experimental results - to the best of ability, an attempt was made to demonstrate professional competence in discussed area of work and to act ethically at all times. These considerations cover point **(2)**. This project's supervisor acted as the Relevant Authority for this project, and therefore the project adhered to the requirements set out beforehand during the project planning stage. All responsibility for the committed work lies on me as the project author and as student of University of Surrey - this covers point **(3)**. Finally, during this project duty to the profession has been demonstrated clearly at all times - work was committed with integrity and respect to other researchers, individuals, and future readers. This showed primarily with every information source being clearly documented and cited, giving both credit to original author of the idea and allowing future readers to trace the idea source for further investigation, if such is required - point **(4)**.

Generally, this entire project had the aim of encouraging and supporting others - encouraging in investigating Learning Classifier Systems and their run parameters further, and supporting by providing information that will facilitate the process of setting the LCS run parameters for different types of problems.

Chapter 3

LCS Overview

> *"Learning Classifier Systems are a quagmire*[1] *- a glorious, wondrous and inventing quagmire, but a quagmire nonetheless."*
>
> - David Goldberg[2], *in a remark to a colleague.*

This chapter will introduce and elaborate on the concept of Learning Classifier Systems, and provide an in-depth overview of how this type of algorithms work. The chapter will provide an introduction and overview of the two most common LCS variations - XCS (eXtended Classifier System) and UCS (sUpervised Classifier System), and will proceed with an overview of other popular and prominent LCS variations. The chapter will also provide insight into Single vs. Multi-step Learning and will conclusion with an in-depth overview of LCS applications in Industry.

3.1 What is an LCS?

Learning Classifier Systems, abbreviated to LCS, are a group of algorithms within the rule-based machine learning paradigm that utilise a combination of a discovery component (commonly a genetic algorithm) and a learning component to identify a set of context-dependent rules that can collectively capture knowledge and patterns within a given environment. This set of rules can then be used to make predictions on novel input data based on the patterns learned, or can be used as a knowledge discovery mechanism, identifying interesting features and patterns within the context of the problem being solved.

Learning Classifier Systems implement a combination of reinforcement learning and evolutionary computing, as well as other methods, to produce systems that are adaptive to the environment and can act in accordance with the environment state. Evolutionary computing techniques are algorithms that are based on the biological principles of natural selection and genetics - Darwin principle of survival of the fittest is implemented via the use of biological processes, e.g. mutation and crossover. The key idea in evolutionary computing is to evolve an initially random set of rule population into a better, fitter rule population, that would provide more accurate - fitter

[1] a swamp, marshland; a complex or awkward situation.
[2] [Goldberg et al., 1992]

- solutions to the problem with every generation, essentially adapting to the problem being solved. Reinforcement learning techniques are based around the notion of learning via trial and error, receiving a numerical reward for correct actions. The goal of reinforcement learning is to maximise the future numerical reward.

LCS fall within the rule-based system category, utilising rules in the form of "IF state THEN action" to capture the environment state and propose the correct action. The LCS discovery component implements evolutionary computing and is responsible for retrieving and finding new rules within the environment, whilst LCS learning component implements reinforcement learning in order to assign rewards to existing rules and guide the algorithm to evolving and using better rules.

3.2 Evolution of LCS

Throughout the years since their introduction in the 1970's, within literature LCS have been referred to by different researchers as adaptive agents [Fukuyama and Holland, 1996], cognitive systems [Holland and Reitman, 1977], genetics-based machine learning systems [Dam et al., 2008] [Goldberg, 1989], production systems [Holland, 1992] - the modern term "Learning Classifier System" was only adopted in late 1980's, arguably being first coined by Goldberg in 1985 [Goldberg, 1985].

LCS were first introduced and described by John H. Holland, an American scientist, Professor of philosophy, electrical engineering and computer science at the University of Michigan in 1975 [Holland, 1975]. That work was revolutionary at the time and was the book that introduced and initiated the study of genetic algorithms and their applications. Among them, first outlines of the LCS algorithm emerged.

In 1977, Holland and Reitman presented the first concrete implementation of a learning classifier system [Holland and Reitman, 1977] - being the first to combine credit assignment with a genetic algorithm to evolve a population of rules as a solution to a problem present within an environment. Within that work, which was based around Holland's earlier and more prominent invention - the Genetic Algorithm - Holland laid out the basic components of an LCS algorithm. These components are still relevant to modern LCS algorithms:

> *"The type of cognitive system (CS) studied here has four basic parts: (1) a set of interacting elementary productions, called classifiers, (2) a performance algorithm that directs the action of the system in the environment, (3) a simple learning algorithm that keeps a record of each classifier's success in bringing about rewards, and (4) a more complex learning algorithm, called the genetic algorithm , that modifies the set of classifiers so that variants of good classifiers persist and new, potentially better ones are created in a provably efficient manner."*
>
> *- [Holland and Reitman, 1977]*

This very first LCS later became known as "Cognitive System One" (CS-1) and had little resemblance to the LCS algorithms of today - its immediate disadvantages were that its implementation was very complex and very difficult to understand

[Goldberg, 1989], limiting its adoption in the scientific community and practical utilisation in the real world [Bull, 2015].

CS-1 and other LCS implementations inspired from it share the same concept at their core - on a high level, these algorithms evolve a population of classifiers, each classifier representing a single rule from the environment. This approach results in the genetic algorithm operating on an individual rule basis, and the entire population of rules is considered to be the learned solution (or model) for the environment or problem. Such LCS have become known as *Michigan-style* LCS. In 1980, Stephen Smith in his PhD thesis [Smith, 1980] from the University of Pittsburgh proposed a fundamentally different approach to LCS - his implementation, LS-1, evolved a population of variable length rule-sets, where each rule-set was a potential solution to the environment problem (See Figure 3.1). The genetic algorithm operated on rule-sets rather than on individual rules, and the best solution to the environment problem was chosen via the best single rule set. Such LCS became known as *Pittsburgh-style* LCS. Through the years, Michigan-style LCS have gained more attention than Pittsburgh-style LCS, notably due to the broader range of problems and larger, complex tasks that Michigan-style LCS can be applied to, as well as due to Pittsburgh-style LCS greatest limitation - heavy computational requirements [Urbanowicz and Moore, 2009]. In recent years, a combination of the two LCS styles is being experimented with - such systems are denoted as "Hybrid", but they remain a topic of active research.

Stagnation within the LCS scientific field ensued, until in 1994 Stewart Wilson caused a breakthrough via the introduction of a new LCS - Zeroth Level Classifier System (ZCS) [Wilson, 1994]. ZCS was much simpler than previous LCS implementations (notably not incorporating several complex functionalities of the original LCS approach). This was the first LCS of a next generation of learning classifier systems.

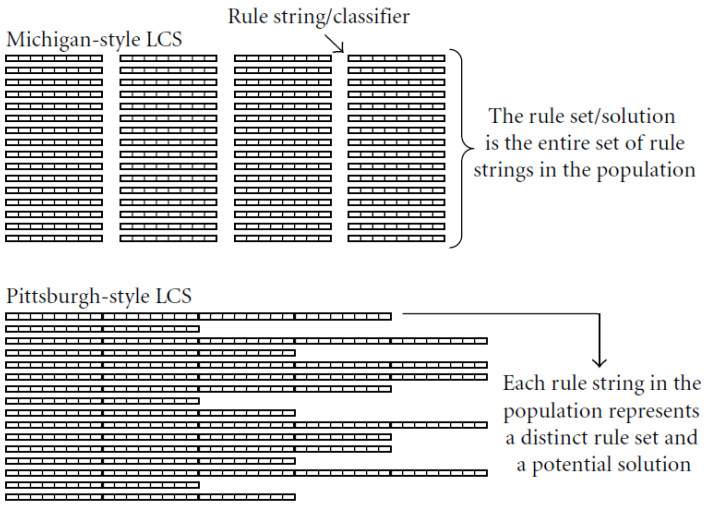


Figure 3.1: Michigan-style vs Pittsburgh-style LCS [Urbanowicz and Moore, 2009]

The following year Wilson proposed a new LCS - the eXtended Classifier System (XCS) [Wilson, 1995]. XCS was much more efficient than all other LCS that were before it, and initiated a new wave of interest towards Learning Classifier Systems and their applications in the real world. Most of the modern actively researched and commonly used LCS algorithms stemmed directly from XCS, what has been possible due to a number of excellent publications explaining XCS in a clear and comprehensible way, e.g. [Butz and Wilson, 2001].

In the past decade, the scientific field of LCS experienced a metaphorical rebirth, with more and more researchers conducting wealth of research on a wide range of LCS-related areas and topics. New LCS are being created and are achieving outstanding results - for example, ExSTraCS 2.0 [Urbanowicz and Moore, 2015], supervised LCS based on the UCS learning classifier system, was able to successfully solve the 135-bit Multiplexer problem - a remarkable achievement, considering that no other algorithm was able to solve that complex benchmark [Urbanowicz, 2015]. Publication of *Introduction to Learning Classifier Systems* [Urbanowicz and Browne, 2017] has further popularised LCS and increased their availability to researchers in other disciplines, students, and generally curious readers. All of LCS advancements and achievements in the last decade have produced an abundance of literature - clarifying understanding, providing algorithmic descriptions and comprehensible approaches to LCS implementations in code. As a result of this, the field of LCS is no longer "a quagmire" and has an exciting, interesting future ahead.

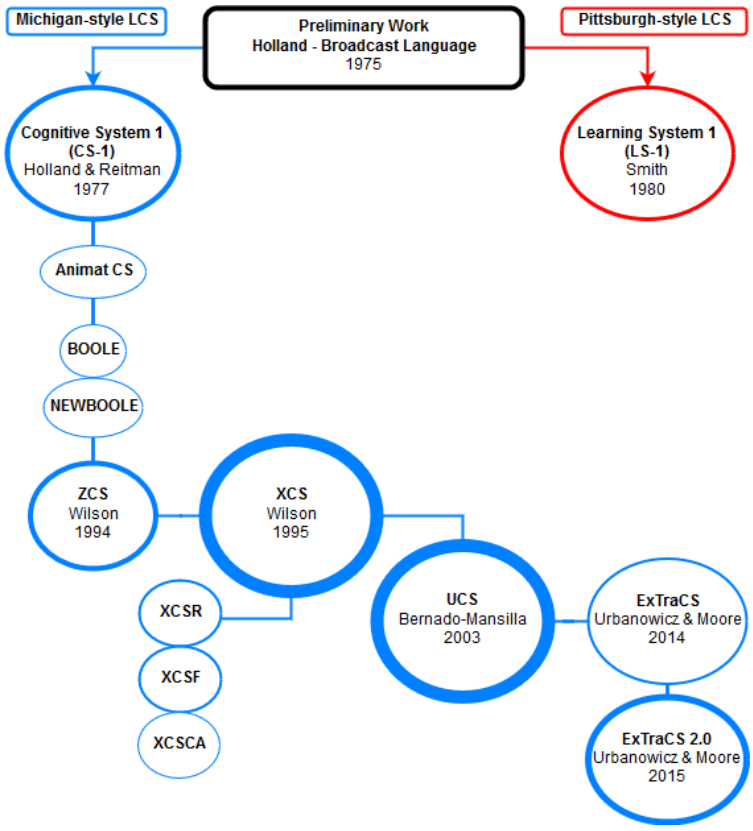


Figure 3.2: Snippet of LCS family tree. Circle thickness corresponds to LCS impact on the field.

3.3 LCS Functional Cycle Overview

This section will begin by providing a description of how a generic Learning Classifier System works and stepping through the LCS functional cycle. The description will be based on a Michigan-style LCS designed for supervised learning (similar to UCS). The section will then elaborate on how LCS components can be adapted and altered to fit different needs and solve various kinds of problems.

3.3.1 LCS Functional Cycle

The LCS functional cycle (Figure 3.3) can be separated into three components: the external environment, the learning machine, and, optionally, a rule compaction algorithm initiated during the post-processing stage. The main component of an LCS is the learning machine. The learning machine also includes several co-dependent sub-components. Together, all these components operate in a step-wise learning cycle in order to evolve a set of rules (model) that can capture knowledge within the environment, as well as be used for prediction.

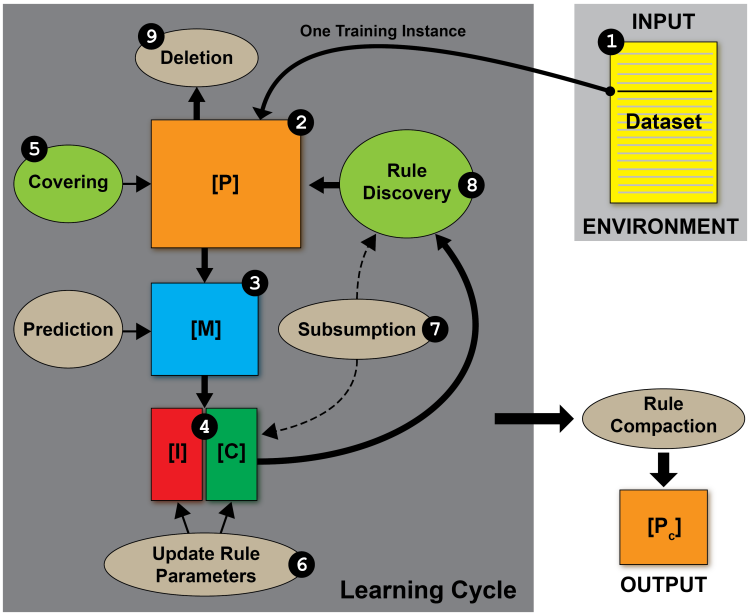


Figure 3.3: Functional cycle of a Generic LCS. [Urbanowicz and Browne, 2017]

The environment is the source of data that is supplied to the LCS algorithm to learn. The environment essentially could be abstracted to a dataset containing a certain number of training instances comprised of condition-action pairs. Figure 3.4 demonstrates an example of a training instance. Each instance condition has 11 attributes that can have a value of 0 or 1. The term 'class', 'action' or 'phenotype' refers to the condition label that needs to be predicted, and can have a value of either 0 or 1. In Figure 3.4, the action value is 1. LCS relies on incremental learning, meaning that it learns one training instance at a time - the environment supplies the learning machine with a training instance, which then completes a learning cycle and returns the training instance back to the environment. The process is then repeated, until some specified stop criteria is met or until the specified maximum number of learning iterations is exceeded.

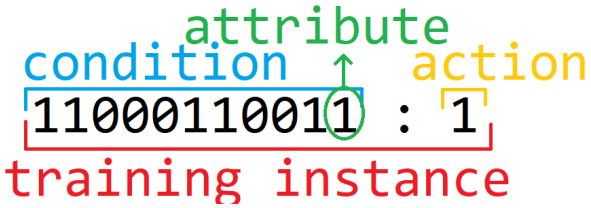


Figure 3.4: An LCS training instance example.

The learning cycle consists of several distinct steps and components. One of the most important components in an LCS is the rule population [P] (Table 3.1), which essentially is a collection of rules that together form the prediction model (this is a characteristic of Michigan-style LCS - alternatively, in Pittsburgh-style LCS, every rule is by itself a model). A single rule follows an "IF-THEN" expression and can be separated into two parts - the condition and the action. In different literature, the condition may also be known as *state*, whilst the action may be known as *class* or *phenotype*. The condition of a rule specifies values for some attributes in the dataset - in ternary alphabet, an attribute can have a value of 0 or 1. An attribute can also have a third value, denoted by a # and known as the 'wild card' - a # can match both 0 or 1. Wild cards allow to generalise rules and make them applicable to more than one instance - this in turn allows to capture useful data relationships without overfitting the rules to training data.

The action part of a rule represents the predicted class. Each rule has important parameters: numerosity is the number of copies of a given classifier present in the population; match count is the number of times the rule appeared in a match set; correct count is the number of times the rule was also present in the correct set. The rule's accuracy is calculated by dividing the correct count by the match count. Another rule parameter, rule fitness, is a function of accuracy calculated to reflect the quality of a classifier. At this stage, it is important to highlight the distinction between classifiers and rules: a classifier is the combination of a rule with its corresponding rule parameters.

A1	A2	A3	A4	A5	A6	Class	Fit.	Acc.	Num.	CorrectC.	MatchC.
#	#	#	**0**	**0**	#	**0**	0.4012	0.8330	7	1921	2306
#	#	**1**	**1**	**1**	#	**1**	0.4632	0.8574	4	1154	1346
#	#	#	#	#	**0**	**0**	0.0990	0.6297	1	3261	5179
0	#	#	**0**	#	#	**0**	0.1859	0.7142	4	1917	2684
#	#	#	**1**	#	#	**1**	0.0641	0.5772	1	2875	4981
0	#	**0**	#	**0**	#	**0**	0.2368	0.7497	3	1147	1530
#	#	**0**	#	#	#	**0**	0.1315	0.6665	3	3441	5163

Table 3.1: Snippet of an evolved rule population for a 6-bit Multiplexer Problem.

The first step of the LCS learning cycle (recall Figure 3.3) is to import a training instance from the environment **(1)**. The next step is to identify and choose all rules in the population [P] whose condition attributes match those of the training instance **(2)**. Matching (Figure 3.5) compares the current training instance to all rules in the population, and any wild cards in the rule match by default. Importantly, the class value is ignored in matching. If all specified attributes match, the rule is moved to the match set [M] **(3)**. The match set is then divided into two sets - a correct set [C] and an incorrect set [I] **(4)**. Rules that have a class prediction that matches the class of the current training instance form the correct set; the rest form the incorrect set. In the case that the correct set is empty, i.e. no matched rules have the correct class prediction, the covering mechanism is applied **(5)** - covering allows to introduce new rules to the population; this is known as 'rule discovery'. Covering guarantees that LCS explores relevant rules, preventing the LCS from wasting time and resources exploring rules that wouldn't match anything in the data. Since LCS rule populations start off empty, covering is also used for population initialisation - during the process, some of the attributes in the rule's condition and the class are generated using values taken from the current training instance; the rest are set to wild card value (Figure 3.6).

A1	A2	A3	A4	A5	A6	Class	MatchSet	CorrectSet
#	#	#	0	0	#	0		N/A
#	#	1	1	1	#	1		N/A
#	#	#	#	#	0	0		
0	#	#	0	#	#	0		N/A
#	#	#	1	#	#	1		
0	#	0	#	0	#	0		N/A
#	#	0	#	#	#	0		

Figure 3.5: Some examples of matching for 110100:0 training instance.

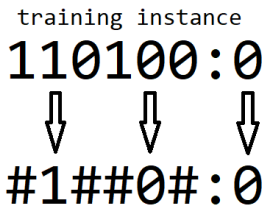


Figure 3.6: Example of Covering.

If there is at least one rule in the correct set, the next step is to update the classifier parameters for the classifiers that have made it into either the correct or incorrect sets **(6)**. Classifiers with the correct class prediction in the correct set will have their match and correct counts increased, in turn leading to increased accuracy and fitness. Alternatively, classifiers in the incorrect set will have their match count increased, since they matched the training instance, but not the correct count, in turn resulting in accuracy and fitness decrease. These fluctuations in classifier parameters allow to distinguish between good and bad rules. For different LCS algorithms, the fitness and accuracy measures utilise different computation formulas; the simplest fitness measure is simply finding the ratio of the correct classifications to the total number of times a rule appeared in the match set. This is known as long-term accuracy of

the i-th classifier [Urbanowicz and Browne, 2017]:

$$Fitness_i = \frac{CorrectCount_i}{MatchCount_i} \tag{3.1}$$

Alternatively, the "error" parameter can used to identify how often a classifier was unable to correctly model the data:

$$Error_i = 1 - \frac{CorrectCount_i}{MatchCount_i} \tag{3.2}$$

In LCS, more general rules (i.e. containing more wild card attributes) are more likely to appear in match sets and so in turn are more likely to be chosen as parents by the genetic algorithm in the next step. This encourages the LCS to evolve more general rules. Additionally, a mechanism called subsumption (Figure 3.7) is often applied **(7)** - during subsumption, pairs of rules are examined and situations are identified where the rule can act as a subsumer. A subsumer rule must match all the attributes of the other rule, must be more general and just as accurate. The subsumer essentially absorbs the more specific rule, with the subsumers numerosity being increased and the specific rule eliminated from the population. Subsumption can be applied to the rules at different stages of the learning cycle, depending on the LCS variation, for example to rules in the correct set or at the GA step closer to the learning cycle end.

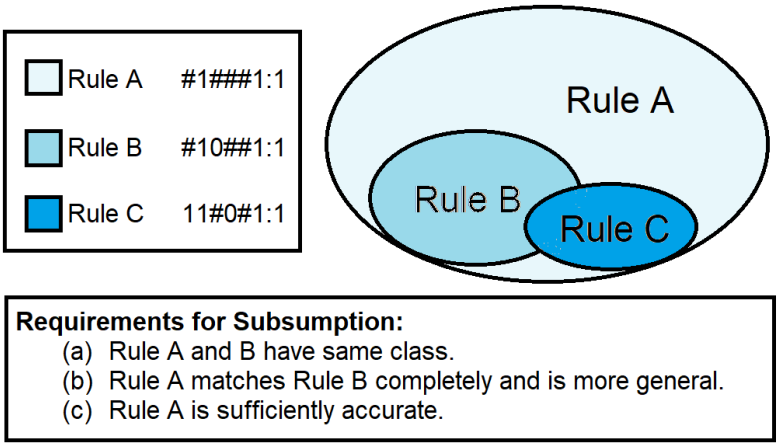


Figure 3.7: An LCS training instance example.

Next, a genetic algorithm (GA) is applied **(8)**. This is the second rule discovery mechanism in LCS, in addition to Covering. The LCS genetic algorithm preserves the majority of classifiers in the population during the learning iteration run, generating only two new rules that are added to the population. Three mechanisms are used within the GA: selection, crossover, and mutation. The role of selection is to pick two parent rules from the correct set as the basis for newly evolved rules. Several selection mechanisms exist, each with their own advantages - the preference lately lays with the tournament selection approach, where a portion of classifiers is randomly selected from the correct set and the classifier with the highest fitness is chosen to be the parent. New rules emerge from the parent rule by applying mutation and crossover, the two most common genetic operators. These operators

are designed to generate new rules that still match the current training instance, as well as retain the correct class prediction. Crossover is applied first in accordance to a user-defined probability. There are various crossover strategies, but uniform crossover is one that is most often encountered, as it allows for good generation of diversity. Both crossover and mutation modify offspring rules by either changing a specified attribute into a wild card, or turning a wild card into a specified attribute that has the value seen in the current training instance. Crossover looks for attributes that are specified in one parent rule but not the other, and for each decides whether to swap a specified attribute with a wild card. Following crossover, each new rule has the potential to be modified further by the mutation operator. Each attribute in the rule has some random chance of swapping from specified to generalised or vice versa, thus adding new variation to the offspring rules. Before adding the offspring rules to the rule population, their rule parameters are initialised to make them classifiers, and GA subsumption can be applied. After that, any classifiers that have been involved in this learning cycle are returned to the rule population.

The last step of the learning cycle is to make sure the size of the rule population is within the specified limits. This is done via the deletion mechanism **(9)**. The population size, which is the sum of all rule numerosities, is compared to the specified maximum population size parameter, and if the population size exceeds the specified limit, rules are deleted in turn from the population until the size limit is met. Classifiers for deletion are selected with a probability inversely proportional to their fitness. Now, the resulting rule population can be used as a prediction model, or as a knowledge representation of the data.

Sometimes, another post-processing step is applied to the rule population after training. This is known as 'rule compaction' - rule compaction aims to remove poor, redundant, or inexperienced rules from the population by removing rules that are inaccurate, have a low fitness or whose error threshold does not exceed the specified limit. Advantages of rule compaction include interpretability, compacted population size, and improved predictive performance.

Assume there is a need to use the resulting classifier set as a predictive model. For this, the rule population will be applied as a model to some testing data that was not seen before during the training stage. All learning components of the system are deactivated, and an instance is passed from the testing data to the LCS, where all classifiers in the population that match it form a match set. Differently to training, the match set is now passed to a prediction array which is responsible for making a class prediction based on the votes of these matching classifiers. The vote of each classifier is typically proportional to its fitness and numerosity. One of the simplest voting schemes is to sum the votes of all rules with the class and choose the class with the highest total vote. Consequentially, the testing accuracy of LCS can be defined as the ratio of correct predictions that were made by the prediction array over the entire testing data set.

3.3.2 LCS Pressures

LCS operation can be understood by looking at the LCS algorithm through a prism of "pressures" - trade-offs that must be balanced in order to ensure successful LCS operation [Urbanowicz and Browne, 2017]. Setting up LCS correctly to fit the problem is crucial - failure to do so will render the LCS ineffective and will not yield an acceptable result. There are a total of four pressures that must be balanced [Butz et al., 2005]:

1. Set Pressure
2. Mutation/Crossover Pressure
3. Subsumption Pressure
4. Fitness Pressure

Set Pressure

Set pressure encourages classifier generality, basing on the idea that the more general the classifier is, the higher the probability that it will be chosen as a parent for new rules (due to a higher chance to Match the training instance). Set pressure will encourage classifier generality until classifiers become completely general, what will render them essentially useless (###### : 0). Therefore, a balance must be found in order to evolve classifiers that are not overly general to the extent that they become useless, but at the same time sufficiently general to correctly generalise existing rules and patterns within the data set.

Mutation/Crossover Pressure

Mutation/Crossover Pressure addresses specifically the specificity (number of defined, specific, attributes) of classifiers. A balance must be found between evolving overly specific and overly general classifiers during mutation and crossover stages.

Subsumption Pressure

Subsumption Pressure addresses the need to subsume overly specific versions of a classifier. This ensures that sufficient level of generality is present within the evolved population of rules.

Fitness Pressure

Fitness Pressure seeks to evolve the population towards classifiers with the highest fitness. The problem that can result from this is overly specific classifiers and can result in overfitting of the LCS algorithm.

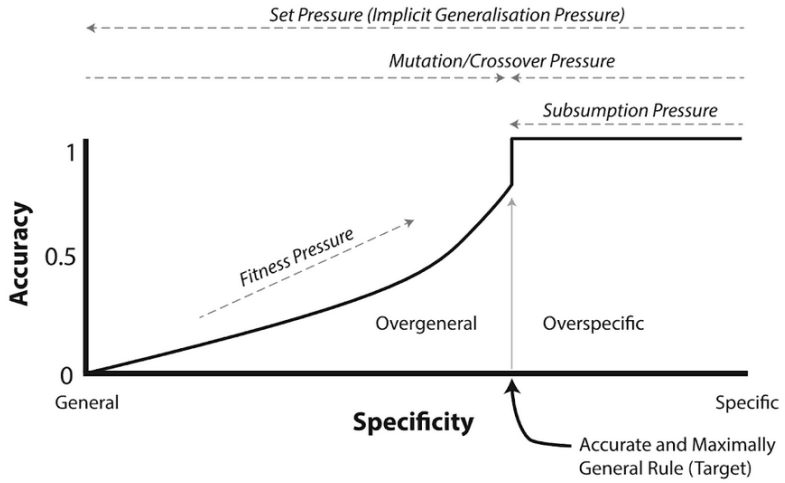


Figure 3.8: Different LCS Pressures collaborating with each other to achieve optimal solution [Urbanowicz and Browne, 2017].

3.3.3 LCS Adaptability

Apart from the run parameters that can be used to tune the Learning Classifier System and make it more fitting to certain types of problems and scenarios, numerous other methods and techniques exist. This section will describe these methods, as well as justify their choice in this particular project.

Alphabets for Rule Representation

Early LCS all employed the ternary alphabet {0,1,#} as their alphabet of choice, as it was, and still is, the simplest form of rule encoding and was sufficient for binary problem domains. Complexity arose when a need emerged to apply LCS to problem domains that can only be described by integers. Early suggestions included converting the environmental features (supplied as an integer) into binary for use with the ternary alphabet, but this was found to be inefficient and limiting the system ability to represent a problem and its solution [Schuurmans and Schaeffer, 1989]. Gray coding and the Hamming distance were suggested as a possible alternative binary alphabet encoding [Kovacs and Kerber, 2001] - this brought both advantages and disadvantages. The main disadvantage was that Gray encoding for integer domains lacked the ability to generalise the solutions, resulting in an increased number of rules required to describe the problem solution. This hindered the interpretability of the final solution, what defied one of LCS main advantages and use cases - to evolve an understandable and compact rule set describing the problem.

An alternative to binary encoding for integer domains is to adopt an integer alphabet. The simplest example of an integer alphabet is the quaternary alphabet, consisting of four possible values, e.g. {0,1,2,#}. Such alphabets found application in genetic data mining analysis, where all genotypes (i.e. data features) could be represented with values in the range of [0, 2]. A new LCS adapted from XCS - XCSI - was developed by Wilson in 2001 specifically for Integer-valued Data Mining - it was introduced and successfully utilised to extract data from the Wisconsin Breast Cancer (WBC) dataset in [Wilson, 2001].

Throughout the years, Learning Classifier Systems have been adapted for use with numerous different alphabet encodings, e.g. real-valued alphabet was implemented in XCS variant XCSR, used for Real-Valued Multiplexor Problems [Wilson, 2000]. Some other alphabets and rule representations can be seen in Figure 3.9:

Alphabet/ Rule Representation	Example Instance	Example Matching Rule	Example Non-matching Rule
Ternary (state values 0 or 1)	101000 : 0	##10#0 : 0	0####0 : 0
	001110 : 1	0#1### : 0	010### : 1
Integer (e.g. state values 0 - 5)	0,5,2,1,3,3 : 0	#,5,2,#,#,3 : 0	#,#,3,#,#,# : 0
	5,5,0,1,1,1 : 1	5,#,#,#,#,1 : 1	3,1,0,#,1,# : 1
Real Lower-Upper Bound (e.g. state values 0.0 - 1.0)	0.1,0.7,0.5,0.9 : 0	u #,0.7,0.6,# : 1 l #,0.5,0.4,# : 1	u 0.1,#,1.0,1.0 : 0 l 0.0,#,0.6,0.8 : 0
	0.4,0.8,0.2,0.2 : 1	u 0.6,#,0.3,# : 1 l 0.3,#,0.2,# : 1	u #,#,0.9,# : 1 l #,#,0.6,# : 1
Real Center-Spread (e.g. state values 0.0 - 1.0)	0.1,0.7,0.5,0.9 : 0	c #,0.6,0.5,# : 0 s #,0.2,0.1,# : 0	c 0.5,#,0.9,0.3 : 0 s 0.1,#,0.2,0.4 : 0
	0.4,0.8,0.2,0.2 : 1	c 0.4,#,0.3,# : 1 s 0.2,#,0.5,# : 1	c #,#,0.5,# : 1 s #,#,0.1,# : 1

Figure 3.9: Different LCS alphabets used for rule representations and associated matching and non-matching examples [Urbanowicz and Browne, 2017].

Lately, mixed alphabet representations have been proposed (Figure 3.10) - such have already been successfully implemented in more recent LCS, such as BioHEL and ExSTraCS. Mixed alphabet representation, also officially known as mixed discrete-continuous attribute-list knowledge representation, has the principle advantage that it utilises attribute-list knowledge representation (ALKR). ALKR ignores unspecified attributes (marked by #) and only stores specified values - this greatly enhances the matching time in large problem domains containing many features, as the LCS only has to determine whether the specified features match, rather than the entire condition.

Specifically for this project, the ternary alphabet was utilised, as it was the simplest and sufficient to explore the effect of run parameter on LCS effectiveness.

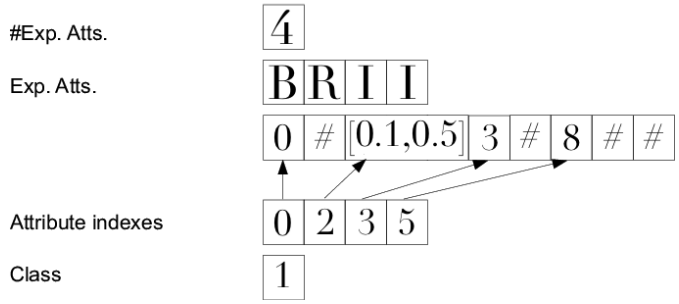


Figure 3.10: Example of mixed attribute specification: 4 specified attributes out of 8 are expected, with the attributes expected to be Binary (B), Real-valued (R), Integer from 1-5 (I) and Integer from 5-9 (I). By ignoring the # wild cards and creating an attribute list via attribute index referencing, matching efficiency can be significantly enhanced.

Modes of Operation: Explore vs Exploit

After LCS has added the matching classifiers to the Match set, it needs to determine the correct action to take. This might not be as simple as it initially seems, considering that classifiers in the Match set might not necessarily recommend the same action. Depending on the LCS variant, different methods exist to choose the correct action, but they are all based around finding the correct balance between the Explore (choose random action from available) and Exploit (only choose best action) modes of operation (this is also applicable and relevant to other machine learning algorithms). The LCS mode of operation can be separated into three cases [Urbanowicz and Browne, 2017]:

1. **Deterministic Greedy Selection:** in this mode of operation, the best action is always selected. How good the action is, or action fitness, is determined whether it was recommended by the most fit classifier, or whether it was the action with the highest number of votes from the classifiers in the Match set. This is an example of a pure exploit mode of operation. The caveat of only using the exploit mode of operation is that the system may become stuck in a local optimum without ability to discover the optimal solution. This does not occur in supervised systems like UCS, but could become a problem in reinforcement learning systems like XCS.

2. **Random Selection:** in this mode of operation, the action is selected randomly from all the available classifier actions present in the Match set. The advantage of this mode of operation is that the system will have a chance to test all available actions and thus will never get stuck in a local optimum, but this comes with a disadvantage: such mode of operation is inefficient and may take longer periods of time to converge to a global optimum.

3. **Probabilistic Greedy Selection:** in this mode of operation, a balance is found between exploit and explore modes. In order to achieve this, the action is chosen with a probability proportional to the classifier fitness. This introduces a stochastic possibility that the system will choose a worse action (less fit) than the best action in the Match set, but this in turn will allow to avoid

the problems present in Deterministic Greedy Selection and prevent+ getting stuck in the local optimum.

Unfortunately, it is challenging to find the optimum balance between Explore and Exploit at any given instance during training. If the Exploit mode is activated prematurely, the system may become trapped in the local optima; on the contrary, utilising Exploration already after good solutions have been discovered is inefficient. This is illustrated in Figure 3.11. Modern reinforcement LCS attempt to tackle this challenge by employing several of the above modes of operation at once - for example, utilise Exploitation in the early iterations of the algorithm cycle, and utilise Exploration in the later stages. Like this, sacrificing a bit of efficiency, the local optima may be avoided.

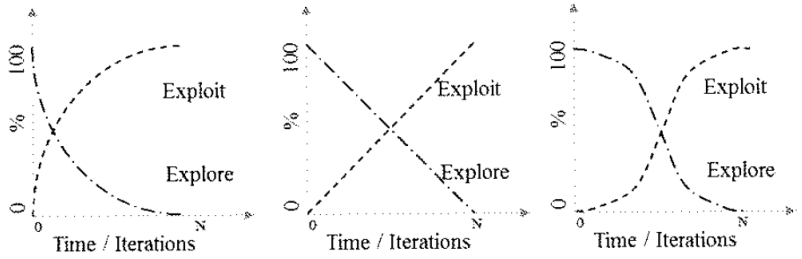


Figure 3.11: Various possible relationships between Exploit and Explore modes of operation in finding the optimal solution with the increase of iterations. What relationship is applicable to the problem is difficult to determine a priori [Urbanowicz and Browne, 2017].

It is worth highlighting again that the majority of said above is primarily relevant to reinforcement learning LCS variations, such as XCS. Supervised learning classifier systems, like UCS, do not need to choose the action, as it is already available from the environment - therefore such LCS primarily use the Exploit mode of operation when making predictions on testing data. Therefore, in this project UCS utilised the Exploit mode of operation when making predictions on testing data, whilst XCS utilised a 50/50 balance approach between Exploit and Explore.

Parent Selection Methods

Closer to the end of the LCS run, a Genetic Algorithm may be employed to generate new rules. For a new rule to be generated, two parent rules must be chosen. For this task, several parent selection methods exist. They are especially useful when solving complex problems, where simple selection of the fittest classifiers as parents would simply not suffice (due to risk of getting stuck in local optimum). Several parent selection techniques exist, most notable being Roulette Wheel Selection and Tournament Selection. Figure 3.12. and Figure 3.13. demonstrate the two approaches. Tournament selection has become lately the preferred parent selection approach, primarily due to its ability to identify best classifiers closer to the end of the LCS training run - on the contrary, Roulette wheel selection struggles with that task, failing to separate and distinguish between close classifiers, resulting in weaker classifiers being chosen whilst better ones were available. Specifically for this reason, the implementation in this project utilised Tournament selection.

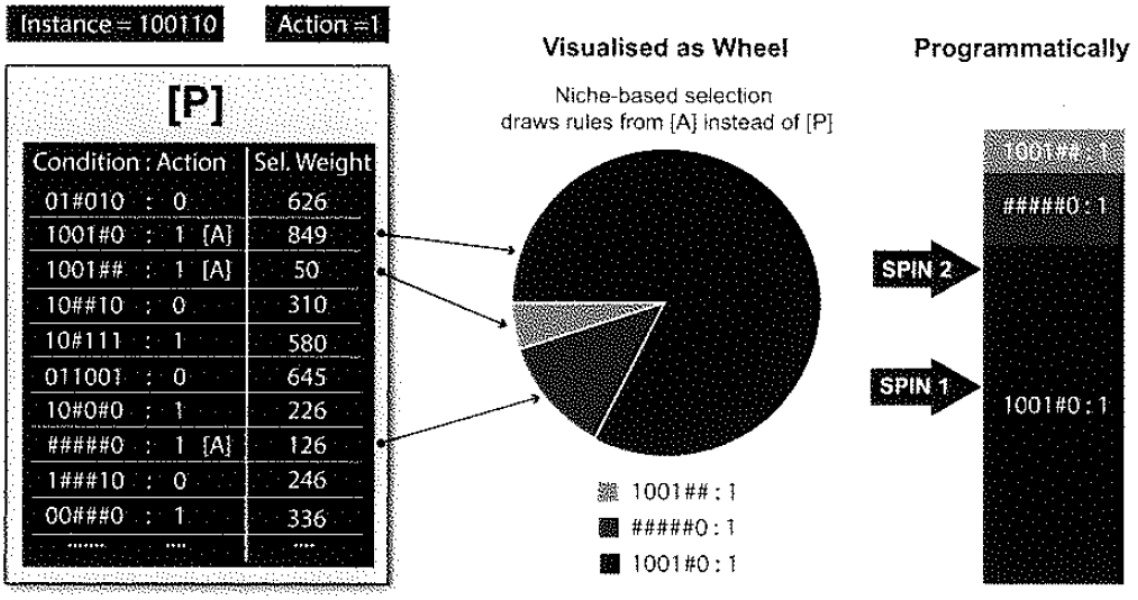


Figure 3.12: Roulette Wheel Selection [Urbanowicz and Browne, 2017].

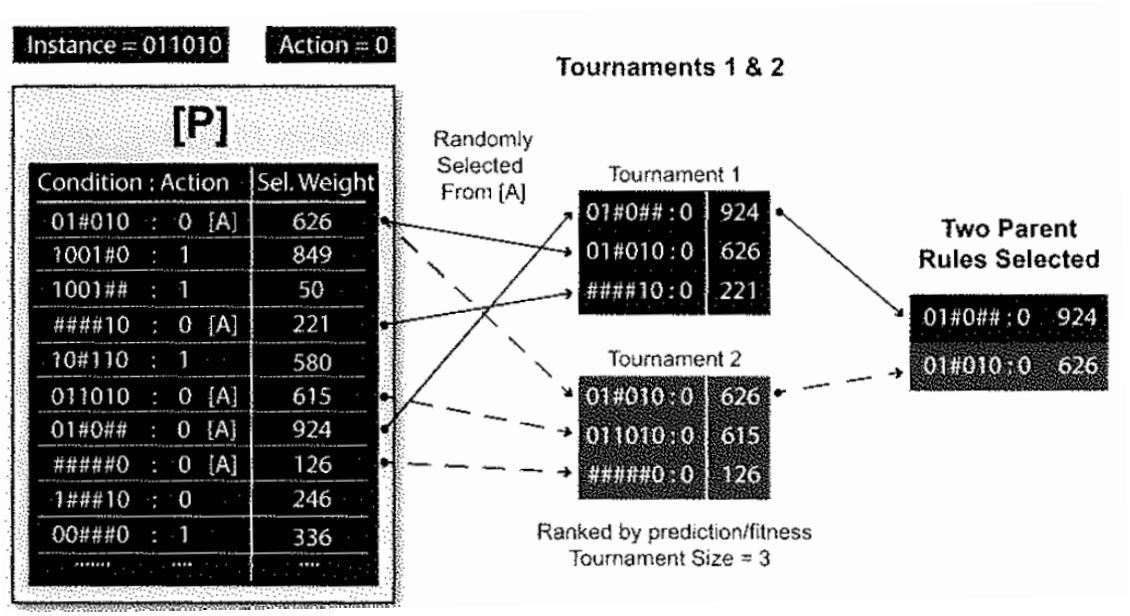


Figure 3.13: Tournament Selection [Urbanowicz and Browne, 2017].

Crossover

Crossover is one of the two ways to evolve new rules from two chosen parents. This is done by considering the features of each parent and utilising one of the three techniques (Figure 3.14) to swap bits between them. This in turn will result in two new rules that can then, after their parameter initialisation, be added to the population as new classifiers.

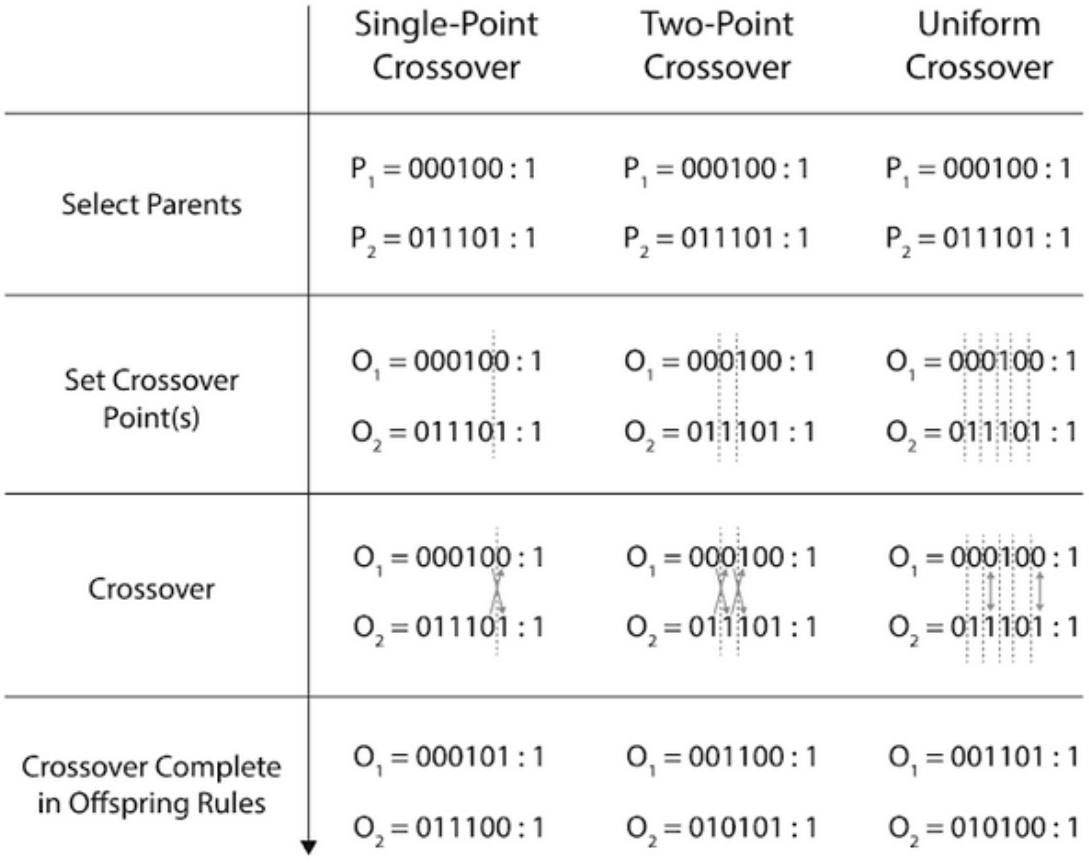


Figure 3.14: The three crossover techniques [Urbanowicz and Browne, 2017].

Despite being inefficient for simpler domains, Uniform crossover is the most commonly used crossover strategy, as it is applicable and works well in all problem domains. For this reason, Uniform crossover was utilised in this project.

3.4 XCS (eXtended Classifier System)

One of the LCS algorithms used in this project is XCS (eXtended Classifier System), developed by Wilson in 1995. XCS is an accuracy-based, Michigan-style LCS developed specifically to be applicable to reinforcement learning problems and differs from other LCS implementations by being an accuracy-based system, rather than a strength-based system, which were popular prior to XCS existence. Figure 3.15 demonstrates the XCS Learning Cycle:

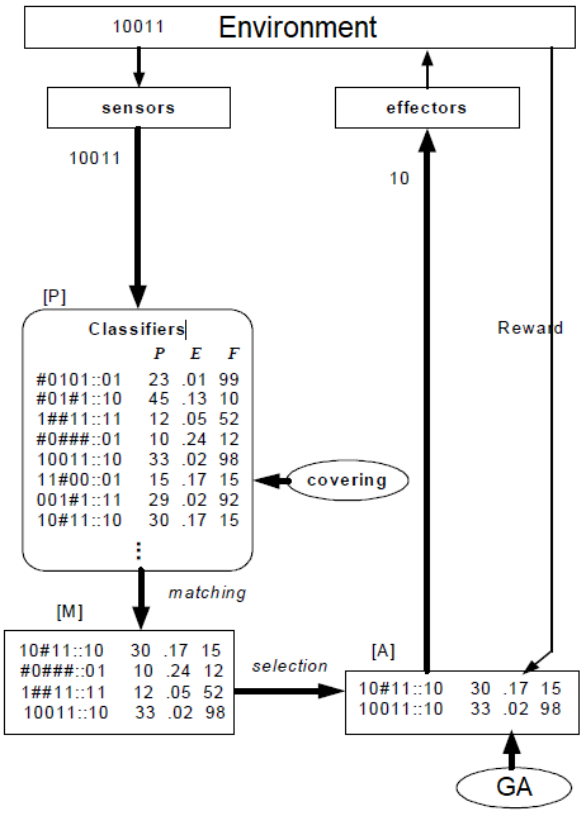


Figure 3.15: Diagram of XCS Functional Cycle [Sanchez et al., 2006].

The steps in the XCS learning cycle are the following:

1. XCS receives a training instance from the environment via one of the sensors.

2. Rules that match the training instance are added into the match set [M]. If no rules are present in the match set, the covering mechanism can be utilised to generate new matching rules.

3. XCS must decide what action to execute. Considering that rules in the match set will not necessarily recommend the same action for execution, the action of the best classifier present in the match set is chosen. In an effort to discover new solutions, the Genetic Algorithm may be employed to select an action for execution via roulette-wheel or tournament selection methods (the latter being most popular in modern LCS). Then, all rules from the Match set that advocate the chosen action are moved to the Action set [A].

4. XCS executes the chosen action and receives reward from the environment as feedback. Based on the feedback, rule parameters are adjusted (see Listing 5.28 and 5.29). Details on how exactly the new values for rule parameters are calculated can be found in [Butz and Wilson, 2001].

5. Finally, the Genetic Algorithm may be called to generate new rules or delete existing rules from the population on the basis of their rule parameters.

Despite being created a respectable amount of time ago, XCS still continues to be a relevant and useful algorithm today. Many modern LCS have used XCS as their base, extending it to fit various scenarios and problems. In-depth analysis of the XCS algorithm is outside the scope of this project, and an algorithmic description and implementation will be provided in Chapter 5 following the publication by Butz [Butz and Wilson, 2001]. More important is the justification of why XCS was chosen to be utilised in this project.

The primary reason for choosing XCS to be one of the two LCS algorithms used in this project is due to the importance and impact that XCS has had on the LCS research area: XCS is widespread and prominent to this day, forms a basis of other numerous LCS algorithms and thus identifying optimal parameters for XCS will automatically make those parameters applicable for other XCS-based LCS for problems of similar scope and size. This will thus cover a very wide range of reinforcement learning LCS algorithms. Additionally, XCS advantages include that it is well understood (due to the original published paper - [Wilson, 1995]), has an easy-to-understand and in-depth algorithmic description ([Butz and Wilson, 2001]), as well as has been the topic of research in many journals and works by numerous authors. Numerous projects in the industry utilised XCS as its LCS of choice - the successful implementation of those projects once again complements XCS maturity and sustainability, making it an ideal LCS to use in this project (see Section 3.10).

3.4.1 Updating XCS Classifier Parameters

This section will explain how the XCS classifier parameters are updated following the reward received from the environment. The updating of XCS parameters can only happen in the Action set. To update its parameters, the XCS classifier needs to compute the updates of reward predictions, error of prediction, and fitness. The formulae below are applicable for when XCS is used for single-step problems; for multi-step problems, updates are different and are related to the Q-Learning method for propagating reward in multi-step problems. This topic is outside this report scope, but if required, more information on updating XCS parameters for Multi-step XCS can be found in [Urbanowicz and Browne, 2017].
Parameter updates in XCS follow the time-weighted recency average (TWRA) principle [Urbanowicz and Browne, 2017], which can be described as:

$$Average_{New} = Average_{Current} + \beta(Value_{Current} - Average_{Current}) \tag{3.3}$$

This is directly inspired by the Widrow-Hoff update, which is commonly used in other areas of artificial intelligence like neural networks for updating the weights of the inputs of the artificial neurons:

$$\nu_{i+1} = \nu_i + \beta(u_i - \nu_i) \tag{3.4}$$

When the value u is received, the equation above is used to record the value of variable ν at each generation i. The value of ν_{i+1} would thus be a combination of the old value of ν_i and the newly received value of u_i. The split balance between using old and new values is controlled by the learning rate parameter β - if β is zero, then no learning takes place as only the old value ν_i is kept; if $\beta = 1$, then only the new value is stored and there is no recollection of the past values. The optimal value for β is therefore in the range from [0, 1] - this has been discussed further in Section 6.7.

The reward prediction for the XCS classifier is updated in the following way:

$$p \leftarrow p + \beta(r - p), \quad 0 \leq \beta \leq 1, \tag{3.5}$$

where p corresponds to reward prediction and r corresponds to the reward gained from the environment (see lines 3-7, Figure 3.16(a)). The error between the actual reward and the reward predicted is computed as (see lines 8-12, Figure 3.16(a)):

$$\epsilon \leftarrow \epsilon + \beta(|r - p| - \epsilon) \tag{3.6}$$

In order to calculate the fitness of an XCS parameter, first the accuracy must be computed (see lines 3-7, Figure 3.16(b)). This can be done via the following formula [Butz and Wilson, 2001]:

$$\kappa = \begin{cases} 1, & \text{if } \epsilon < \epsilon_0 \\ \alpha(\frac{\epsilon}{\epsilon_0})^{-\nu}, & \text{otherwise} \end{cases} \tag{3.7}$$

In XCS, accuracy does not directly influence fitness; rather, a form of fitness-sharing is applied in XCS, where the relative accuracy of each classifier is calculated and compared with other classifiers:

$$\kappa' = \frac{\kappa}{\sum_{cl \in |A|} \kappa_{cl}} \tag{3.8}$$

The fitness classifier (see line 10, Figure 3.16(b)) can thus be calculated and updated using:

$$F \leftarrow F + \beta(\kappa' - F) \tag{3.9}$$

```
UPDATE SET([A], P, [P]):
 1 for each classifier cl in [A]
 2    cl.exp++
 3    //update prediction cl.p
 4    if(cl.exp < 1/β)
 5       cl.p ← cl.p + (P - cl.p) / cl.exp
 6    else
 7       cl.p ← cl.p + β * (P - cl.p)
 8    //update prediction error cl.ε
 9    if(cl.exp < 1/β)
10       cl.ε ← cl.ε + (|P - cl.p| - cl.ε) / cl.exp
11    else
12       cl.ε ← cl.ε + β * (|P - cl.p| - cl.ε)
13    //update action set size estimate cl.as
14    if(cl.exp < 1/β)
15       cl.as ← cl.as + (∑_{c∈[A]} c.n - cl.as) / cl.exp
16    else
17       cl.as ← cl.as + β * (∑_{c∈[A]} c.n - cl.as)
18 UPDATE FITNESS in set [A]
19 if(doActionSetSubsumption)
20    DO ACTION SET SUBSUMPTION in [A] updating [P]
```

(a) Updating classifier parameters in Action set

```
UPDATE FITNESS([A]):
 1 accuracySum ← 0
 2 initialize accuracy vector κ
 3 for each classifier cl in [A]
 4    if(cl.ε < ε_0)
 5       κ(cl) ← 1
 6    else
 7       κ(cl) ← α * (cl.ε / ε_0)^{-ν}
 8    accuracySum ← accuracySum + κ(cl) * cl.n
 9 for each classifier cl in [A]
10    cl.F ← cl.F + β * (κ(cl) * cl.n / accuracySum - cl.F)
```

(b) Calculating classifier Accuracy and Fitness

Figure 3.16: Pseudo code from [Butz and Wilson, 2001]

3.5 UCS (sUpervised Classifier System)

UCS was a logical continuation of the XCS algorithm, developed specifically for supervised learning problems by [Bernadó-Mansilla and Garrell-Guiu, 2003]. The UCS functional cycle can be found in Section 3.3.1 - that section describes closely how UCS works. More details regarding fitness and accuracy calculation, modes of operation, implementation and algorithmic details will be provided in Chapter 5 - the report is especially structured in such a way as to provide a connection between the theoretical pseudocode described in research papers and the actual implementation done for this project. The main difference between XCS and UCS is that UCS makes two important assumptions [Urbanowicz and Browne, 2017]:

1. The correct action can be acquired from the environment. This is a common feature of supervised learning, rather than reinforcement learning.

2. Every state of the environment has a known correct action that is immediately made available to the LCS.

This makes UCS especially more applicable to data mining problems, as well as for knowledge discovery within complex datasets.

UCS was chosen for this project, because it is specifically built for knowledge extraction and creating a model describing the patterns in the datasets, as well as due to the fact that it is the second most common and prominent LCS after XCS. Being a supervised learning algorithm, UCS provides a contrast to reinforcement-learning XCS, what allows to see the effects of various parameter values from a different LCS angle.

3.5.1 Updating UCS Classifier Parameters

Although UCS has similar classifier parameters to XCS (recall Section 3.4.1), in UCS the parameters are updated differently. Each classifier in UCS had an additional classifier parameter known as the *correctTrack* parameter, which is incremented every time a classifier is added into the Correct set. The classifier accuracy can thus be calculated as:

$$accuracy = \frac{correctTrack}{experience}, \tag{3.10}$$

where *experience* refers to the number of times that a classifier has been added to the Match set. In comparison to XCS, the classifier fitness is calculated differently and much simpler:

$$fitness = accuracy^{\nu}, \tag{3.11}$$

where parameter ν determines the pressure on the rules to be accurate (recall Section 3.3.2).

3.6 Other LCS Variants

This section will provide a high-level overview of other important and prominent LCS that have been developed and used to solve real-world problems. A comprehensive list of all major LCS developed from 1978-2008 can be found in [Urbanowicz and Moore, 2009].

Table 3.2 provides the name of the LCS, the paradigm that the LCS follows (Michigan-style (M), Pittsburgh-style (P), Hybrid (H)), how fitness is calculated and what encoding is used for the rules. Finally, a short description is provided to explain each LCS origin and use case.

SYSTEM	STYLE	FITNESS	ENCODING	DESCRIPTION
ZCS (Zeroth-level Classifier System) [Wilson, 1994]	M	Strength	Ternary	Basic classifier system that simplified existing LCS at the time. ZCS proved that much simpler LCS architectures could perform just as well as the early, complex implementations.
S-ELF (Symbolic Evolutionary Learning of Fuzzy Rules) [Bonarini and Basso, 1997]	H	Strength	Binary-Fuzzy Logic	Developed for the activation of fuzzy logic controllers, controlling fuzzy behavior of an autonomous agent.
X-NCS [Bull and O'Hara, 2002]	M	Accuracy	Neural Network	First use of neural network-based representation schemes within XCS. Each rule's condition and action were represented by a neural network.
NCS (Neuro- Classifier System) [Hurst and Bull, 2003]	M	Strength	Neural Network	Used rule structure where each rule was represented by an artificial neural network. The LCS was successful in emerging appropriate rules to allow the agent to navigate simulated mazes. The authors then ported the LCS to a robot platform.
GAssist (Genetic clASSIfier sySTem) [Bacardit, 2004]	P	Accuracy	ADI-Binary	Developed specifically for data-mining - intended to deal with problems that can be solved using very compact rule sets.
EpiXCS [Holmes and Sager, 2005]	M	Accuracy	Ternary	Successor to EpiCS - essentially, EpiCS built on top of XCS, rather than on the original NEWBOOLE LCS. Proved successful in discovering knowledge from clinical research databases.
BioHEL (Bioinformatics-oriented Hierarchical Evolutionary Learning) [Bacardit et al., 2009]	P	Accuracy	ADI-Binary	Successor to GAssist; designed to operate on large bioinformatic datasets and on very complex real-life problems.
Fuzzy-XCS [Casillas et al., 2007]	M	Accuracy	Binary-Fuzzy Logic	Development proposal of a fuzzy XCS system for single-step reinforcement problems. Fuzzy-XCS should evolve fuzzy models whilst retaining interpretability and maintaining accuracy.
Fuzzy-UCS [Orriols-Puig et al., 2009]	M	Accuracy	Binary-Fuzzy Logic	Similar justification to Fuzzy-XCS, but specifically designed for supervised learning tasks.
ExSTraCS 2.0 (Extended Supervised Tracking and Classifying System) [Urbanowicz and Moore, 2015]	M	Accuracy	Ternary	Created to tackle classification, prediction, and knowledge discovery problems in complex and noisy problem domains.

Table 3.2: Table to show several prominent LCS and their descriptions.

3.7 LCS Example Run

This section will present an easy to understand example of a single run of a Michigan-style LCS. The main purpose of this demonstration is to show how an LCS operates on a very high level - this example does not implement any particular LCS variant, as those are much more complex and contain substantial differences between each other.

Consider an environment resembling a chessboard, consisting of boxes that can be either filled (reserved) or unfilled (free). Assume a robot or agent wants to traverse the environment in all directions a single step at a time, avoiding the filled boxes (Figure 3.17). By numbering boxes 1-8, starting from the top left box, it is possible to map the environment, where each bit in the condition represents whether the corresponding adjacent box is filled or not, and the action represents the number of the next square for the robot to move to (Figure 3.18).

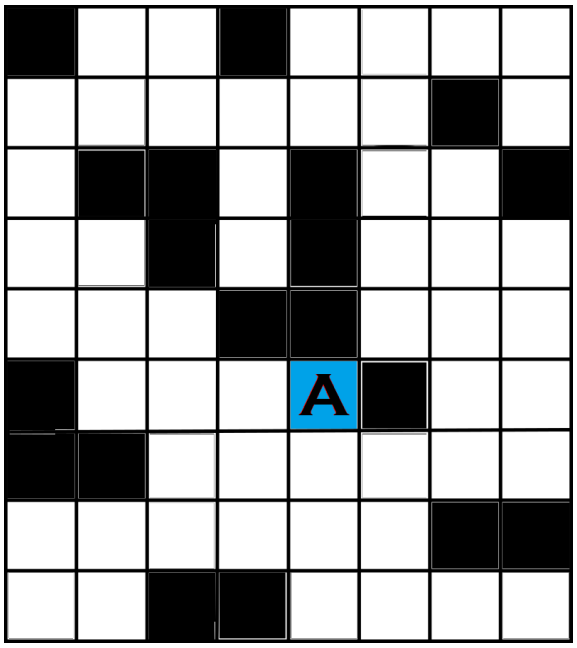


Figure 3.17: Visualisation of one of environment states.

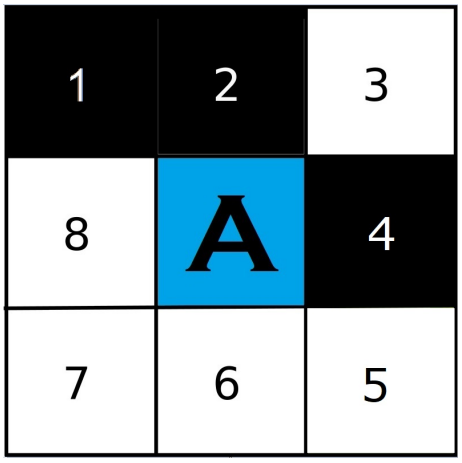


Figure 3.18: Numbering of boxes adjacent to agent.

For example, classifier 11010000 → 6 could be read as:

```
IF      BOX#1=Filled
        BOX#2=Filled
        BOX#4=Filled
THEN    MOVE TO BOX#6
```

Figure 3.19: Example of a rule interpretation

Incorporating the wild card symbol (#) allows to make classifiers more general and applicable to larger amount of environment states - for example, 11#1#0## → 6 would indicate that regardless whether boxes numbered 3, 5, 7, 8 are filled or not, the agent should always move to box number 6.

For the sake of simplicity of explanation, several classifiers within the population have been initialised (at the early stages of the LCS run, this would have been done by the covering mechanism, as the LCS would start with an empty population). Notice the addition of "weight" - in a real LCS, weight is not a single value and several run parameters determine the "weight", i.e. how "good" a particular classifier is.

Table 3.3: Initialised classifiers with weight.

Condition	Action	Weight
#0######	→ 2	0.5
#1#0#1##	→ 3	0.2
0#11#0##	→ 8	0.4
1#00#0##	→ 5	0.3
#1#1#0##	→ 2	0.9

After the LCS receives the environment state, all applicable rules that match the state are selected. Two rules are applicable to the state shown in Figure 3.18 - 0#00#0## → 5 (Rule 1) and #1#0#0## → 2 (Rule 2). The LCS must choose one of the two rules as a course of action for the agent, with the probability proportional to the ratio of individual classifier weight and the total weight of chosen classifiers:

$$Probability(Rule1) = \frac{Rule1.weight}{Rule1.weight + Rule2.weight} = \frac{0.3}{0.3 + 0.9} = 0.25 \tag{3.12}$$

$$Probability(Rule2) = \frac{Rule2.weight}{Rule1.weight + Rule2.weight} = \frac{0.9}{0.3 + 0.9} = 0.75 \tag{3.13}$$

Since Rule 2 has a higher probability of being chosen, assume that Rule 2 has been accepted and the agent is being recommended to proceed to box number 2. But, as can be seen from our environment (Figure 3.18), box number 2 is coloured, therefore the agent can not move there; since the suggestion given was incorrect, Rule 2 weight will be lowered, and Rule 1 will be chosen as the correct action to take given the environment state.

In order to improve the behaviour of the robot, a genetic algorithm is used to modify the rules before inserting them back into the population. For example, assume two rules were chosen:

Table 3.4: Classifiers chosen for crossover.

Condition	Action	Weight
#1#0#1##	→ 3	0.2
0#11#0##	→ 8	0.4

By applying uniform crossover, two new rules can be produced.

Table 3.5: Classifiers after crossover.

Condition	Action	Weight
0#10#1##	→ 8	0.2
#1#1#0##	→ 3	0.4

The actions above may be continued in a cycle until the behaviour of the robot is appropriate and produces an acceptable error rate.

3.8 Knowledge Discovery vs. Prediction

Learning classifier systems can be used in two ways - to discover new knowledge, i.e. patterns and relationships, within a domain, or to predict/classify the outcome of a previously unseen state.

In knowledge discovery, the resulting rule population is used to provide domain experts with an understanding of the patterns and relationships in data - a key requirement of a rule population for knowledge discovery is small size, in order to make it comprehensible. Essentially, in such scenarios LCS is used as a **data mining technique**. Note: for such LCS applications, population compression is crucial in order to produce a small enough population list to be easily comprehensible by a human operator.

In prediction scenarios, the rule population is used to classify (i.e. determine the action) of a previously unseen instance from the environment. In such scenarios, LCS are used as a **classification technique**. In such cases, population compression is not crucial for the operation of the algorithm (in fact, keeping the population uncompressed demonstrated a slightly better testing accuracy in comparison to when it was compressed), but may be helpful for interpretability.

In this project, XCS and UCS are used within a prediction scenario - 80% of the available data is used for training, whilst the rest 20% are used as testing data that needs to be classified/predicted ($k = 5$ in k-fold validation, see Section 5.3.3). Prediction scenario was chosen, because it is a more complex form for LCS to solve and therefore would provide more interesting and useful results.

3.9 Single- vs. Multi-step Learning

This project will focus solely on single-step domains, but it is useful to explain what that means and what other alternatives for LCS exist. In single-step learning, the reward or the best action corresponding to an environment condition or state is known straightaway. This is the most basic type of learning, and a lot of LCS domains successfully operate in single-step domains. Specifically due to single-step learning simplicity and prevalence in many LCS-related domains, this mode of learning was used in this project.

An alternative to single-step learning is multi-step learning. In multi-step problems, the reward or best action for a condition is delayed and is given only following a specific number of steps. Multi-step problems also branch out into Markovian and non-Markovian domains, but analysis of those is outside the scope of this report.

3.10 Applications of LCS in Industry

Learning classifier systems have found a wide range of applications within real-world applications - the only crucial requirement for LCS to be applicable is for the domain and data to contain generalisable patterns. Real-world applications utilising LCS can be separated in the following domains, although this list is not conclusive: adaptive-control systems, data mining, function approximation, gaming, image classification, medical diagnosis, modelling, navigation, optimisation, prediction, robotics. Some prominent LCS real-world use-cases will be described in this section - they will demonstrate how flexible and useful LCS can be in a plethora of various and unrelated industries, and how effective it can be in solving the challenges of the modern world.

3.10.1 Data Mining

MONK Problems and Epidemiology

In *Data Mining using Learning Classifier Systems* [Barry et al., 2004], XCS and other LCS have been used to produce interpretable solutions following data mining on several data sets. XCS produced very good results on all three MONKs[3] problems, registering comparable performance to neural networks on MONK 1 problem, 100% classification accuracy and optimal performance on MONK 2 problem, and 95% accuracy on MONK 3 test with 5% noise presence. Conclusion was made that XCS has the ability to operate on the level of existing neural networks at the time and achieve good performance, making it applicable as a data mining technique. The paper proceeded with experiments on using LCS for Knowledge Discovery in Clinical Research Databases, this time utilising EpiCS [Holmes et al., 2000] (an LCS developed for epidemiologic surveillance) to undertake analysis on the Fatal Accident Reporting System[4] (FARS) data. Utilisation of EpiCS was successful in training the

[3]The MONKs problem were the basis of a first international comparison of learning algorithms.

[4]Large database provided and maintained by National Highway Traffic Safety Administration (NHTSA) of the United States Department of Transportation. Provides documents for all fatal crashes involving automobiles occurring in the United States and Puerto Rico since 1975.

system using the aforementioned data, creating understandable rules that in their "IF.-THEN" form suggested previously unseen relationships within accidents (see Figure 3.20 - Rule 1 could be expected, but Rule 2 was a surprise, as previously it was believed that speed limits set on urban roads should reduce amount of fatal accidents). Classification performance of EpiCS has been excellent (98% accuracy),

```
1.  IF      ROLLOVER=Yes or
            EJECTED=Yes or
            FIRE=Yes
    THEN    FATALITY=Yes

2.  IF      URBAN HIGHWAY=Yes
    THEN    FATALITY=Yes

3.  IF      COLLISION ≠ MOTOR VEHICLE or
            ALCOHOL=No or
            DRUGS=No
    THEN    FATALITY=No
```

Figure 3.20: Some rules extracted using EpiCS [Barry et al., 2004].

passing that of See5 Tool[5] (97%). Risk models built by the LCS have also performed better (97%) than those built using logical regression (89%). This experiment was preceded by another one, in which EpiCS was able to successfully derive a continuous measure of disease risk in a series of 250 individuals [Holmes, 1998].

Conclusions were made that LCS, specifically EpiCS, is a capable technique for extracting information from medical databases and using it to suggest future actions. Since EpiCS originated from the NEWBOOLE LCS, authors suggested that rewriting EpiCS using the XCS LCS paradigm will further improve results. This has been identified as the case when EpiXCS was developed - EpiXCS improved the systems stability in comparison with EpiCS, was able to better cope with noise and missing data, as well as evolve more accurate and generalised population of rules.

[5]See5 is a sophisticated data mining tools for discovering patterns that delineate categories, assembling them into classifiers, and using them to make predictions. [Quinlan, 1998]

3.10.2 Modelling and Optimisation

Discovering New Fighter Aircraft Manoeuvres

In *The Fighter Aircraft LCS: A Real-World, Machine Innovation Application* [Smith et al., 2004], LCS was utilised to simulate and discover new fighter aircraft combat manoeuvres. This LCS-powered real-world application was a long-running United States Air Force project first described in [Smith et al., 2000] and was very successful, essentially creating a system that can duplicate a test pilot and discover new complex manoeuvres similarly as to how a test pilot would. This is one of several early works where the primary goal was generation and discovery of new information, rather than consolidation or discovery of patterns within the domain.

The problem being solved set serious challenges for LCS and computational requirement at the time. Below is the extract of the problem scenario, as described by the original authors:

> *"Consider the basic problem of one-versus-one fighter aircraft combat. Two aircraft start their engagement at some initial configuration and velocity in space. In our simulations, an engagement lasts for a pre-specified amount of time (typically 30 seconds). An engagement is divided into discrete time instants (in our simulation, 1/10th second instants). At each instant, each "aircraft" must observe the state of the environment, and make a decision as its own action for that instant. A score for a given aircraft can be calculated at the end of an engagement by comparing that aircraft's probability of damaging its opponent to its own probability of being damaged."*
>
> *- [Smith et al., 2000]*

Of special interest is the encoding of the environment conditions and actions - the encoding utilised the standard ternary alphabet {1, 0, #}, yet incorporated elements of fuzzy encoding to represent notions of "low, medium, high" within the environment. Additional complexity stemmed from the fact that certain actions, or sequence of actions, may be physically impossible to execute under certain conditions and situations (due to aerodynamics). The authors were able to overcome this by rejecting the idea of "deterministic" actions and treating the actions given by LCS as "suggestions" - actions were "suggested" to the agent for execution and the agent responded with the nearest feasible response to that of the suggestion. Within the simulation environment, this was achieved via AASPEM[6], which acted as a sort of action sanity-checking mechanism.

[6]The Air-to-Air System Performance Evaluation Model, industry standard model for fighter aircraft combat.

	Bins (represented in binary, plus the # character)							
3 bits:	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Own Aspect Angle (Degrees)	< 45	45 to 90	90 to .35	135 to 180	180 to 215	215 to 270	270 to 315	315 to 360
Opponent Aspect Angle (Degrees)	< 45	45 to 90	90 to 135	135 to 180	180 to 215	215 to 270	270 to 315	315 to 360
2 bits:	**00**	**01**	**10**	**11**				
Range (1000 feet)	< 1	1 to 4.5	4.5 to 7.5	> 7.5				
Speed (100 knots)	< 2	2 to 3.5	3.5 to 4.8	> 4.8				
Delta Speed (100 knots)	< -.5	-.5 to .5	.5 to 1	> 1				
Altitude (1000 feet)	< 10	10 to 20	20 to 30	> 30				
Delta Altitude (1000 feet)	< -2	-2 to 2	2 to 4	> 4				
Climb Angle	< -30	-30 to 30	30 to 60	> 60				
Opponent Climb Angle	< -30	-30 to 30	30 to 60	> 60				
Total: 20 bits								

Figure 3.21: Aircraft environment encoding. [Smith et al., 2000]

	Action Values (suggested to AASPEM)							
3 bits:	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Relative Bank Angle (Degrees)	0	30	45	90	180	-30	-45	-90
Angle of Attack (Degrees)	0	10	20	30	40	50	60	70
2 bits:	**00**	**01**	**10**	**11**				
Speed (100 Knots)	1	2	3.5	4.8				
Total: 8 bits								

Figure 3.22: Aircraft suggested action encoding. [Smith et al., 2000]

Encoded Rule:
01010010110111010010 → 11010010

```
IF
        (90 > OwnAspectAngle > 35) AND
        (215 > OpponentAspectAngle > 180) AND
        (7.5 > Range > 4.5) AND
        (Speed > 4.8) AND
        (0.5 > DeltaSpeed -0.5) AND
        (Altitude > 30) AND
        (2 > DeltaAltitude > -2) AND
        (ClimbAngle < -30) AND
        (60 > OpponentClimbAngle> 30)
THEN ATTEMPT TO OBTAIN
        (RelativeBankAngle = -45) AND
        (AngleOfAttack = 40) AND
        (Speed = 3.5)
```

Figure 3.23: Example of rule/manoeuvre from encoding [Smith et al., 2000].

City Traffic Capacity Balance

In *Traffic Balance using Classifier Systems in an Agent-based Simulation* [Hercog, 2004b], XCS was used to create a best distribution model for agents representing different transport types within a city. The primary challenge that is being solved within the paper is balance of capacity for different transport modes, which is an increasingly complex task as the city size grows. The authors used the Multibar Problem[7] [Hercog, 2003] as an abstraction problem for traffic capacity, each bar within the original problem representing a transport mode, e.g. metro, car, pedestrian. An original approach within this paper was utilisation of MAXCS [Hercog and Fogarty, 2000], a multi-agent system composed of individual agents, where each agent individually utilises XCS to learn - this approach was required in order to encapsulate and bring together all the individual agents into a single system.

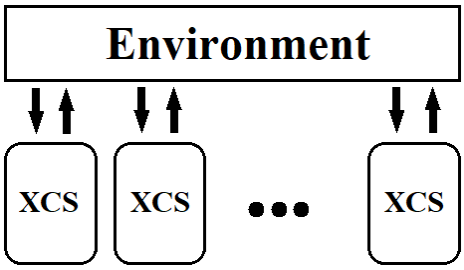


Figure 3.24: MAXCS: each agent learns via individual XCS [Hercog, 2004b].

The results of the project were encouraging - the agents were able to evolve rules to achieve transport equilibrium; a state, as per [Nash, 1950], where agents are choosing the best action possible, yet consider other agents' actions, and eventually can no longer achieve payoff improvement. Several important conclusions were made, notably that LCS can be used for studying and achieving optimum performance in multi-agent systems, as well as that they are applicable for studying individual agent behaviour within a global system. Interestingly, this work has found partial practical implementation in Mexico City in 2003, where a forecast system was introduced on important streets and roads to monitor the traffic volume and fetch weather forecast from the news; the government could then make alternative route suggestions to citizens in accordance with congestion levels and their impact on travel time for the following day [Hercog, 2004a].

[7]Deriving from "El Farol" Problem [Arthur, 1994] and the Minority Game [Challet and Zhang, 1997], agents must decide between attending a specific bar from a given set of bars, or stay at home, making decisions based on attendance numbers from previous weeks. Choosing to stay at home when bar is "overcrowded" or choosing to go when bar is "empty" will yield high levels of reward. The aim is to simulate a realistic society environment, where several choices are present, best-choice decisions must be made and are encouraged via rewards.

3.10.3 Function Approximation

The basis of mathematical models, describing numerous processes in physics, chemistry, biology or economics, is formed by equations of varying type: linear equations, differential equations, etc. Solving such equations requires the ability to calculate the function's value y given the input argument x. For complex models, such calculations can be very time-consuming and computationally expensive. Functions used for mathematical modelling can be given as a formula, but can also be represented via a table, where the function is known only for given discrete values of the argument. Specifically, such would be the case if the calculations were being run on a computer or have been found during an experiment. In practice, often a situation occurs where a function value for an argument not specified in the table must be known - but such value may only be obtained via complex calculations or through expensive experiments. The problem being solved can thus be summarised as calculating approximate values of the function at any given argument based on initial data specified in the function table.

This problem can be solved by substituting complex function $f(x)$ with a simpler function $\phi(x)$ that can be easily calculated for any given argument x - the function $\phi(x)$ could then be used for approximating values for $f(x)$. This is known as *function approximation* (from Latin *approximo* - come closer, approach). Function $\phi(x)$ is built in such a way that the differences between $\phi(x)$ and $f(x)$ over a given domain are minimal - the notion of what constitutes as sufficiently minimal difference is dependent on how the difference between the two functions is calculated, and therefore varies between different approximation methods and implementations.

XCSF (XCS for Function Approximation) was an LCS introduced by Wilson in 2001 with the aim of investigating whether XCS could be used for function approximation [Wilson, 2002]. This was a significant addition to the LCS family. Building upon and extending XCS, XCSF made several changes to the original XCS algorithm: the classifier condition was changed to accept real-valued input rather than ternary, where the pair of integers in the condition ($int_i = (l_i, u_i)$) specified the lower and upper boundaries of the accepted input respectively, i.e. an input x_i to the classifier would only match if $l_i \leq x_i \leq u_i$ for all x_i; the prediction component was modified to no longer predict single values, but rather calculate the prediction from the aforementioned input using linear approximation functions; and finally, the Action set was removed and it's functionality replaced by the Match set [Butz, 2015]. See Figure 3.25 for a diagram of XCSF iterative learning process. XCSF demonstrated good results when used for Piecewise-Constant Approximation (PCA) and Piecewise-linear Approximation (PLA). Both will be discussed further.

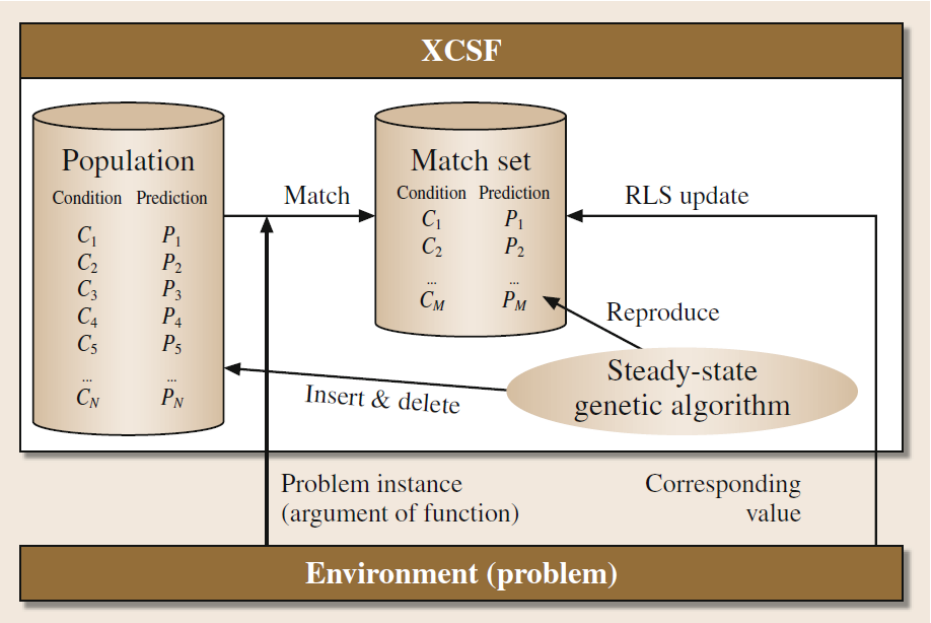


Figure 3.25: XCSF iterative learning process [Butz, 2015]

Piecewise-Constant Approximation (PCA)

PCA is the simplest type of approximation that can be done using XCSF - every function to approximate is represented in the form of $y = f(x)$, where y represents the payoff and x corresponds to the input. The idea is that after a thorough sampling of the problem input space, the prediction value should, given a certain as input, predict the value of y accurately - the closeness of the prediction could be adjusted by the error threshold ϵ_0 parameter, where value of ϵ_0 is proportionate to the closeness of the approximation [Wilson, 2002]. Figure 3.26 demonstrates the results achieved when using XCSF for approximating function $y = x^2$ over the input interval [0, 99]. The LCS used a dataset of 1000 (x, y) pairs, with x chosen at random and the y being the corresponding function value. For plotting of the prediction, an (x, y) pair was drawn at random from the dataset: x was used as XCSF input, whilst the value of y was used as a reward. All classifiers matching input x were inserted into the Match set [M], where the system prediction was calculated and adjusted using the reward y. At this stage, the genetic algorithm could be run if required. To obtain Figure 3.26, the cycle above was run 50,000 times [Wilson, 2002].

The figure above allows to make several conclusions regarding XCSF ability to approximate functions: the step-like shape of the prediction curve suggests that during the "flat" phase of the step, control is exerted by the classifiers in the Match set. Every "jump" or "step" means that a different set of classifiers has taken over the control. This is expected behaviour and demonstrates the ability of XCSF to distribute classifier resources efficiently over the domain. Finally, the average error in the graph is around 500 - this is consistent with the value specified for the error threshold ϵ_0, and so is inline with the belief that ϵ_0 can be used to control the closeness of approximation.

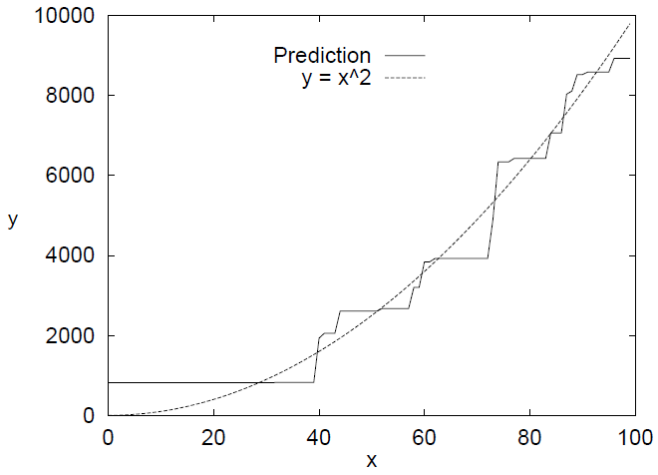


Figure 3.26: Piecewise-constant approximation for $y = x^2$, $\epsilon_0 = 500$ [Wilson, 2002].

To conclude, XCSF is able to successfully execute Piecewise-constant approximation, using the classifier resources well and employing the ϵ_0 to state the closeness of approximation. Nonetheless, PCA is quite a primitive approximation and is rarely used in real-life problems, where much better accuracy and following of original function contour is required.

Piecewise-linear Approximation

Assume there is a function defined as a set of argument-value pairs in a table. In an attempt to approximate this function at point x, let us identify two neighbouring arguments in the table on either side of that point, e.g. $x_k < x < x_{k+1}$. The corresponding function values at those points will thus be $y_k = f(x_k)$ and $y_{k+1} = f(x_{k+1})$. *Assume* that on the interval (x_k, x_{k+1}) this function can be represented with sufficient accuracy by a linear function (see Figure 3.27).

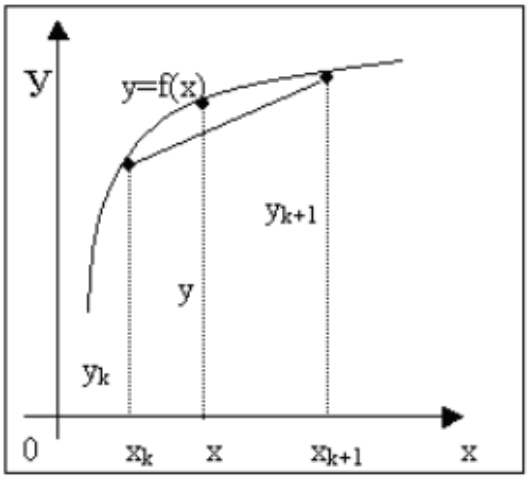


Figure 3.27: Linear interpolation, also known as piecewise-linear approximation [Wilson, 2002].

The equation of the approximating straight line passing through points (x_k, y_k) and (x_{k+1}, y_{k+1}) can be described as:

$$\frac{y - y_k}{y_{k+1} - y_k} = \frac{x - x_k}{x_{k+1} - x_k}, \tag{3.14}$$

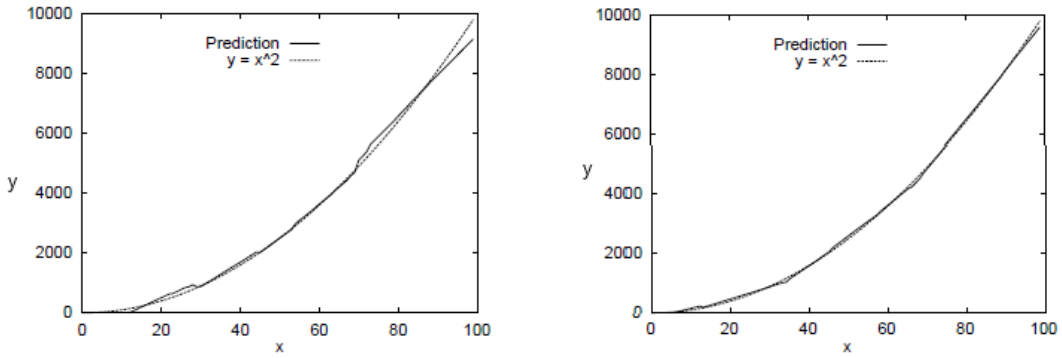
or in a more traditional form:

$$y = y_k + \frac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k) \tag{3.15}$$

The challenge for XCSF is therefore to predict the equation of the approximating line $h(x)$, and then use the input x to calculate the actual prediction given the $h(x)$ [Wilson, 2002]. For XCSF, equation 3.15 can be adapted as $h(x) = w_0 + w_1 x_1$ for 1-dimensional functions, where w_1 represents the slope of the straight line and w_0 represents the y-axis intercept, and $h(x) = w \cdot x'$ for n-dimensional functions, where w is the weight vector and x' is the input vector. [Wilson, 2002] contains additional mathematical theoretical base on the operation of XCSF, but this is outside the scope of this project.

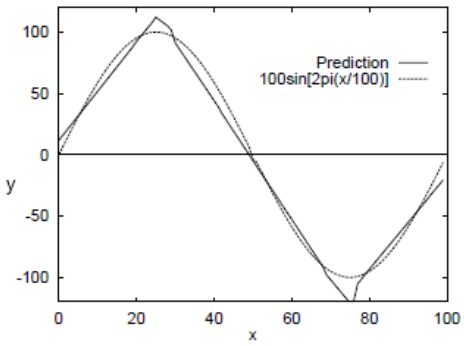
In order to start using XCSF for piecewise-linear approximation, only minor changes were necessary for the original algorithm, specifically weight vectors had to be added to the classifiers, as well as ability to calculate predictions via the above formulas per cycle step.

XCSF performed well, showing good approximations for 2-line functions, parabolas, and sine function. The authors again experimented with different values for the error threshold, and demonstrated that it can be used to determine the closeness of approximation. Results of the experiments done on parabolic and sine functions can be seen in the Figures below:

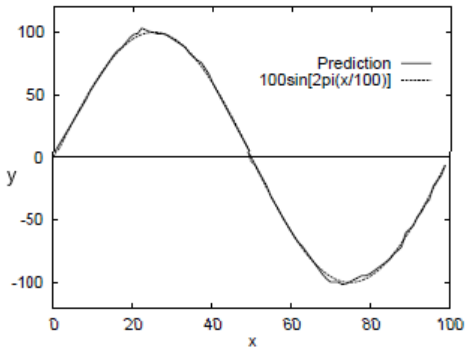


(a) $y = x^2$, $\epsilon_0 = 500$

(b) $y = x^2$, $\epsilon_0 = 100$



(c) $100sin(\frac{\pi x}{50})$, $\epsilon_0 = 10$



(d) $100sin(\frac{\pi x}{50})$, $\epsilon_0 = 2$

Figure 3.28: Piecewise-linear approximation for parabolic and sine functions [Wilson, 2002].

To conclude, through the undertaken experiments it is evident that XCSF is capable of approximating functions of various complexity in an efficient manner. The highest complexity tested by the authors were six-dimensional nonlinear functions, and even there XCSF continued to evolve highly accurate approximations. Thus, one can conclude that using learning classifier systems, in particular XCSF, is a viable approach to approximating linear and non-linear functions.

3.10.4 Game Industry

Application of classical artificial and computational intelligence techniques, such as finite-state-machines, decision tress, scripting, to games has a long and successful history. But everything is evolving, and the gaming industry is no exception - more advanced techniques and algorithms are required to improve the behaviour of artificial agents within the game environment, make it more realistic and creative. Learning Classifier Systems may provide a solution for this demand.

In *A Survey of Learning Classifier Systems in Games* [Shafi and Abbass, 2017], a comprehensive survey of LCS utilisation in the game industry is provided, with the authors dividing games into three distinct groups: video games, combinatorial games and simulation games. The implementation of LCS for video games will be discussed in this section, as well as a critical evaluation of the advantages and disadvantages.

LCS in Video Games

The majority of video games, if not all, contain non-playable characters (NPC's) or some kind of agents[8] that are controlled by artificial intelligence. Regardless of the video game genre, whether it is a first-person shooter (FPS), role-playing game (RPG) or a strategy games, in all of these genres NPC's play an important role within the gameplay. [Mac Namee and Cunningham, 2001] devised several characteristics of a potentially successful agent action selection mechanism for a video game: it must be reactive (NPC's must behave in accordance to event-action rules), proactive (NPC's must be working towards a concrete goal), autonomous (NPC's must operate independently from the player or the game master) and, finally, be configurable and maintainable even by a non-programmer. Learning Classifier Systems can accommodate for such requirements. An obvious use case for LCS would therefore be to model NPC's or agents behaviour and actions in the game world.

A fair amount of research has been conducted in the area of using LCS for modelling NPC game behaviour. [Robert et al., 2002] was one of the earliest projects that resolved around modelling NPCs in a Massively multiplayer online role-playing game (MMORPG) called Ryzom (Figure 3.29). A multi-layer architecture, *MHiCS (A Modular and Hierarchical Classifier System)*, was developed, in which every layer consisted of several LCS working together to determine the agents actions. Layer 1 (Motivational layer) determined the motivation of the NPC, for example self-protection, hunger or flocking. Each motivation is controlled by a single independent LCS, but the resulting actions are shared between the three motivations. Actions triggered in Layer 1 are then used to trigger Layer 2 (Common LCS Layer), where a new LCS is activated depending on the action received from Layer 1. Layer 2 is used to determine the Motivational Intensity, which uses Layer 1-computed motivation and fitness. A classifier triggered by several motivations in Layer 1 will have a higher chance of getting chosen, and vice-versa. Layer 3 (Action Layer) is responsible for determining the action for the agent to take, whilst Layer 4 (Action Resources Layer) is responsible for providing resources to execute the action and display the NPC behaviour. The full MHiCS cycle can be seen in Figure 3.30.

[8] NPC's and agents will be used interchangeably within this text.

Figure 3.29: Screenshot from Ryzom. Ryzom is an online game played over the Internet by thousands of players simultaneously [Robert et al., 2002].

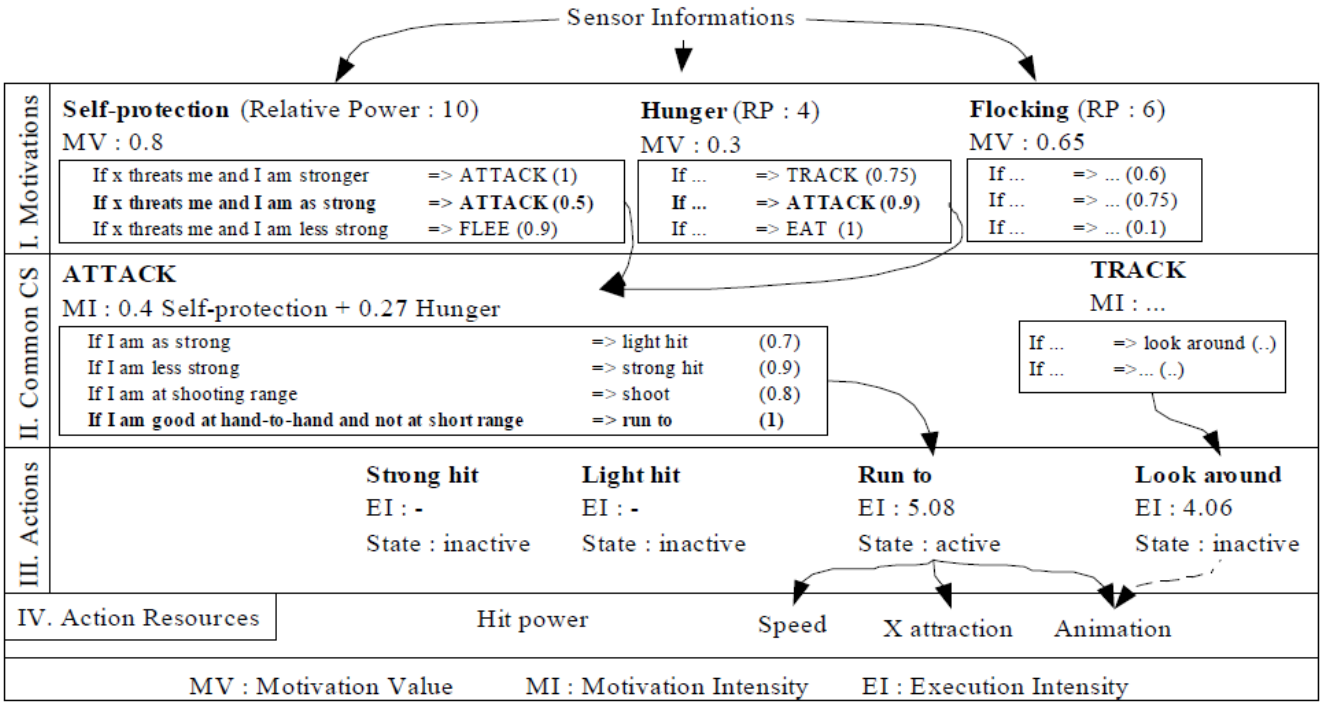


Figure 3.30: Illustration of the MHiCS cycle. For simplicity, LCS are only shown with their classifier list, and the classifier condition and action are demonstrated as plain text instead of how they would originally look like in (0,1,#) encoding [Robert et al., 2002].

Following several experimental runs, the authors concluded that MHiCS was successful in its task of modelling NPC behaviour. Personality attribution to NPC's was simplified, as well as in all experiments the original NPC motivations generated the correct chain of actions. MHiCS essentially proved that using LCS for NPC action selection is an appropriate and viable approach. The authors carried out further experiments involving the MHiCS architecture - in [Robert and Guillot, 2003], MHiCS was used in a different game[9] to demonstrate the advantages of using MHiCS for modelling bot behaviour and teaching bots correct actions in comparison to hand-tuned HPB bots[10] that were usually used in similar game scenarios. In [Robert and Guillot, 2006], a more theoretical and mathematical justification of the success of MHiCS was provided, with the authors conclusion being that MHiCS and its multi-layered architecture provide an efficient decisional autonomy for NPCs and agents in highly non-deterministic environments of real-time video games [Robert and Guillot, 2006].

There have been other attempts at using LCS for modelling NPC behaviour. [Sanchez et al., 2006] developed a behavioural animation framework called *ViBes (Virtual Behaviours)*, which utilised XCS within a rather complex architecture to learn rules appropriate to different NPC behaviours, eventually resulting in a model that was used to successfully determine agent behaviour and actions in a virtual kitchen environment [Shafi and Abbass, 2017]. An interesting detail and implementation advantage emerging from the project was that the LCS-based NPC was able to provide reactive behaviour in parallel to planning a long sequence of actions required for the NPC to undertake in the given environment. An interesting project was undertaken by [Kop et al., 2015], in which a new method called *EDS (Evolutionary Dynamic Scripting)* was proposed: EDS borrows concepts from LCS, but has two important changes - instead of a genetic algorithm, EDS uses tree representations for rules and uses genetic programming for evolving the rules, as well as uses *Dynamic Scripting* to replace the Q-Learning-based reinforcement learning component [Shafi and Abbass, 2017]. One of the advantages of these changes is that it allows for easier integration of EDS into existing game development libraries, many of which are based around or contain some type of finite-state machines, decision/behaviour trees, etc. EDS was used and tested in an air combat simulation game, which demonstrated that EDS was able to produce better NPC behaviour and also evolve previously unknown rules and patterns. Finally, [Small and Congdon, 2009] used a Pittsburgh-style LCS with the goal of developing an autonomous NPC capable of learning and making decisions quickly within a dynamic environment. The environment chosen for the experiment was *Unreal Tournament 2004*, a fast-paced FPS game very popular during its time. The NPC accepted multiple high-level conditions from the environment and determined an appropriate single action to undertake given the circumstances. The project was a success, with the NPC bot gradually improving over time and able to compete on-par against an NPC using the original in-built AI in the game.

[9] *Capture The Flag* scenario in *Team Fortress Classic*, a modification of the FPS game *Half-Life*.
[10] [Broome, 2004]

Figure 3.31: Screenshot from *Unreal Tournament 2004* [Small and Congdon, 2009]

.

LCS Advantages and Disadvantages in Video Games

A logical question arises - if LCS are so applicable to modelling NPC behaviour across such a wide variety of game genres, why has that not become the standard? This is a difficult question to answer, because there can be several limiting factors. Firstly, game development is not usually done from scratch, but utilises libraries that have been compiled previously. Once the AI component within those libraries has been programmed, it is rare for it to change dramatically - it can definitely be improved game to game, but introducing a completely new paradigm like LCS within existing libraries poses a serious challenge. Additionally, using LCS can be quite slow and computationally-intensive - what is an important limiting factor, especially in large and complex games that are dominant in the gaming industry today.

3.10.5 Control

Using LCS for Data-Mining in a Steel Hot Strip Mill

In *The Development of an Industrial Learning Classifier System for Data-Mining in a Steel Hop Strip Mill* [Browne, 2004], a description is provided of how LCS can be used to control a steel hot strip mill[11] and how the population of rules produced can enable operating engineers to learn more about the system. The main challenge of this task was that the industrial controls for a strip mill are very dynamic and can contain excessive amounts of noise, rendering traditional means of control very difficult to utilise or even entirely inapplicable. Essentially, the strip mill environment can be described as a "complex environment", which, as summarised by [Booker et al., 1989], contains following features:

- Data sets can contain noise, be incomplete and/or very large.
- Actions may be required only following a long sequence of steps.
- Goals towards which the environment is working may defined implicitly or vaguely.
- Complex environment parameter relationships.
- Lack of complete knowledge about the domain.

In contrast to other available techniques at the time (the project lasted a decade, from 1994 to 2004), LCS was most suited for the environment described above: LCS were able to work with noisy data to learn patterns from the environment [Fogarty, 1988], they did not require knowledge of the environment domain [Goldberg, 1989], LCS could issue or delay actions for as long as needed, acting either immediately or following a long sequence of events, whatever was required for the problem being solved [Wilson and Goldberg, 1989]. Another great advantage of using LCS above other techniques was the link that LCS provides between conditions and actions - rules can be understood and validated by engineers, and eventually could be used to understand system behaviour, improve system performance, thus improving the mill availability and product quality [Browne, 2004].

A new LCS, called *iLCS*, was developed specifically for the mill scenario. It made several improvements to the original LCS architecture, making it more relevant to the industrial domain. The performance results have been divided by the authors into two categories - how well the LCS was able to extract rules from the environment (Discovery) and how well LCS was able to generate accurate predictions for the mill operators to consider (Advisory) - it was decided that engineers must sanity-check the system predictions and apply them manually if they are appropriate, as the cost of mistake was too high to allow the LCS to act by itself.

[11] Mill used to convert a steel block or slab into a thin sheet, also known as strip.

iLCS did not obtain satisfactory performance during the Discovery stage - despite learning something, the LCS was not able to converge to an optimum solution. Computational power was an important limitation of the LCS - training time took 29 hours to complete 2 million iterations for a data set containing 64 parameters. Nonetheless, expert confirmation was acquired that the LCS was able to generate some good rules - although those were already known prior (most being general rules of thumb, common practices), this confirmation was important in attesting the ability of LCS to if not solve the problem, then at least solve it partially.

iLCS performed worse than what was expected at the Advisory stage, too - prediction accuracy was only 60%, what was deemed as insufficient by the mill operators. The LCS was overwhelmed with the amount of data and dimensionality of the domain, and thus failed to identify more good rules and patterns within the search space.

This project is difficult to call a success, but it nonetheless provided a plethora of valuable information that later benefited future industry-grade LCS. The project was simply too ambitious for its time [Browne, 2004]; possibly, the authors would be able to achieve better results had they decided to repeat their project today.

Using LCS to minimise loss in Electric Power Distribution Networks

In *Application of Learning Classifier Systems to the On-Line Reconfiguration of Electric Power Distribution Networks* [Vargas et al., 2004], Holland LCS [Holland, 1992] was used to obtain a solution to the challenge of reducing loss in electric power distribution systems - this work is an extension to research conducted by [Cavellucci and Lyra, 1997], where an informed search framework was used to solve the problem.

An electric power distribution system (see Figure 3.32) consists of four important components - root node (entry point of energy into the system, denoted as R), substations (denoted as SS), load blocks (L) and switches (S). For simplicity, assume that electric power is simply "energy" that is dissipated through the network. It has been determined that due to electrical resistance in transmission and distribution lines, energy losses amount to around 7% of total energy production [Bunch et al., 1982]. This number can be minimised through improved energy path distribution, distributing energy within the system equally and without bottlenecks by opening or closing the switches between load blocks. Essentially, the problem was to determine the optimal set of paths for energy to take, whilst considering the network demands, electrical constraints and maintaining radial structure. The project authors hoped that LCS would be suitable for determining rules within the distribution system, and suggest actions based on the system demand load to minimise the loss of energy within the network.

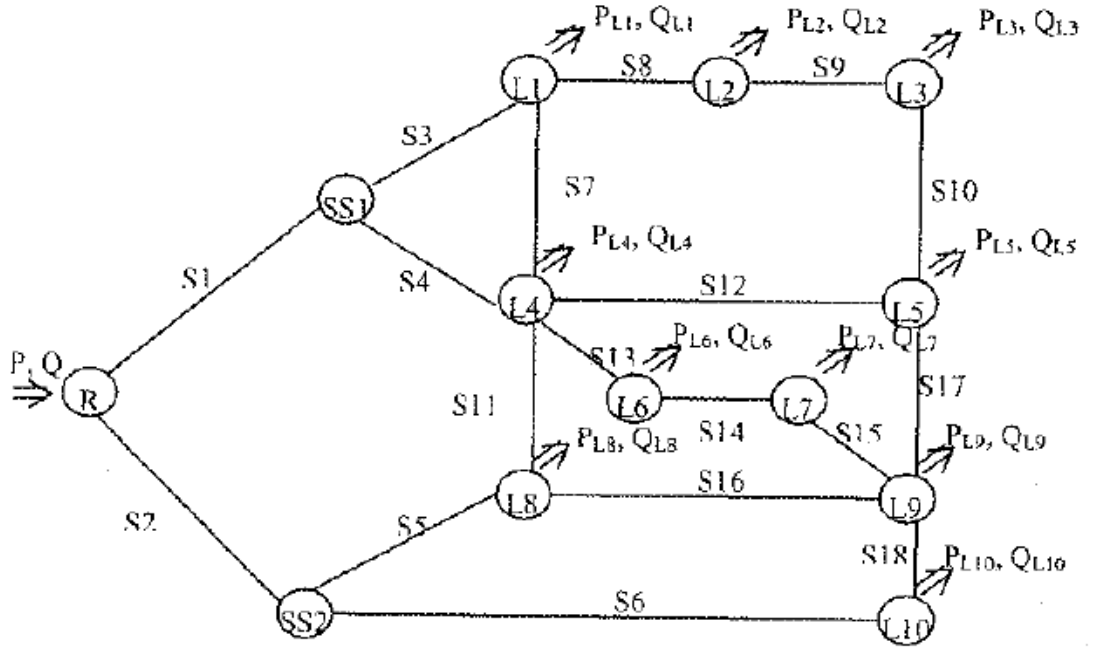


Figure 3.32: Distribution System Graph Model [Vargas et al., 2004].

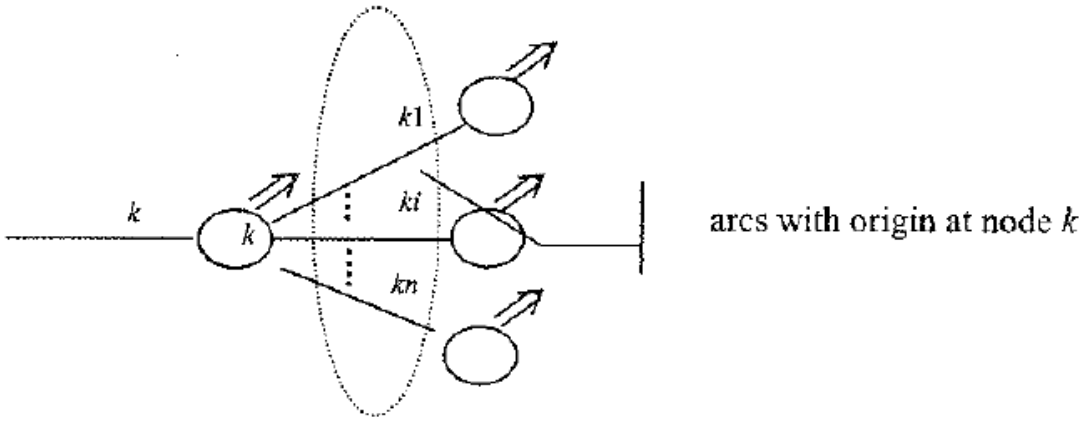


Figure 3.33: Diagram of switches *kl, ki, kn* with origin at node *k* [Vargas et al., 2004].

In order to conduct load balancing activities via switch opening and closing, the state of the power flow at each switch in the system must be known (see Figure 3.33). In order to make it comprehensible for the LCS, the power flow at each switch was encoded into a 2-bit string representation - like this, the complete system state could be encoded via the composition of all 2-bit strings for all switches (number of classifier bits is twice the size of the number of switches within the system). The power flow was then divided into four categories, each category receiving its own "class", or "sector" as is used by the research paper authors (see Figure 3.34 and Table 3.6). As the energy flow through the system changes, so will the string encoding the system state change.

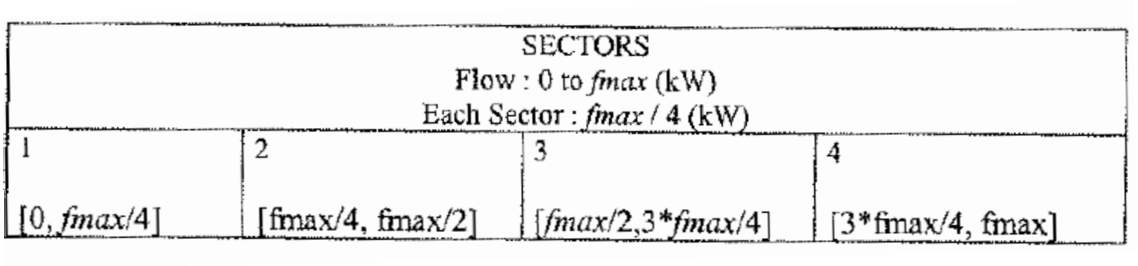
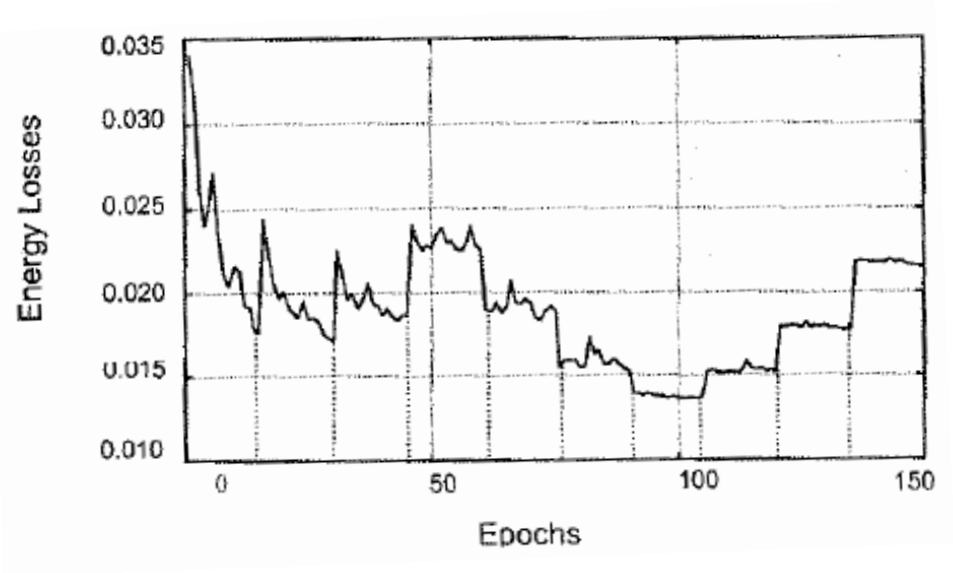


Figure 3.34: Power Flow Codification [Vargas et al., 2004].

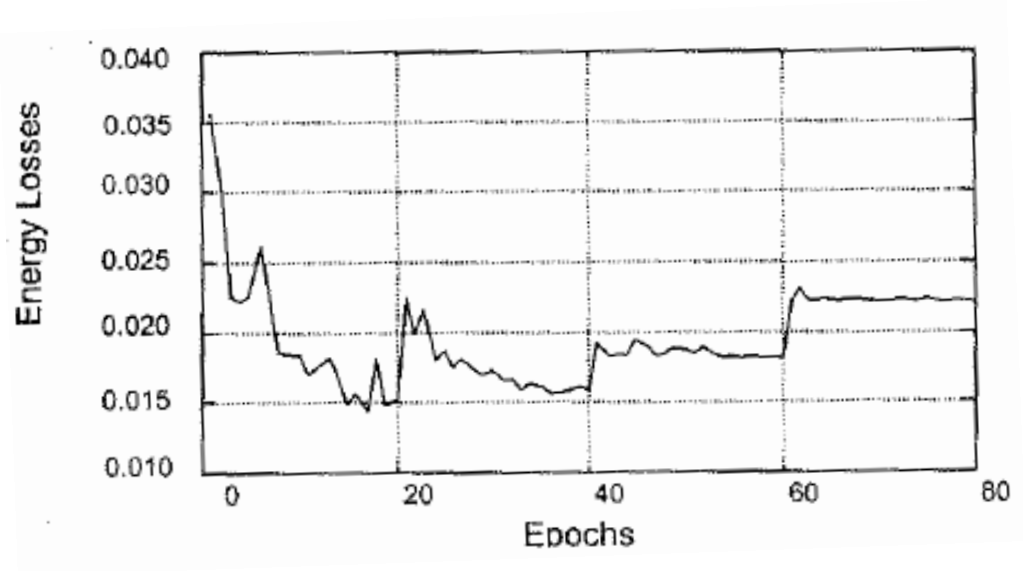
SECTOR	FROM	TO	ENCODING
1	0	784	00
2	785	1569	01
3	1570	2354	10
4	2355	3140	11

Table 3.6: Coding Example ($fmax = 3140$) [Vargas et al., 2004].

As can be seen from the results achieved (Figure 3.35), LCS was successful in solving the problem, reacting quickly to change of system state and suggesting effective actions to be undertaken to minimise system loss - after an initial spike of energy loss following the demand change, it is evident that energy loss is slowly decreased. This is applicable to both when demand changes took place every 15 epochs and every 20 epochs. Additionally, the work conducted confirmed that LCS can be used for online supervision of real distributed networks; unfortunately, even at the time of writing Holland LCS could be considered rather old, and so more research could be conducted to see how well the problem can be solved with more modern LCS variations.



(a) 15 epochs



(b) 20 epochs

Figure 3.35: Average energy loss following demand changes every 15/20 epochs [Vargas et al., 2004].

Chapter 4

LCS Run Parameters Overview

This chapter will provide an in-depth overview of the LCS Run Parameters and present a literature review together with a description of background research on the topic of Run Parameters in Learning Classifier Systems. For an overview of LCS in general, refer to Chapter 3.

4.1 Run Parameter Overview

In Section 3.3.1, the LCS Functional cycle has been described. The majority of the mentioned components can be altered, adapted or "tuned" via the use of run parameters - constants that are set at the beginning of the LCS run and that can have a significant effect on the behaviour of different LCS components and, therefore, on the resulting model. Some examples of parameters include the size of the rule population, the probability of specifying a wild card (#) during the covering stage, number of LCS learning cycle iterations and others. An extensive list of the parameters being investigated in this project is given in the next section in Table 4.1.

Not all LCS run parameters (in future for simplicity, just parameters) are equal in importance - objectively, some parameters are more important than others and can have a greater impact on the operation of the algorithm. First and foremost, it is crucial to set specifically such parameters correctly, as their incorrect values may have a devastating effect on the resulting algorithm model accuracy, making it unfit to be used for prediction.

Setting correct LCS parameters is a challenging task that involves a lot of experimentation and trial & error. Complexity stems from the fact that LCS run parameters are directly related with the type of data that the LCS needs to learn - quality of the data (noise), data size, how much training and testing samples are available, and even whether the data is at all generalisable - all has a direct impact on what the optimal value of the LCS run parameters should be.

Unfortunately, limited literature exists on the topic of what the optimal value of the run parameters should be. Often, researchers provide the values of run parameters they used in their experiments, but do not provide any justification for the choice of parameter values. Even if some insight is provided in regards to run parameter values, the insight is high-level and quite vague - recommendations such

as "must be set low", "should be relatively high" are useful, but do not provide a reference point of what exactly *is* high or low. What is the upper boundary of the parameter? What is the lower boundary? What is the relationship between parameters? Such questions often go unanswered when reading research papers, primarily because researchers do not believe in the necessity of providing further insight into the chosen parameter values. This adds additional complexity to the already complicated task of determining the optimal run parameters for a given problem, especially for someone new to LCS.

Specifically tackling this complexity is the purpose of this project. The aim is to provide optimal values for the most important run parameters for different problem types (see Section 5.3.2 for examples of Benchmark Problems used in the project). If a single optimal value can not be determined (what is usually the case, as there is great dependence on the environment that is being learned), a range of values will be provided, with justification that hopefully can be applicable to determine optimal run parameters for other problems and learning environments.

4.2 List of Parameters Investigated

The parameters that will be investigated in this project can be seen in Table 4.1. They have been suggested for use by [Urbanowicz and Browne, 2017]. Urbanowicz is an acknowledged and renowned expert in the field of LCS with numerous related publications, and therefore using his suggestions as the starting point of investigation is a valid approach. [Urbanowicz and Browne, 2017] does not provide any information regarding the source of the values and why they have been set to what they are, but at this stage this is the best that is available in regards to at least some concrete values and the range possible for each parameter. The table below will be used as the basis of the investigation in this project and can be perceived as the scope of parameter investigation for this project. Just to confirm, the table below is not considered as *absolute truth*, but is perceived solely as the starting point of investigation. The research carried out in this project will be able to confirm whether the table devised by [Urbanowicz and Browne, 2017] is correct, but more crucially will also provide a better understanding of the effect each parameter has on the LCS output and how run parameters are related to each other.

Parameter	**Sym.**	**Initial Value**	**Common Range**	**Increment**	**Changeable**
Env. Iterations	I	10,000	10k-2M	x10	Often
Population size	N	1,000	500-50k	$\pm 1,000$	Often
Don't care probability	$P_{\#}$	0.3	0-0.99	± 0.1	Often
Accuracy threshold	ϵ_0	0.01	0-0.01	± 0.01	Moderately
Fitness exponent	ν	5	1-10	± 1	Moderately
Learning rate	β	0.1	0.1-0.2	± 0.02	Moderately
Mutation probability	μ	0.4	0.2-0.5	± 0.1	Rarely
Crossover probability	χ	0.8	0.7-0.9	± 0.1	Rarely
Fitness fall-off	α	0.1	0.1	NA	Never

Table 4.1: Run parameters for Michigan-style LCS [Urbanowicz and Browne, 2017].

4.3 Run Parameters Literature Review

This section provides background research focusing on LCS run parameters. General overview of LCS has been provided in Chapter 3.

4.3.1 Iteration parameter

The iteration parameter ($\boldsymbol{I}$) specifies the maximum number of learning iterations that the LCS completes before an evaluation of the algorithm generated model commences. Within a single LCS iteration, a single training instance is retrieved from the training data set (environment) and the LCS performs a learning iteration over it.

This parameter is not LCS-specific, in the sense that it is a common parameter for other machine learning algorithms and should be set similarly to how it is done in other algorithms, e.g. neural networks. The general consensus among researchers is that there is *no optimal* value for the number of iterations - rather, one must iterate until the error does not significantly decrease. A lazy approach would be to simply set the number of learning iterations as high as possible, but this approach is not feasible, as it will lead to a great waste of computational resources.

[Urbanowicz and Browne, 2017] stress the importance of correctly setting the iteration parameter - too few iterations, and the LCS will not have sufficient time/attempts to learn the domain, whilst too many, as suggested above, will lead to a waste of computational resources with very little gain. The challenge, therefore, in this project is to potentially narrow down the range of possible value for the iteration parameter, as a value of 2 million suggested in the table above already appears to be quite excessive (after looking through numerous research papers, no paper was identified with such a large value of environment iterations). There is evidence that substantial and complex problems can be solved with a much smaller number of iterations, e.g. [Iqbal et al., 2013] used 50000 environment iterations for their XCS algorithm when solving the 20-bit Multiplexer problem. Evidently, the actual value for the number of learning iterations is dependent on the problem being learned - this project attempts to provide a relationship between the complexity of the problem, and the number of iterations needed to learn it.

4.3.2 Population size parameter

The ***population size parameter*** ($\boldsymbol{N}$) specifies the allowed maximum LCS population size before the deletion mechanism kicks in to delete excessive rules to maintain this specified amount.

[Urbanowicz and Browne, 2017] suggest that the value for the population size must be 10 times the number of expected rules in the model. Given the stochastic nature of LCS, it is very difficult, if not impossible to accurately predict what the resulting number of rules in the final model will be - therefore, this suggestion is of limited value. [Butz and Wilson, 2001] in their XCS algorithmic description paper give the suggestion of setting the population size sufficiently large so that covering occurs only at the beginning of the LCS learning process - again, this is unfortunately

too vague to draw any conclusions regarding the relationship between problem size and LCS population size, as well as to what value, or within what range of values, the population size must be set. Finally, a quite advanced method of setting the population size was proposed by [Butz, 2004] - he developed a complex method of estimating the bounds to what the population size could be set to in consideration to the covering mechanism, how specified the rules in the rule population are, as well as numerous other factors. The resulting method was very complex and has not found implementation in any LCS variants.

Generally, several sources ([Urbanowicz and Browne, 2017], [Karlsen and Moschoyiannis, 2018a], [Butz, 2004]) agree on a similar set of principles when setting the population size run parameter:

Population size must be set in accordance to the problem being learned. More complex problems will require larger populations to capture all relationships.

Too small population size will result in low training performance, too high population size is inefficient and hinders convergence to an optimal solution, although setting the population size higher than required is better than have it be of non-sufficient size to completely learn the problem.

4.3.3 Don't care probability parameter

The ***don't care probability parameter*** (***P#***) specifies the probability of inserting a "wild card" (denoted as #) symbol into a classifier that has been generated during the covering stage.

[Urbanowicz and Browne, 2017] highlights the connection between the P# parameter and the type of problem that is being learnt. If the problem being learnt has no generality, the author suggests setting P# to be set to zero, whilst for problems containing high generality, P# may be set as high as 0.95. This is indirectly supported by [Karlsen and Moschoyiannis, 2018a], who suggest that setting P# too low may increase the number of rules required in the population and therefore may conflict with the population size parameter (author note - lower P# will lead to less # in the classifier, meaning that the classifiers are more specific and less general, therefore more rules are required to describe the problem as generalisation is limited), whilst setting P# too high may result in too many overly general rules in the population that are useless in describing the problem at hand. [Butz and Wilson, 2001] suggest a value of 0.33 within their work, referring to Wilson's earlier experiments ([Wilson, 2000], [Wilson, 1995]) where such value was used. [Urbanowicz and Moore, 2015] suggested a mechanism called rule specificity limit (RSL) to be utilised for situations where determining P# may be too difficult or impossible - RSL limits the number of attributes that may be specified within a given LCS rule, determining this based on the environment prior to learning. Such approach was used, for example, in ExSTraCS 2.0 LCS variation, but this is a relatively recent "invention" and will require further research.

4.3.4 Accuracy threshold parameter

The ***accuracy threshold parameter*** (ϵ_0) identifies the limit under which the classifier error ϵ must be in order to consider a classifier accurate.

[Butz and Wilson, 2001] suggest this parameter to be 1% of the maximum payoff being utilised (e.g. if maximum payoff = 1000, $\epsilon_0 = 10$), but other approaches exist – for example, [Tamee et al., 2007] suggested use of an adaptive threshold parameter, which can change during run-time based on the average error of each rule in the match set and the total number of rules present within the match set. The relationship can be described as:

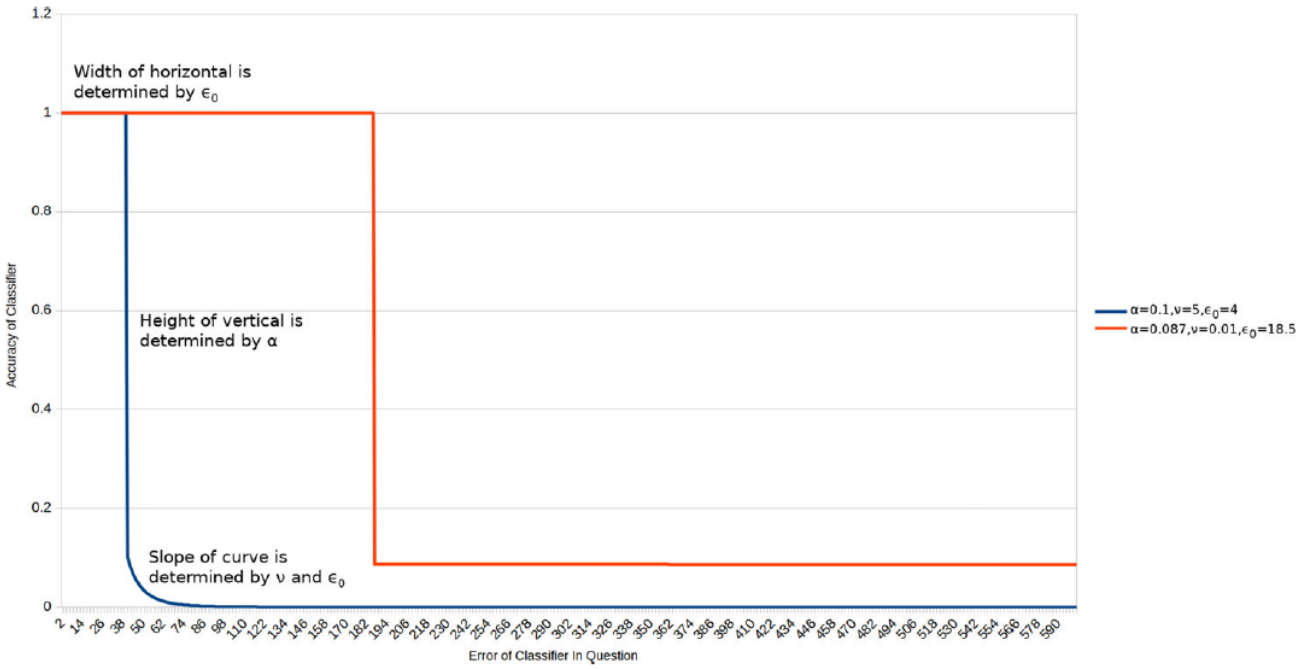
$$\epsilon_0 = \tau(\sum \epsilon_j / N_{[M]}), \tag{4.1}$$

where ϵ_j is the average error of each rule in the match set, $N_{[M]}$ represents the number of rules in the match set, and τ equal 1.2 (constant experimentally found to be most effective).

Values found in experiments differ greatly depending on LCS use case, from $\epsilon_0 = 0.001$ (XCSF is trained on a three-degrees of freedom arm control problem, encoding arm posture [Butz and Herbort, 2008]) to $\epsilon_0 = 18.5$ (Controlling Random Boolean Networks with LCS [Karlsen and Moschoyiannis, 2018a]). It appears that the "cleaner" and simpler the problem, the lower values of ϵ_0 can be used, as in such problems we would not expect the error to be high. For complex and noisy problems, expected error will be higher, so ϵ_0 should be set higher.

4.3.5 Fitness exponent parameter

The fitness exponent parameter (ν) plays an important and direct role in calculating the updated value for the fitness parameter of the classifier. It is utilised differently in XCS and UCS algorithms, and therefore the optimal value will be different in accordance to the algorithm being used. This parameter contributes to how rapidly the accuracy of the algorithm drops to zero after the classifier error exceeds the classifier's error threshold ε_0 - for high values of ν, accuracy drops rapidly, whilst for low values of ν, accuracy decreases slowly. See Figure 4.1 below [Karlsen and Moschoyiannis, 2018a]. Figure 4.1 extends the original chart found in [Butz et al., 2001] (Figure 4.2) - it aims to demonstrate the idea behind accuracy calculation: ϵ_0 is the threshold to what extent errors are accepted; α value affects the distinction between accurate and not quite accurate classifiers; the actual steepness of the slope is influenced by both ν and ϵ_0.



A chart showing the relationship between a classifier's error and its accuracy for the XCS classifier accuracy function. Accuracy is defined as: $\alpha(\epsilon_j/\epsilon_0)^{-\nu}$ where ϵ_j is the error of classifier j. This chart is based on that presented by Butz et al. Fig. 1 (Butz et al. 2001)

Figure 4.1: Relationship between classifier error and accuracy in XCS.

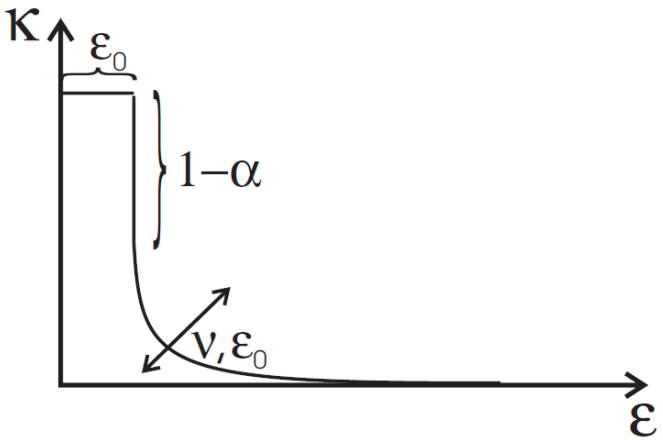


Figure 4.2: Relationship between classifier error and accuracy in XCS.

4.3.6 Learning rate parameter

The ***learning rate parameter*** ($\boldsymbol{\beta}$) denotes the learning rate of the algorithm. The learning rate or step size in machine learning determines to what extent newly acquired information overrides old information [Zulkifli, 2018], and is a crucial parameter in many machine learning algorithms.

In learning classifier systems, setting this parameter is quite simple: [Urbanowicz and Browne, 2017] suggests for complex, inaccurate, noisy problems to use value of 0.1, whilst for clean, complete problems use value of 0.2. [Wilson, 1995] utilises value of 0.2 for his experiments, whilst [Karlsen and Moschoyiannis, 2018b] has used similar value within their own experiment. Since this was a parameter that wasn't deeply investigated in this report, and the data supplied for learning to the LCS utilised within this report's experiments was clean, noiseless and complete, the learning rate was set to 0.2 for all experimental runs.

4.3.7 Mutation probability parameter

The ***mutation probability parameter*** ($\boldsymbol{\mu}$) determines the probability of mutation at every index of a classifier generated by the genetic algorithm. This parameter is not solely present in learning classifier systems and is an important part of any genetic algorithm. Due to its importance, a wealth of research has been conducted to determine the optimum value for the mutation probability. The conclusion – parameter value is tightly dependent on the problem and data that is being learned by the algorithm.

Values of [0.2, 0.5] have been proposed by [Urbanowicz and Browne, 2017], but, quite surprisingly, this range does not correspond to values that the majority of LCS researchers utilised in their experiments. The claim made by [Urbanowicz and Browne, 2017] does *not* find support in literature and in practical research experiments. The range of values that have been used for the mutation probability parameter are within the [0.01, 0.05] range – numerous works support this. On the other hand, some works use much larger values, for example in [Karlsen and Moschoyiannis, 2018a], where LCS was used on a 5-node Boolean Network and $\mu = 0.263$. [Deb, 2011] proposed a formula for calculating the mutation probability given the number of attributes in a rule condition (further in Section 6.6).

Literature agrees that mutations are responsible for the exploration of new solutions – as well as agrees that setting the mutation probability parameter too low could result in preliminary conversion at a local maximum rather than at the optimal solution (in practice this will appear as if new solutions derived from good rules are simply copies of their parents, which hinders and complicates exploration of new areas of the rule space), whilst setting the parameter too high will yield poor results, as the algorithm will have a reduced search ability and will be unable to adjust good solutions – essentially, the mutation process will begin to resemble the covering stage.

4.3.8 Crossover probability parameter

The crossover probability parameter (χ) determines the probability of applying crossover to some selected classifiers during the run of the genetic algorithm. This is not one of the parameters that is often tuned, and if it is then nearly always in unison with another parameter, mutation probability μ. If set to a high value, crossover will take place frequently, what might be desirable in certain problem domains. Setting this value low will result in rules crossing-over less frequently, what will have a direct effect in reducing the likelihood of better rules emerging within the population [Karlsen and Moschoyiannis, 2018a]. Either way, fine-tuning this parameter value should in all cases lead to efficiency improvements in number of iterations required to converge to optimal classification performance and in the utilisation of related computational resources, although this might not be essential if only the algorithm accuracy is important [Urbanowicz and Browne, 2017].

Initially suggested to be of the range [0.5, 1] in [Butz and Wilson, 2001], the range suggested in [Urbanowicz and Browne, 2017] narrows down the appropriate range even further, leaving the suggested values to be from [0.7, 0.9].

4.3.9 Fitness fall-off parameter

The ***fitness fall-off parameter*** (α) determines the severity of accuracy decline if the classifier is inaccurate. The calculation is completed as follows: when a classifier error is greater or equal to the accuracy threshold, the accuracy is determined by the values of the fitness fall-off and the fitness exponent parameter. If α is small, when the classifier error exceeds the accuracy threshold the drop off will be larger, whilst when α approaches 1, the drop off will be small to non-existent, and so the fitness exponent becomes the main determinant of the rate of accuracy decrease as error increases (see Figure 4.1). [Urbanowicz and Browne, 2017] suggests for single-step problems α to be set to 1 - looking through numerous other LCS experiments and other implementations, it is rare to see this value be anything other than 1. Since this parameter was not focused on particularly during this project, a value of 1 as suggested by [Urbanowicz and Browne, 2017] was used.

Chapter 5

LCS Implementation Design & Overview

This chapter will provide an overview of the XCS/UCS implementation that has been utilised in order to gather the data for this project. It will introduce an overview of other existing LCS implementations, provide justification for this projects implementation design decisions, provide an in-depth overview of the project methodology, as well as how data was prepared and generated. This Chapter will continue with an overview of the Python files making up the UCS/XCS implementation, as well as will provide an overview of a single iteration run for both of the aforementioned algorithms. Code snippets of main LCS components will also be provided.

5.1 Overview of Existing LCS Implementations

In order to investigate the effect of run parameters on the LCS resulting model prediction accuracy, computational time and size, an implementation of XCS and UCS had to be utilised. As creating an original implementation was initially not the main goal of the project, an investigation was conducted in order to see whether existing implementations exist that can be used within the project. The results of the investigation determined that no fitting single implementation that fit all of the requirements existed, and therefore an original implementation will need to be created. Further justification can be found in Section 5.2. The table below lists existing LCS implementation that have been considered for use within the project:

Name	Released	Language	Description
Multi-LCS-Variants [Karlsen, 2018]	2018	Java	Implementation of XCS and UCS algorithms for domain knowledge discovery. Does not contain predictive capability, no validation.
xcs 1.0.0 [Hosford, 2015]	2015	Python	Implementation of XCS algorithm and available as a library in pip package manager. Output not easy to interpret, difficult to use, not trivial to convert into UCS.
eLCS [Urbanowicz, 2014]	2014	Python	Implementation of a basic generic LCS algorithm, closely relating to UCS. Easy to use and interpret, output provides a lot of information on population and individual rules. Possible to convert to XCS. Provides ability to see state of individual LCS components such as population, rules in population, match set, etc.
pylcs [Lukins, 2013]	2013	C++/Python	Python interface to C++ implementation of XCS. Faster than other implementations, but non-trivial to change run parameters and non-trivial to see the state of individual LCS components. Non-trivial to convert to UCS.
XCS/UCS [Urbanowicz, 2010]	2010	Python	Python implementation of XCS and UCS used by author in their own research work. Does not contain capability to easily visualize individual LCS components and to easily alter run parameters.
xcslib [Lanzi, 2009]	2009	C++	Open source project providing a C++ framework to perform research with XCS. Non-trivial to convert to UCS.

Table 5.1: Table to show LCS implementations considered for use in the project.

5.2 Justification

Following the research carried out and described in Section 5.1, a conclusion was made that no single existing implementation is sufficient to cover all the needs of the project. It was decided to use **eLCS** [Urbanowicz, 2014] as the basis for the UCS and XCS algorithms: eLCS was already very close to the implementation of UCS, whilst at the same time it was trivial how to extend eLCS to operate like the XCS algorithm. The advantages and disadvantes of utilising the eLCS implementation included:

- Advantages
    1. *Simple implementation* - implementation contains bare minimum to make a functional LCS, not adding unnecessary complexity.
    2. *Easy result interpretability* - implementation allows to see operation of every LCS component in the system, allowing to interpret the actions of the system in detail.
    3. *UCS implementation* - eLCS is very close to the implementation of UCS, and therefore requires minimal changes to be converted into UCS.
    4. *Python implementation* - Python was used as the coding language of choice. Python is a popular language, easy to use, and is very commonly used for machine learning. Considering that the final implementation would be released for open-source use, using Python would target a larger potential audience.
    5. *Setting of run parameters* - easy setting of parameters and transparent how the parameters operate within the system.
- Disadvantages
    1. *No XCS implementation* - no XCS implementation is available, so will need to create one based on eLCS.
    2. *No model validation* - No model validation was implemented, therefore there was no way to evaluate the evolved model.

Specifically due to the aforementioned important advantages and negotiable disadvantages, eLCS was chosen to be the base of the UCS and XCS implementations for this project. Some eLCS improvements included:

- *Iteration cycles* - two paradigms in LCS exist - running the LCS for a set number of iterations (defined as a run parameter), or until the training accuracy becomes 100%. In the latter case, training happens faster and uses a lower number of iterations, but the resulting model is not as accurate and provides no guarantee for the required number of cycles needed. Since the number of LCS iterations required had to be kept constant for a fair test experiment, the LCS iteration strategy was set to iterate exactly the specified amount of times, regardless if the training accuracy reached 100%.

- *K-Fold Cross-Validation* - 5-fold Cross-Validation was added to evaluate the resulting model following parameter changes. More details can be found in 5.3.3.

- *Timer* - timer was added to record LCS algorithm run.

- *Population compression* - three different population compression strategies were added, ranging from "soft" to "extreme" compression. More can be found in Section 5.6.2.

- *XCS* - eLCS algorithm was adjusted to become an XCS implementation of the algorithm. More in Section 5.6.2.

- *Run loop* - the process of gathering results has been automated by creating a loop where each run of the loop corresponds to one entire LCS training run, with results of the model evaluation being printed in correct format to an external file.

- *Benchmarks* - additional four benchmark problems were developed specifically for this project to run with eLCS (UCS) and the adapted XCS.

5.3 Methodology

5.3.1 Experiment Methodology

This section will describe the overall experiment methodology and how results will be gathered, processed and evaluated. Results will be gathered by starting with the "recommended" parameter values suggested by [Urbanowicz and Browne, 2017] and then altering every parameter consecutively, changing value from the smallest recommended value to the highest recommended value whilst keeping other parameters constant. For every experiment, the resulting model cross-validation accuracy, number of rules in population and run time will be recorded. Each experiment will be repeated 10 times to minimise the LCS stochastic effect[1], after which the gathered results will be averaged for the experiments. This will be repeated for every benchmark problem (Section 5.3.2) and the averaged values recorded appropriately in a table.

This is massive amount of data and experiments to be completed, and therefore two machines were used to gather results: Dell XPS 15 (9570)[2] and a Desktop-PC[3]. Of course, since the two systems are different the algorithm run time will be affected and will not be the same for the two systems - it is therefore important to stress that **the results gathered should not be considered absolute values, but rather, the trend demonstrated by the results must be considered and their relation to each other**. The two systems were not used simultaneously on a single benchmark and have been separated in order not to contaminate results. This project does not aim to demonstrate any absolute values, e.g. that 10000 iterations take 49 seconds for a 9-bit Position XCS run; rather, the project aims to demonstrate that if 10000 iterations take 49 seconds, then 20000 will take twice as long, for example.

LCS run parameters are supplied via a configuration file, which specifies the LCS run parameters, other implementation parameters used (e.g. random seed), algorithm heuristic options (e.g. GA or Action set subsumption, crossover method). A separate configuration file exists for UCS and XCS implementations; snippet of a UCS configuration file can be seen on the next page.

[1] A value of 10 repetitions has been chosen, as any less would provide insufficient data and would not allow to minimise the LCS stochastic effect, whilst any more than 10 would be unlikely to yield any different results and would simply take more time with minimal gain.

[2] Intel® Core™ i7-8750H Processor @ 2.20 GHz, 16GB DDR4 2666MHz, Windows 10 Home.

[3] Intel® Core™ i7-7700K Processor @ 4.20 GHz, 16GB DDR4 2133MHz, Windows 10 Home.

```
###### Configuration File (UCS)

######-------------------------------------------------------
###### Major Run Parameters - Essential to be set
###### correctly for a successful run of the algorithm
######-------------------------------------------------------
datasetDirectory=Demo_Datasets  #        Directory of training/testing datasets
trainFile=11Parity_Data_Complete_Randomized.txt     # Name of training dataset
testFile=None                   #        Name of testing dataset
outputDirectory=Local_Output    #        Output file directory
outputFile=ExampleRun           #        Name of output files
learningIterations=10000        # (I)    Iteration parameter
N=1000                          # (N)    Maximum population size parameter
p_spec=0.01                     # (1-P#) Chance of an attribute when covering
kfold=5                         #        K-Fold number - If not used, set to 0

######-------------------------------------------------------
###### Logistical Run Parameters
######-------------------------------------------------------
randomSeed=False                # Set a constant random seed value to some integer
labelphenotypeID=phenotypeID    # Label for the phenotype ID's column header
labelphenotype=Class            # Label for the phenotype label column header
labelMissingData=NA             # Label used for any missing data in the data set
discreteAttributeLimit=100      # Limit before attribute or phenotype is continuous
trackingFrequency=0             # Value 0 = Report progress every epoch

######-------------------------------------------------------
###### Supervised Learning Parameters - Generally just
###### use default values.
######-------------------------------------------------------
nu=5                          # (v) Fitness exponent parameter
chi=0.8                       # (X) Crossover probability parameter
upsilon=0.4                   # (u) Mutation probability parameter
theta_GA=25                   #     GA Threshold
theta_del=20                  #     The deletion experience threshold
theta_sub=20                  #     The subsumption experience threshold
acc_sub=0.99                  #     Subsumption accuracy requirement
beta=0.2                      # (b) Learning rate parameter
delta=0.1                     #     Deletion parameter
init_fit=0.01                 #     The initial fitness for a new classifier
fitnessReduction=0.1          #     Initial fitness reduction in GA offspring rules

######-------------------------------------------------------
###### Algorithm Heuristic Options
######-------------------------------------------------------
doSubsumption=1               # Activate Subsumption? (1 is True, 0 is False)
selectionMethod=tournament    # GA parent selection strategy
theta_sel=0.5                 # Fraction of correct set used in tourn. selection

######-------------------------------------------------------
###### PopulationReboot - An option to begin UCS
###### learning from an existing, saved rule population.
###### Note that the training data is re-shuffled
###### during a reboot.
######-------------------------------------------------------
doPopulationReboot=0                # Start UCS from an existing rule population?
popRebootPath=ExampleRun_UCS_50000  # Path of previously evolved population file
```

Listing 5.1: UCS Configuration File

5.3.2 Benchmark Problems

This section will describe the benchmark problems that have been used for this research project, providing their description, operation and justifying their choice. Inspiration for benchmarks to use was gained from [Orriols-Puig and Bernadó-Mansilla, 2008], who utilised four different benchmarks in their experiments: multiplexer, position, parity and decoder. Advantage of these benchmarks include that they represent fundamentally different problems in terms of data generality and complexity.

Multiplexer

> *"Multiplexer functions have long been identified by researchers as functions that often pose difficulties for paradigms for machine learning, artificial intelligence, neural nets, and classifier systems."*
>
> *- [Koza, 1991]*

The multiplexer, strictly known as the Boolean n-bit Multiplexer, defines a set of supervised learning problems that conceptually describe the behaviour of an electronic multiplexer (MUX), a device that can combine multiple analog or digital input signals and forward them into a single output signal [Dean, 2013]. This is a popular benchmark problem in the LCS community, as it describes a problem that is close and relevant to the complex challenges that LCS might be required to solve in the real world.

A single multiplexer problem instance contains two parts - the condition, made out of individual bits (where each bit represents a distinct input), and the class (phenotype), which corresponds to the multiplexer output. Within the condition, bits can be divided into address bits and register bits - address bits point to one of the register bits and the value of the target register bit determines the instance class. The relationship between address bits and the string length is:

$$l = k + 2^k, \tag{5.1}$$

where l refers to the multiplexer binary string length (condition length), and k refers to the position bits. See Figure 5.1 for an example run for 6-bit Multiplexer class determination.

Multiplexer problems have always been considered difficult for machine learning algorithms, LCS included, primarily due to the complexity of determining the instance class. Multiplexer problems exhibit both epistasis[4] (class is determined via interaction between multiple features, rather than single one) and heterogeneous[5] behaviour (for different instances, the class value is determined by a different subset of features) [Urbanowicz and Browne, 2017] - this elevates the complexity of the problem due to the non-linear association between the condition and the corresponding action. Consider a situation where only a single bit in the instance component is

[4] *Bio.* A type of gene interaction, in which any single gene is under the influence of other, distinct, genes.

[5] consisting of different or diverse components.

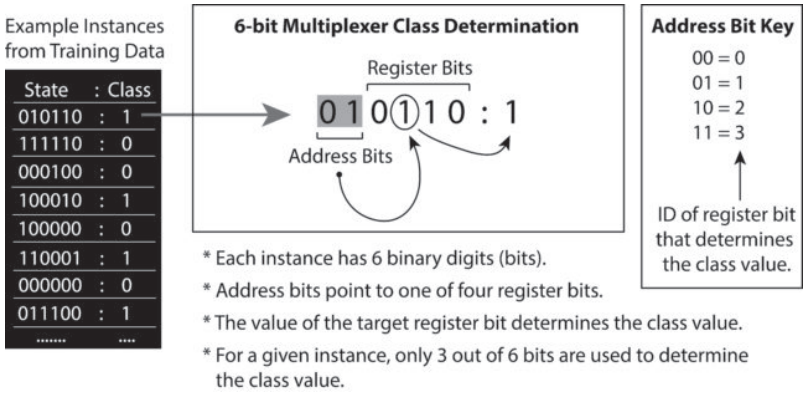


Figure 5.1: 6-bit Multiplexer class determination - example run [Urbanowicz, 2015]

known - in a multiplexer problem, this information will be insufficient to determine the class, as information regarding the state of both the address bits is required; this would be a complex problem for most machine learning algorithms, but is appropriate for a learning classifier system, as it can accommodate for both the epistasis and heterogeneity of the problem.

Epistasis and heterogeneity are closely related to common features of problems that can be encountered by LCS when solving real life challenges - therefore, the multiplexer problem became a staple benchmark for LCS research. Another advantage of Multiplexer problems is that their difficult can be scaled upwards and downwards very easily - by simply increasing or decreasing the number of address bits. The recommended choice for the range is for the lowest value of address bits to be 2 (for a 6-bit Multiplexer), the highest value to be 7 (135-bit Multiplexer, higher has not been solved). Any value in that range may be used as a benchmark for the complexity of the problem being solved, albeit higher address bit values will require extreme computational capabilities.

Finally, the commodity of utilising the Multiplexer problem in LCS experiments can be attributed to the fact that it contains knowledge-based patterns, making it perfect and applicable for LCS - if there is no knowledge-based patterns in the data, LCS will not learn and is therefore not applicable for solving the problem [Urbanowicz and Browne, 2017]. This is an especially important, considering the aims of this project is to evaluate the performance of the LCS from a predictive machine angle - for this to be successful, presence of distinct knowledge-based patterns within data is crucial. See Figure 5.2 and Figure 5.3 for illustration of knowledge-based patterns in a 6-bit Multiplexer.
Due to all the problem characteristics described above, the multiplexer benchmark problem was chosen for use in this project as an example of a balanced, generalisable problem. Specifically for this project, the 11-bit Multiplexer was used - this yielded a sufficient amount of training and testing data for research, yet not incurring a large overhead on the system.

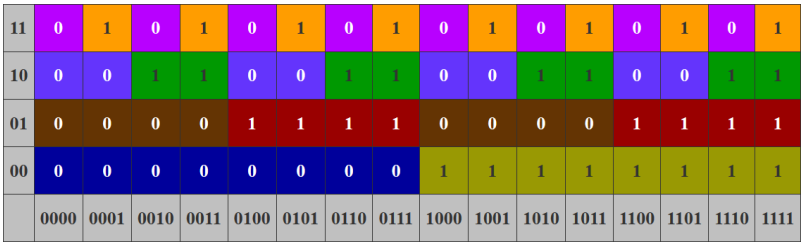


Figure 5.2: 6-bit Multiplexer patterns - vertical axis corresponds to address bits; horizontal - register bits; cells - class value. Each region (shade) represents a condition with associated class that covers a pattern within the domain, e.g. 000### : 0 and 001### : 1. As can be seen, the entire domain can be separated into 8 maximally general rules - this is the theoretical lower limit (optimal) number of rules that the LCS must generate to completely cover the problem.

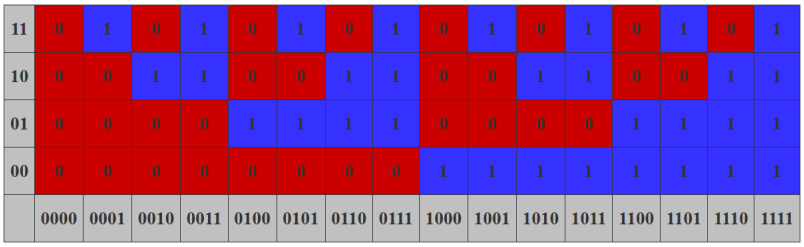


Figure 5.3: 6-bit Multiplexer patterns - domain class symmetry, property of a balanced problem.

Position

The position problem benchmark is an example of an unbalanced multi-class problem, where, given a binary string of length l as input, the output is determined by the value of the index position of the left-most one-valued bit.

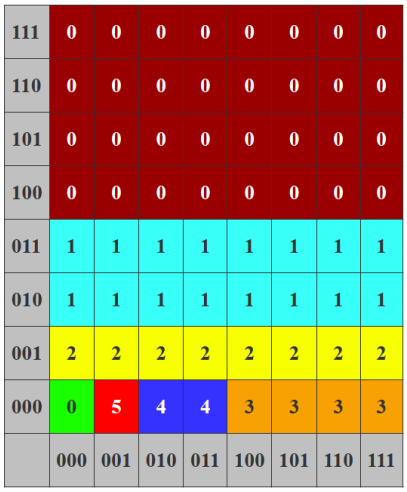


Figure 5.4: 6-bit Position problem - example of an unbalanced generalisable problem.

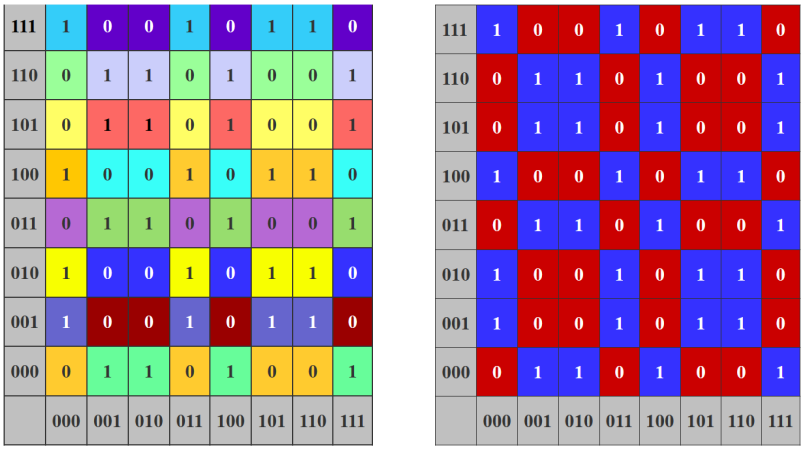
Although being a generalisable problem, the position benchmark differs from other benchmarks as it is unbalanced - this is clearly seen in Figure 5.4, where there are much more values for class 0 than for class 2, for example. Specifically for this property, the 9-bit Position benchmark was utilised within this project - it was interesting to see how a problem with in-built class imbalances would react to different LCS parameters.

Parity

The parity problem benchmark has been a popular benchmark used with LCS since its introduction in [Kovacs and Kerber, 2001] in order to demonstrate the relationship between the population size and XCS performance. The benchmark accepts a binary string as input and returns as output the number of 1's in the input string modulo two:

$$\text{output} = (\text{number of 1's in input string}) \bmod 2 \tag{5.2}$$

This benchmark is different to the multiplexer and position benchmarks, primarily because the parity problem **can not be generalised**, since every bit in the condition plays an important role in determining the instance class. Therefore, the prediction accuracy for this benchmark will never exceed 50%.



(a) Parity domain classes (b) Parity class balance

Figure 5.5: Overview of the Parity problem benchmark.

For this project, the 11-bit parity benchmark was chosen as contrast to the multiplexer and position benchmarks, which are highly generalisable problems. It was interesting to see whether the effect of specifying different run parameters will be different for problems of varying generality.

(Decoder)

The decoder problem benchmark was initially considered for use in this project, but this decision was later reverted and the benchmark was not used in gaining results described in Chapter 6. Nonetheless, it is worthy of mention as an example of a problem that can not be easily predicted with the use of LCS - furthermore, the decoder problem can be considered entirely inappropriate to be learned using LCS.

The decoder problem is a multiclass problem where given a binary string of length l, the output corresponds to the decimal value of binary string l. Such problem will contain no generality at all - this breaches one of the main requirements for a problem to be suitable to be learned with LCS - to be generalisable. As there is absolutely no generality in this benchmark problem, when attempting to use this benchmark for prediction, the prediction accuracy was 0%. This result is unsurprising and is quite logical if looking at Figure 5.6 below:

111	56	57	58	59	60	61	62	63
110	48	49	50	51	52	53	54	55
101	40	41	42	43	44	45	46	47
100	32	33	34	35	36	37	38	39
011	24	25	26	27	28	29	30	31
010	16	17	18	19	20	21	22	23
001	8	9	10	11	12	13	14	15
000	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111

Figure 5.6: 6-bit Decoder problem - unbalanced, non-generalisable problem - unsuitable for LCS.

Therefore, although the LCS used for this project are able to work with this benchmark problem and an appropriate data generation script for it was created, this benchmark was not used to gather data described in Chapter 6.

5.3.3 K-Fold Cross-Validation

Cross-validation is a validation method that allows to evaluate a model using a set of training data. Such evaluation is done in situations where a need arises to evaluate a model and determine its capabilities, but no testing set is available. Several cross-validation methods exist and each cross-validation method has it's own advantages and disadvantages, but considering the amount of data being used for this project, it was decided to use **k-fold cross-validation** with $k = 5$, as this would yield the best balance between the training set and the testing set ($k = 10$ would result in the testing set being too small).

K-Fold cross-validation separates the training data set into k distinct equal sections, also known as "folds". Each block in turn is considered as a testing set (also known as "validation set"), whilst the rest $k - 1$ blocks become the training set. The model trains by iterating through $k - 1$ blocks and tests the resulting model on the validation set block, where some metric is measured, e.g. accuracy, standard deviation, etc. The process is repeated k times, after which the mean of all model evaluations is calculated to determine the final model evaluation.

To summarize, K-fold cross-validation can be achieved in the following steps:

1. Shuffle randomly initial data set.
2. Split data set into k folds.
3. For each fold:
   (a) Set first fold as the testing data set.
   (b) Set remaining folds as training data set.
   (c) Use training set to evolve model and use the model to evaluate the testing data set.
   (d) Repeat k times.
4. Calculate average of model evaluations for k testing data set evaluations.

An important consideration is the value of k to be used. The most common values are $k = 5$ and $k = 10$ [Kuhn and Johnson, 2016], as they have shown "empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance" [James et al., 2015]. Other values for k are also possible. For this project, $\boldsymbol{k = 5}$ was chosen, as it allowed to generate sufficiently large training and testing data sets, representative of the data being learned.

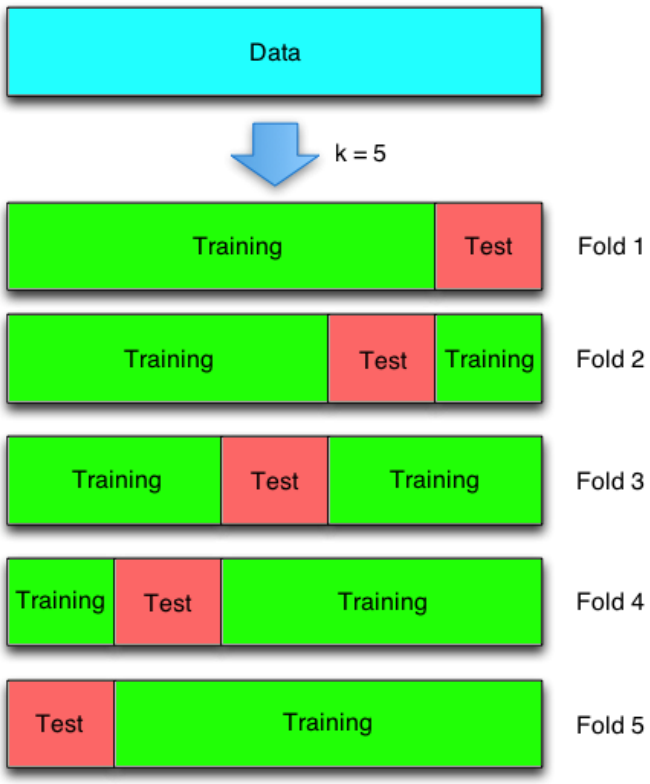


Figure 5.7: Demonstration of original data set split into training and testing data sets for $k = 5$.

5.4 Data Preparation and Overview

Data was prepared with the use of Python scripts, developed from scratch for this project. Every benchmark had a corresponding data generation script, allowing to generate either a single data instance for the given problem, several data instances, or an entire complete data set. The resulting data set was then randomised via a simple randomiser function. The scripts (see Appendix A) allow to generate data instances for problems of various complexity, with the condition bit length being set at the beginning of the script. For problems were the bit-length is important, e.g. multiplexer problem, a sanity checking function exists (see line 51). The resulting data set is written to a file and then given into the LCS as a training set file, where the data is partitioned into k-folds and the algorithm run commences. Example of generating a complete 11-bit Parity data set:

```
#set condition length
bits = 11
def generate_complete_parity_data(myfile, bits):
    """ Attempts to generate a complete non-redundant parity dataset."""
    num_bits = sanity_check(bits)
    if num_bits is None:
        print 'Problem_Parity: ERROR - Specified binary string bits is smaller than
     or 0'
    else:
        try:
            fp = open('Demo_Datasets/' + myfile, 'w')

            # Make File Header
            for i in range(num_bits):
                # Bits
                fp.write('B_' + str(i) + '\t')
            # Class
            fp.write('Class' + '\n')

            # Generating all range of bits
            for i in range(2 ** num_bits):
                binary_str = bin(i)
                string_array = binary_str.split('b')
                binary = string_array[1]

                while len(binary) < num_bits:
                    binary = '0' + binary

                # Starting counter of "1" bits
                one_count = 0
                for j in binary:
                    if int(j) == 1:
                        one_count += 1

                # Determining instance class
                if one_count % 2 == 0:
                    output = 0
                else:
                    output = 1

                for j in range(num_bits):
                    fp.write(binary[j] + '\t')

                # Write output to file
                fp.write(str(output) + '\n')
            fp.close()
            print 'Problem_Parity: Dataset Generation Complete'
        except:
            print 'ERROR - Cannot generate data for binary problem'

#sanity check condition length input
def sanity_check(bits):
    if bits > 0:
        return bits
    return None

#randomize generated data set
def randomize():
    with open("../XCS/Demo_Datasets/"+str(bits)+"Parity_Data_Complete.txt", 'r') as
     source:
        data = [(random.random(), line) for line in source]
    data[1:] = sorted(data[1:])
    with open("../XCS/Demo_Datasets/"+str(bits)+"Parity_Data_Complete_Randomized.
    txt", 'w') as target:
        for _, line in data:
            target.write(line)

if __name__ == '__main__':
    generate_complete_parity_data(str(bits)+"Parity_Data_Complete.txt", bits)
    randomize()
```

Listing 5.2: Code snippet to generate a complete 11-bit Parity data set

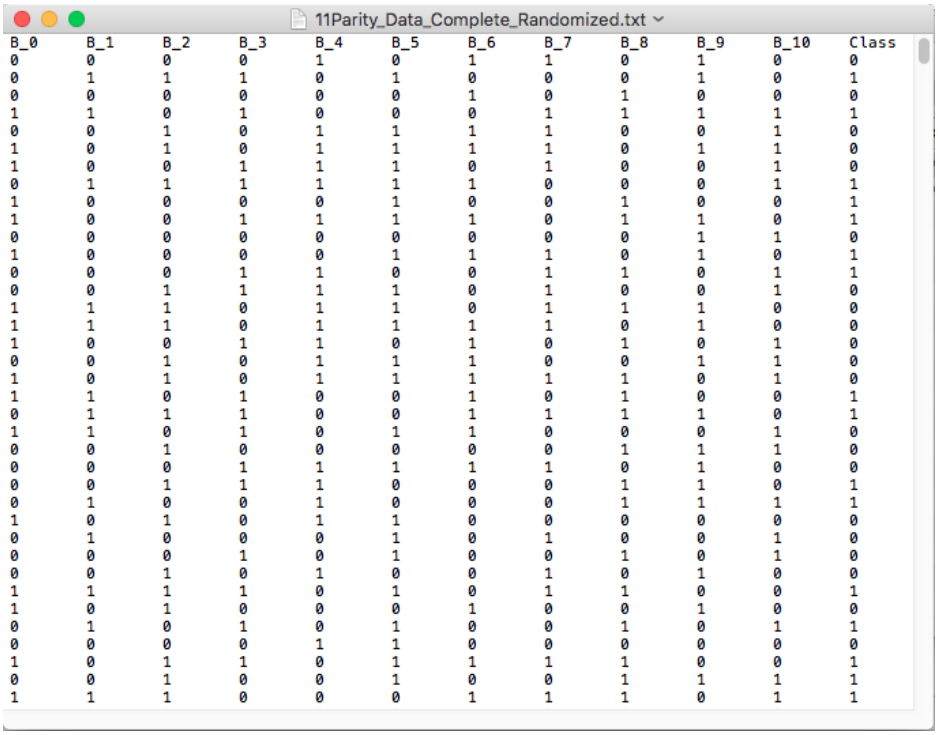


Figure 5.8: Snippet of a complete 11-bit Parity data set.

After the training data set file has been generated, the data must be split into training and testing sets in accordance to the set amount of k-folds. The following code snippets will demonstrate the split of training data into k-folds and setting the folds as training and testing sets (Note: prior to this, the data had to be formatted into a format convenient for the algorithm to interact with. Each instance was stored in a list in the form [Condition, Class, InstanceID]. This will vary from implementation to implementation):

```
def splitDataIntoKSets(self):
    """ divide data set into kfold sets. """
    data_size = len(self.trainFormatted)
    self.folds = [[] for _ in range(cons.kfold)]
    for fold_id in range(cons.kfold):
        fold_size = int(data_size / cons.kfold)
        if fold_id < data_size % cons.kfold:
            fold_size += 1
            offset = fold_id
        else:
            offset = data_size % cons.kfold
        first = fold_id * (int(data_size / cons.kfold)) + offset
        self.folds[fold_id] = self.trainFormatted[first: (first + fold_size)]

def selectTrainTestSets(self, fold_id):
    """ select subset for testing and the rest for training. """
    self.trainFormatted = []
    for i in range(len(self.folds)):
        if i != fold_id:
            self.trainFormatted += self.folds[i]
    self.formatted_test_data = self.folds[fold_id]
    self.numTrainphenotypes = len(self.trainFormatted)
    self.numTestphenotypes = len(self.formatted_test_data)
    print("DataManagement: Number of Train Instances = " + str(self.
    numTrainphenotypes))
    print("DataManagement: Number of Test Instances = " + str(self.
    numTestphenotypes))
```

Listing 5.3: Splitting data into K-Folds and setting training/testing sets

```
Environment: Formatting Data...
DataManagement: Loading Data... Demo_Datasets\11Parity_Data_Complete_Randomized.txt
DataManagement: Number of Attributes = 11
DataManagement: Number of phenotypes = 2048
DataManagement: phenotype Detected as Discrete.
DataManagement: Following Classes Detected:['0', '1']
Class: 0 count = 1024
Class: 1 count = 1024
DataManagement: Detecting Attributes...
DataManagement: Identified 11 discrete and 0 continuous attributes.
...
DataManagement: Number of Train Instances = 1638
DataManagement: Number of Test Instances = 410
```

Figure 5.9: Example output when training with 11-bit Parity set.

5.5 Implementation Files Overview

This section will look into the Python files making up the implementation and provide a description of the functionalities that each Python file implements. The files for UCS and XCS implementations have the same purpose.

Problem_Multiplexer.py

File to generate training data for the Multiplexer benchmark. Provides functionality to generate either a single training instance, non-redundant several training instances or a complete non-redundant randomised Multiplexer data set. Limited computationally to 20 bits, as any number larger takes extremely long to be computed and processed.

Problem_Parity.py

File to generate training data for the Parity benchmark. Provides functionality to generate either a single training instance, non-redundant several training instances or a complete non-redundant randomised Parity data set.

Problem_Position.py

File to generate training data for the Position benchmark. Provides functionality to generate either a single training instance, non-redundant several training instances or a complete non-redundant randomised Position data set.

Problem_Decoder.py

File to generate training data for the Decoder benchmark. Provides functionality to generate either a single training instance, non-redundant several training instances or a complete non-redundant randomised Decoder data set.

UCS/XCS_Algorithm.py

This is the major controlling module of the LCS. This file includes the major run loop which controls learning over a specified number of iterations, as well as includes tracking of estimated performance and checkpoints where complete evaluations of the LCS rule population are performed.

UCS/XCS_ClassAccuracy.py

This file allows to initialise the accuracy calculation for a single class, as well as build a confusion matrix. Manages the logistical aspects of balance accuracy calculations which can handle unbalanced datasets, and/or datasets with multiple discrete classes [Urbanowicz, 2014].

UCS/XCS_ClassifierSet.py

This file handles all classifier sets (population, match set, correct set) along with mechanisms and heuristics that act on these sets, e.g. creating sets sets, initiating covering, subsumption, adding and deleting classifiers from sets, population evaluation, etc.

UCS/XCS_ConfigParser.py

This file interacts with the configuration file - by loading it, parsing its values and passing aforementioned values to the 'Constants' module. Effectively a mechanism to set algorithm parameters to values set in the configuration file.

UCS/XCS_Configuration_File.txt

This is the configuration file which defines the essential run parameters and other settings for the algorithm to use. Recall Listing 5.1 for a snippet of the XCS configuration file.

UCS/XCS_Constants.py

Stores and manages all algorithm run parameters, making them accessible anywhere in the rest of the algorithm code via import.

UCS/XCS_DataManagement.py

This file provides capability to manage both training and testing data. This module loads the data set, detects and characterises all attributes in the data set, handles missing data, and formats the data so that it may be conveniently utilised by the LCS. Data formatting includes detecting the number of attributes and classes, determining if class and/or attributes are continuous or discrete, as well as extracting data from the training file.

UCS/XCS_Offline_Environment.py

This module loads the data set, automatically detects features of the data. This file is specifically created for offline problems - in the context of data mining and classification tasks, the environment is perceived as a data set with a limited number of classes with X attributes and some class of interest.

UCS/XCS_OutputFileManager.py

This module contains the methods for generating the different output files generated by the LCS. These files are generated at each learning checkpoint, and the last iteration. The files are then stored in the *Local_Output* folder within the project structure. Two types of files are generated by this module - firstly, this module generates an output text file which includes all of the evaluation statistics for a complete analysis of all training and testing data on the current LCS rule population (Generated file name is in the format *ExampleRun_LCS_NumberOfIterations_PopStats.txt*), and secondly, this module writes a tab delimited text file outputting the entire evolved rule population, including conditions, classes, and all rule parameters (Generated file name is in the format *ExampleRun_LCS_NumberOfIterations_RulePop.txt*).

UCS/XCS_Prediction.py

Given a match set, this module uses a voting scheme to select the phenotype prediction. Set up to handle both discrete and continuous classes (continuous not used in this project), as well as also capable of handling prediction ties if such occur.

UCS/XCS_Run.py

To run the LCS, this module needs to be run - it contains the *main method* and acts essentially as the implementations *main class*. A properly formatted configuration file, with all run parameters specified, must be provided for the program to launch.

UCS/XCS_Timer.py

Tracks and stores the run time of algorithm and some of it's major components.

5.6 Implementation Overview

This section will provide step-by-step execution of a single XCS and UCS algorithm iteration. The code snippets have been shortened in order to demonstrate only the crucial bits and not distract with helper methods - this is denoted by "..." symbol. The snippets have been commented in order to make their understanding as easy as possible. Full method implementations can be easily accessed in the submitted source code. For differences between UCS and XCS, recall Section 3.5 .

5.6.1 UCS Implementation Run

The UCS algorithm can be launched from the **UCS_Run.py** file main method. Within the main method, it is possible to specify how many time we want our UCS algorithm to be re-run after completion - this allows to automate the data gathering process by having the algorithm re-run itself a specified amount of times (in the case of this project, 10 times) and gather the required data for analysis.

```
1 if __name__ == '__main__':
2
3     # Run code 10 times
4     for i in range(10):
5         mainRun()
```

Listing 5.4: Main method

At a high-level, a single run of the UCS implementation main method encompasses the following steps (an in-depth insight into each step will be provided lower):

1. Specify the configuration file name and file path.
2. Obtain all run parameters from the configuration file and store them in the *Constants* module.
3. Initialise the *Timer* module which tracks the algorithm run time.
4. Initialise the *Environment* module which manages the data presented to the algorithm.
5. Clear output folder before Run.
6. Prepare data, begin training and testing:
   (a) If using K-Fold Validation:
      i. Get the data into a format convenient for the algorithm to use.
      ii. Split data into training and testing k-fold sets.
      iii. For the number of K-Folds:
         A. Select subset for testing and the rest for training.
         B. Identify maximum number of learning iterations and evaluation checkpoints.
         C. Run the UCS algorithm.
         D. Record average k-Fold accuracy.
7. Print average k-Fold accuracy and stop timer.
8. Print results to *RESULTS_FILE.txt*, recording accuracy, algorithm run time and total population size.

Specify configuration file

Specifying the configuration file is as simple as simply specifying the path of the configuration file location:

```
def mainRun():
    # Specify the name and file path for the configuration file.
    configurationFile = "UCS_Configuration_File.txt"
```

Listing 5.5: Specifying a configuration file

Obtain run parameters from configuration file and store them in the *Constants* module

In order to achieve this task, we must launch the ConfigParser file with the input being the configuration file that we want to parse:

```
    ...
    ConfigParser(configurationFile)
```

Listing 5.6: Launching the Configuration Parser

The ConfigParser parses the supplied configuration file, finding and retrieving values for parameters specified within the configuration file. The result is a list of parameters that is then fed into a method that sets their values to the corresponding parameters stored in the Constants module. This allows to use the supplied parameters in the configuration file within the program.

```
class ConfigParser:
    def __init__(self, filename):
        self.commentChar = '#'
        self.paramChar = '='
        # Parse the config file and get all parameters.
        self.parameters = self.parseConfig(filename)
        # Store run parameters in the 'Constants' module.
        cons.setConstants(self.parameters)

    def parseConfig(self, filename):
        """ Parses the configuration file. """
        parameters = {}
        try:
            f = open(filename)
        except Exception as inst:
            ...
        else:
            for line in f:
                # Remove text after comment character.
                if self.commentChar in line:
                    # Split on comment character, only keep text before character.
                    line, comment = line.split(self.commentChar,
                                               1)

                # Find lines with parameters (param=something)
                if self.paramChar in line:
                    # Split on parameter character
                    parameter, value = line.split(self.paramChar, 1)
                    # Strip spaces
                    parameter = parameter.strip()
                    value = value.strip()
                    # Store parameters in a dictionary
                    parameters[parameter] = value

            f.close()

        return parameters
```

Listing 5.7: Parsing configuration file

```
class Constants:
    def setConstants(self, par):
        ...
        # Major Run Parameters ————————————————————————————————————
        ...
        self.kfold = int(par['kfold'])
        ...
        self.learningIterations = par['learningIterations']  # Saved as text
        self.N = int(par['N'])                               # Saved as integer
        self.p_spec = float(par['p_spec'])                   # Saved as float

        # Logistical Run Parameters ———————————————————————————————————
        if par['randomSeed'] == 'False' or par['randomSeed'] == 'false':
            self.useSeed = False                             # Saved as Boolean
        else:
            self.useSeed = True                              # Saved as Boolean
            self.randomSeed = int(par['randomSeed'])         # Saved as integer
```

Listing 5.8: Code snippet of setting parameters in Constants file

Initialise the *Timer* module to track the algorithm run time

This is required in order to be able to record the time it takes the algorithm to run. This will allow us to see the effect of different run parameters values on the speed of algorithm execution.

```
    timer = Timer()
    cons.referenceTimer(timer)
```

Listing 5.9: Initialising timer and creating a reference to timer object

Initialise the *Environment* module

This module loads the data set and automatically detects features of the data, as well as allows to retrieve data from the data set. This results in an understanding of the context of the data being solved, e.g. whether the data is discrete or continuous, number of instances and classes, etc.

```
    env = Offline_Environment()
    cons.referenceEnv(env)
```

Listing 5.10: Initialising the Environment module

```
class Offline_Environment:
    def __init__(self):
        # Initialize global variables————————————————————————————————————
        ...
        self.formatData = DataManagement(cons.trainFile, cons.testFile)

        # Initialize the first dataset phenotype to be passed to LCS
        self.currentTrainState = self.formatData.trainFormatted[self.dataRef][0]
        self.currentTrainphenotype=self.formatData.trainFormatted[self.dataRef][1]
        ...

    def getTrainphenotype(self):
        """ Returns the current training phenotype. """
        return [self.currentTrainState, self.currentTrainphenotype]

    def getTestphenotype(self):
        """ Returns the current testing phenotype. """
        return [self.currentTestState, self.currentTestphenotype]
    ...
```

Listing 5.11: Code snippet of the Offline_Environment class

Clear output folder before Run

Clear output folder before every run. This allows to avoid problems that may occur when trying to overwrite an existing file, as well as will not cause confusion that could potentially arise if files for different algorithm runs were kept in the same place.

```
    # Clear Local_Output Folder before Run
    folder = 'Local_Output'
    for file1 in os.listdir(folder):
        file_path = os.path.join(folder, file1)
        try:
            if os.path.isfile(file_path):
                os.unlink(file_path)
        except Exception as e:
            print(e)
    ...
```

Listing 5.12: Clearing output folder

Preparing the Data

Before the data can be used with the LCS, it must be correctly extracted and formatted from the data set file. Specifically, each extracted training instance is stored in a list as follows: [Condition, Class, InstanceID]. The data is split into the corresponding number of training and testing folds, as specified by the *kfold* value in the configuration file. Finally, from the split folds the training and testing sets are selected and the number of iterations for which the LCS algorithm must be run is retrieved from the Constants module (originally specified in the configuration file).

```
        total_instances = env.formatData.numTrainphenotypes
        env.formatData.splitDataIntoKSets()
        ...
        for i in range(cons.kfold):
            ...
            env.formatData.selectTrainTestSets(i)
            # Retrieve number of learning iterations
            cons.parseIterations()
```

Listing 5.13: Preparing the data

```
    def formatData(self, raw_data):
        formatted = []
        # Initialize data format------------------------------------------------
        for _ in range(len(raw_data)):
            # [Attribute States, Phenotype, InstanceID]
            formatted.append([None, None, None])

        for inst in range(len(raw_data)):
            state_list = [0] * self.numAttributes
            attributeID = 0
            for att in range(len(raw_data[0])):
                # Get attribute columns
                if att != self.phenotypeIDRef and att != self.phenotypeRef:
                    target = raw_data[inst][att]
                    # If the attribute is discrete
                    if target == cons.labelMissingData:
                        state_list[attributeID] = target
                    else:
                        state_list[attributeID] = int(target)
                    attributeID += 1

            # Attribute states stored here
            formatted[inst][0] = state_list
            if self.discretephenotype:
                # phenotype stored here
                formatted[inst][1] = int(raw_data[inst][self.phenotypeRef])
            else:
                formatted[inst][1] = float(raw_data[inst][self.phenotypeRef])
            if self.arephenotypeIDs:
                # Instance ID stored here
                formatted[inst][2] = int(raw_data[inst][self.phenotypeIDRef])
            ...
        return formatted
```

Listing 5.14: Formatting data to be used by the algorithm

```
    def splitDataIntoKSets(self):
        """ divide data set into kfold sets. """
        data_size = len(self.trainFormatted)
        self.folds = [[] for _ in range(cons.kfold)]
        for fold_id in range(cons.kfold):
            fold_size = int(data_size / cons.kfold)
            if fold_id < data_size % cons.kfold:
                fold_size += 1
                offset = fold_id
            else:
                offset = data_size % cons.kfold
            first = fold_id * (int(data_size / cons.kfold)) + offset
            self.folds[fold_id] = self.trainFormatted[first: (first + fold_size)]
```

Listing 5.15: Split data into folds

```
    def selectTrainTestSets(self, fold_id):
        """ select subset for testing and the rest for training. """
        self.trainFormatted = []
        for i in range(len(self.folds)):
            if i != fold_id:
                self.trainFormatted += self.folds[i]
        self.testFormatted = self.folds[fold_id]
        self.numTrainphenotypes = len(self.trainFormatted)
        self.numTestphenotypes = len(self.testFormatted)
        print("DataManagement: Number of Train Instances = " + str(self.
    numTrainphenotypes))
        print("DataManagement: Number of Test Instances = " + str(self.
    numTestphenotypes))
```

Listing 5.16: Selecting data subsets for testing and training

Running the UCS Algorithm

The code below will provide the step-through for a single UCS learning iteration. Prior to this, all required global values have been initialised, and the population instantiated. Within the major running loop, a training instance is retrieved and a learning iteration is performed upon it (recall Figure 3.3 and Section 3.3.1).

```
    def runIteration(self, phenotype, exploreIter):
        """ Run a single LCS learning iteration. """
        # -------------------------------------------------------
        # FORM A MATCH SET - includes covering (STEP 3 - FIGURE 3.3)
        # -------------------------------------------------------
        self.population.makeMatchSet(phenotype, exploreIter)
        # -------------------------------------------------------
        # MAKE A PREDICTION - utilised here for tracking learning progress.
        # Typically used in the explore phase of many LCS algorithms.
        # -------------------------------------------------------
        cons.timer.startTimeEvaluation()
        prediction = Prediction(self.population)
        phenotypePrediction = prediction.getDecision()
        # -------------------------------------------------------
        # PREDICTION NOT POSSIBLE
        # -------------------------------------------------------
        if phenotypePrediction is None or phenotypePrediction == 'Tie':
            ...
        else:  # Prediction Successful
            # -------------------------------------------------------
            # DISCRETE phenotype PREDICTION
            # -------------------------------------------------------
            if cons.env.formatData.discretephenotype:
                if phenotypePrediction == phenotype[1]:
                    self.correct[exploreIter % cons.trackingFrequency] = 1
                else:
                    self.correct[exploreIter % cons.trackingFrequency] = 0
            ...
                ...
        cons.timer.stopTimeEvaluation()
        # -------------------------------------------------------
        # FORM A CORRECT SET (STEP 4 - FIGURE 3.3)
        # -------------------------------------------------------
        self.population.makeCorrectSet(phenotype[1])
        # -------------------------------------------------------
        # UPDATE PARAMETERS (STEP 6 - FIGURE 3.3)
        # -------------------------------------------------------
        self.population.updateParams(exploreIter)
        # -------------------------------------------------------
        # SUBSUMPTION - APPLIED TO CORRECT SET - A heuristic for adding
        # additional generalisation pressure to LCS
        # (STEP 7 - FIGURE 3.3)
        # -------------------------------------------------------
        if cons.doSubsumption:
            cons.timer.startTimeSubsumption()
            self.population.doCorrectSetSubsumption()
            cons.timer.stopTimeSubsumption()
        # -------------------------------------------------------
        # RUN THE GENETIC ALGORITHM - Discover new offspring rules
        # from a selected pair of parents
        # (STEP 8 - FIGURE 3.3)
        # -------------------------------------------------------
        self.population.runGA(exploreIter, phenotype[0], phenotype[1])
        # -------------------------------------------------------
        # SELECT RULES FOR DELETION - This is done whenever there are
        # more rules in the population than 'N', the maximum population size.
        # (STEP 9 - FIGURE 3.3)
        # -------------------------------------------------------
        # Clears the match and correct sets for the next learning iteration
        self.population.clearSets()
```

Listing 5.17: A single learning iteration on a training instance

```
    def makeMatchSet(self, phenotype, exploreIter):
        """ Constructs a match set from the population. Covering is initiated if
    the match set is empty or a rule with the current correct phenotype is absent.
    """
        # Initial values
        state = phenotype[0]
        phenotype = phenotype[1]
        # Covering check: Twofold (1)checks that a match is present,
        # and (2) that at least one match dictates the correct phenotype.
        doCovering = True
        setNumerositySum = 0

        # ---------------------------------------------------------
        # MATCHING
        # ---------------------------------------------------------
        cons.timer.startTimeMatching()
        # Go through the population
        for i in range(len(self.popSet)):
            # One classifier at a time
            cl = self.popSet[i]
            # Check for match
            if cl.match(state):
                # If match - add classifier to match set
                self.matchSet.append(i)
                # Increment the set numerosity sum
                setNumerositySum += cl.numerosity

                # Covering Check------------------------------------
                # Discrete phenotype
                if cons.env.formatData.discretephenotype:
                    # Check for phenotype coverage
                    if cl.phenotype == phenotype:
                        doCovering = False
                # Continuous phenotype
                else:
                    # Check for phenotype coverage
                    if float(cl.phenotype[0]) <= float(phenotype) <= float(
                            cl.phenotype[1]):
                        doCovering = False
        cons.timer.stopTimeMatching()

        # ---------------------------------------------------------
        # COVERING
        # ---------------------------------------------------------
        while doCovering:
            newCl = Classifier(setNumerositySum + 1, exploreIter, state, phenotype)
            self.addClassifierToPopulation(newCl, True)
            # Add covered classifier to matchset
            self.matchSet.append(len(self.popSet) - 1)
            doCovering = False
```

Listing 5.18: Creating a match set

```
    def makeCorrectSet(self, phenotype):
        """ Constructs a correct set out of the given match set. """
        for i in range(len(self.matchSet)):
            ref = self.matchSet[i]
            # -----------------------------------------------------
            # DISCRETE phenotype
            # -----------------------------------------------------
            if cons.env.formatData.discretephenotype:
                if self.popSet[ref].phenotype == phenotype:
                    self.correctSet.append(ref)
            # -----------------------------------------------------
            # CONTINUOUS phenotype
            # -----------------------------------------------------
            else:
                if float(phenotype) <= float(self.popSet[ref].phenotype[1]) and
                float(phenotype) >= float(self.popSet[ref].phenotype[0]):
                    self.correctSet.append(ref)
```

Listing 5.19: Creating a correct set from match set

```
    def updateParams(self, exploreIter):
        """ Updates all relevant parameters in the current match and correct sets.
    """
        matchSetNumerosity = 0
        for ref in self.matchSet:
            matchSetNumerosity += self.popSet[ref].numerosity

        for ref in self.matchSet:
            self.popSet[ref].updateExperience()
            self.popSet[ref].updateMatchSetSize(matchSetNumerosity)
            if ref in self.correctSet:
                self.popSet[ref].updateCorrect()

            self.popSet[ref].updateAccuracy()
            self.popSet[ref].updateFitness()
```

Listing 5.20: Update classifier parameters in match and correct sets

```
    def doCorrectSetSubsumption(self):
        """ Executes correct set subsumption.  The correct set subsumption looks
    for the most general subsumer classifier in the correct set
        and subsumes all classifiers that are more specific than the selected one.
    """
        subsumer = None
        for ref in self.correctSet:
            cl = self.popSet[ref]
            if cl.isSubsumer():
                if subsumer is None or cl.isMoreGeneral(subsumer):
                    subsumer = cl

       # If a subsumer was found, subsume all more specific classifiers in the
    correct set
        if subsumer != None:
            i = 0
            while i < len(self.correctSet):
                ref = self.correctSet[i]
                if subsumer.isMoreGeneral(self.popSet[ref]):
                    subsumer.updateNumerosity(self.popSet[ref].numerosity)
                    self.removeMacroClassifier(ref)
                    self.deleteFromMatchSet(ref)
                    self.deleteFromCorrectSet(ref)
                    i = i - 1
                i = i + 1
```

Listing 5.21: Running subsumption on the correct set

Record results into separate file

Finally, after every completed algorithm run (after all iterations have been completed), the results are written out into a separate file and the process repeats from the start again.

```
    ...
    f = open("RESULTS_FILE.txt", 'a')
    f.write(" Accuracy: " + str(kfold_accuracy) + " Total time: " + str(total) + "
    Rules: " + str(OutputFileManager.totalPopulationSize) + "\n")
```

Listing 5.22: Clearing output folder

5.6.2 XCS Implementation Run

The XCS algorithm implementation was created following closely [Butz and Wilson, 2001], which, since its publication, has become the standard to follow when implementing XCS. Since many changes have been proposed by various authors to XCS to improve its robustness, resulting in at times major changes to the implementation, a clarification must be given on what version of XCS the algorithmic description provided in [Butz and Wilson, 2001] is based on. The algorithmic description provided is based on [Wilson, 1995], which was one of the first available descriptions of XCS, but does include some changes that were added in later XCS publications, specifically:

- The order of updating the classifier parameters is changed for convenience.
- Calculation of accuracy is done via a power law function as proposed in [Wilson, 2000], rather than via an exponential function that was used in the original publication.
- Covering occurs based on the presence of classifiers in the match set, rather than the mean prediction of the population, as was suggested in [Wilson, 1995]. The advantage of such approach, according to the authors, is an increase in the program execution speed.
- Genetic algorithm is applied to the action set, as has been described in [Wilson, 1998].

This section will provide the source code for the XCS implementation utilised in this project, based on the pseudocode for components described in [Butz and Wilson, 2001]. Since the XCS implementation has been based on the UCS implementation described in detail above, the way the components operate with each other within the overall system would be similar, i.e. the way the data is loaded and formatted would be the same for both XCS and UCS.

Running the XCS Algorithm

At first, similarly as to how it was done with UCS, a step-through for a single XCS learning iteration is provided below. It is worth noticing that the first major change is that the algorithm has now two types of learning iterations - explore iterations and exploit iterations (recall Section 3.3.3).

```
    def run_XCS(self):
        """ Runs the initialized XCS algorithm. """
        # ---------------------------------------------------------------
        trackedAccuracy = 0
        ...
        exploreMode = 1
        # ---------------------------------------------------------
        # MAJOR LEARNING LOOP
        # ---------------------------------------------------------
        while self.iteration < cons.maxLearningIterations:
            # -----------------------------------------------------------
            # GET NEW INSTANCE AND RUN A LEARNING ITERATION
            # -----------------------------------------------------------
            self.selectAction(exploreMode)
            # Increment current learning iteration
            self.iteration += 1
            ...
            ...
            # -----------------------------------------------------------
            # CHECKPOINT - COMPLETE EVALUATION OF POPULATION
            # -----------------------------------------------------------
            if self.iteration in cons.learningCheckpoints:
                ...
                self.population.runPopAveEval()
                self.population.runAttGeneralitySum(True)
                # Preserves learning position in training data
                cons.env.startEvaluationMode()
                # If a testing file is available.
                if cons.testFile != 'None' or cons.kfold > 0:
                    ...
                # Only a training file is available
                else:
                    if cons.env.format_data.discretephenotype:
                        train_eval = self.doPopEvaluation(True)
                        test_eval = None
                    else:
                        test_eval = None
                ...
                # Write output files ----------------------------------
                OutputFileManager().writePopStats(cons.outFileName, train_eval,
                test_eval, self.iteration, self.population, self.trackedResults)
                OutputFileManager().writePop(cons.outFileName, self.iteration,
                self.population)
                # ---------------------------------------------------------
                ...
            # Switch between explore and exploit modes
            if cons.exploration == 0.5:
                exploreMode = 1 - exploreMode

        # Once XCS has reached the last learning iteration, close the tracking file
        self.learnTrackOut.close()
        print("XCS Run Complete")
        # Compress population
        self.population.compressPopulation()
        ...
```

Listing 5.23: XCS Major Learning Loop

```
    def runExploit(self, phenotype):
        """ Run an exploit iteration. """
        self.population.generateMatchSet(phenotype[0], self.iteration)
        cons.timer.startTimeEvaluation()
        prediction = Prediction(self.population)
        selectedAction = prediction.decide(exploring=False)
        if selectedAction == phenotype[1]:
            reward = 1000.0
            self.trackedResults.append(1)
        else:
            reward = 0.0
            self.trackedResults.append(0)
        cons.timer.stopTimeEvaluation()
        self.population.generateActionSet(selectedAction)
        self.population.updateSets(reward)
        self.population.clearSets()  # Clears the match and action sets for the
    next learning iteration
```

Listing 5.24: Run an exploit iteration

```
    def runExplore(self, phenotype):
        """ Run an explore learning iteration. """
        reward = 0.0
        # ---------------------------------------------------------------
        # FORM A MATCH SET - includes covering
        # ---------------------------------------------------------------
        self.population.generateMatchSet(phenotype[0], self.iteration)
        # ---------------------------------------------------------------
        # MAKE A PREDICTION - utilized here for tracking estimated learning
        # progress.  Typically used in the explore phase of many LCS algorithms.
        # ---------------------------------------------------------------
        ...
        prediction = Prediction(self.population)
        selectedAction = prediction.decide(exploring=True)
        # -----------------------------------------------------------
        # DISCRETE PHENOTYPE PREDICTION
        # -----------------------------------------------------------
        if selectedAction == phenotype[1]:
            reward = 1000.0
        ...
        # ---------------------------------------------------------------
        # FORM AN ACTION SET
        # ---------------------------------------------------------------
        self.population.generateActionSet(selectedAction)
        # ---------------------------------------------------------------
        # UPDATE PARAMETERS
        # ---------------------------------------------------------------
        self.population.updateSets(reward)
        # ---------------------------------------------------------------
        # SUBSUMPTION - APPLIED TO MATCH SET
        # ---------------------------------------------------------------
        if cons.doActionSetSubsumption:
            cons.timer.startTimeSubsumption()
            self.population.doActionSetSubsumption()
            cons.timer.stopTimeSubsumption()
        # ---------------------------------------------------------------
        # RUN THE GENETIC ALGORITHM
        # ---------------------------------------------------------------
        self.population.runGA(self.iteration, phenotype[0])
        self.population.clearSets()
```

Listing 5.25: Run an explore iteration

Generating a Match Set

```
    def generateMatchSet(self, state, iteration):
        """ Constructs a match set from the population.
        Covering is initiated if the match set is empty or
        total prediction of rules in match set is too low. """
        # Initial values
        match_set = []
        self.current_instance = state
        # ---------------------------------------------------------
        # MATCHING
        # ---------------------------------------------------------
        ...
        # Go through the population
        for i in range(len(self.population)):
            # One classifier at a time
            cl = self.population[i]
            # Check for match
            if cl.doesMatch(state):
                # If match - add classifier to match set
                self.matchSet.append(i)
                if cl.action not in match_set:
                    match_set.append(cl.action)
        # ---------------------------------------------------------
        # COVERING
        # ---------------------------------------------------------
        if len(match_set) < cons.theta_mna:
            missing_actions = [a for a in cons.env.format_data.phenotypeList if a
                               not in match_set]
            for action in missing_actions:
                new_cl = Classifier(iteration, state, action)
                self.addClassifierToPopulation(new_cl)
                # Add created classifier to match set
                self.matchSet.append(len(self.population) - 1)
                match_set.append(new_cl.action)
            if len(match_set) >= cons.theta_mna:
                self.deletion()
                match_set = []
                for i in self.matchSet:
                    if self.population[i].action not in match_set:
                        match_set.append(self.population[i].action)
```

Listing 5.26: Generate a Match Set

Generating an Action Set

```
    def generateActionSet(self, selected_action):
        """ Constructs an action set out of the given match set. """
        for ref in self.matchSet:
            if self.population[ref].action == selected_action:
                self.actionSet.append(ref)
```

Listing 5.27: Generate Action set out of Match set

Updating Parameters

```
    def updateSets(self, reward):
        """ Updates relevant parameters in the current action sets. """
        action_set_number = 0
        for ref in self.actionSet:
            action_set_number += self.population[ref].numerosity
        accuracy_sum = 0.0
        for ref in self.actionSet:
            self.population[ref].updateActionExp()
            self.population[ref].updateActionSetSize(action_set_number)
            self.population[ref].updateXCSParameters(reward)
            accuracy_sum +=
                self.population[ref].accuracy * self.population[ref].numerosity
        for ref in self.actionSet:
            self.population[ref].updateFitness(
            self.population[ref].accuracy * self.population[ref].numerosity /
            accuracy_sum)
```

Listing 5.28: Update classifier relevant parameters

```
    def updateXCSParameters(self, reward):
        """ Updating XCS prediction payoff, prediction error and fitness. """
        payoff = reward
        if self.action_cnt >= 1.0 / cons.beta:
            self.error += cons.beta * (abs(payoff - self.prediction) - self.error)
            self.prediction += cons.beta * (payoff - self.prediction)
        else:
            self.error = (self.error * (self.action_cnt - 1) +
                          abs(payoff - self.prediction)) / self.action_cnt
            self.prediction = (self.prediction * (self.action_cnt - 1) + payoff) /
                          self.action_cnt
        #Updating Fitness
        if self.error <= cons.offset_epsilon:
            self.accuracy = 1.0
        else:
            self.accuracy = cons.alpha * ((self.error / cons.offset_epsilon) ** (
                -cons.nu))
```

Listing 5.29: Updating XCS prediction payoff + prediction error and fitness

Apply Action Set Subsumption

```
    def doActionSetSubsumption(self):
        """ Executes action set subsumption. """
        subsumer = None
        for ref in self.actionSet:
            cl = self.population[ref]
            if cl.isPossibleSubsumer():
                if subsumer is None
                    or
                    len(subsumer.specifiedAttributes) > len(cl.specifiedAttributes)
                    or ((
                    len(subsumer.specifiedAttributes)== len(cl.specifiedAttributes)
                    and
                    random.random() < 0.5)):
                    subsumer = cl

        # If a subsumer was found, subsume all more specific classifiers in the
        # action set
        if subsumer != None:
            i = 0
            while i < len(self.actionSet):
                ref = self.actionSet[i]
                if subsumer.isMoreGeneral(self.population[ref]):
                    subsumer.updateNumerosity(self.population[ref].numerosity)
                    self.removeMacroClassifier(ref)
                    self.deleteFromMatchSet(ref)
                    self.deleteFromActionSet(ref)
                    i = i - 1
                i = i + 1
```

Listing 5.30: Method for Action Set Subsumption

5.7 Result Output

The results of the LCS run are shown in the terminal console during and after the LCS run is completed (Figure 5.10). The final output is also written to a separate file (Figure 5.11), from where the results can then be taken and imported into a spreadsheet software, e.g. Microsoft Excel. Specifically for this project, LaTeX tables were created via an online website[6]. The resulting tables (Chapter 6) are sufficient to identify a pattern that the results follow, although it is arguable that line graphs would make the results even clearer - unfortunately, the amount of results gathered would require at least 50 line graphs to represent, if not more - this could hinder the readability of this report. This report follows the belief that if the reader finds it necessary to generate a line graph for clarity following the gathered results, they can easily do so by simply importing the tables into Excel and generating graphs for data that they specifically need.

```
------------------------------------------------------------
Running Population Evaluation after 10001 iterations.
----------------------------------------------
TRAINING Accuracy Results:-------------
Instance Coverage = 100.0%
Prediction Ties = 0.0%
1332 out of 1639 instances covered and correctly classified.
Standard Accuracy = 0.8126906650396584
Standard Accuracy (Adjusted) = 0.8126906650396584
Balanced Accuracy (Adjusted) = 0.8126906650396584
----------------------------------------------
TESTING Accuracy Results:-------------
Instance Coverage = 100.0%
Prediction Ties = 0.0%
312 out of 409 instances covered and correctly classified.
Standard Accuracy = 0.7628361858190709
Standard Accuracy (Adjusted) = 0.7628361858190709
Balanced Accuracy (Adjusted) = 0.7628361858190709
Writing Population Statistical Summary File...
Writing Population as Data File...
984
Continue Learning...
------------------------------------------------------------
Run time in seconds: 36.83
LCS Run Complete
AVERAGE ACCURACY AFTER 5-FOLD CROSS VALIDATION is 0.78515625
Total run time in seconds: 188.27
```

Figure 5.10: Snippet of log output after one complete LCS algorithm run.

```
Accuracy: 0.78515625 Total time: 188.27 Rules: 984
```

Figure 5.11: Entry in results file, corresponding to algorithm run above.

[6] https://www.tablesgenerator.com/

Chapter 6

Analysis of Acquired Results

This chapter will provide context, analysis and explanation for the gathered result data, as well as provide recommendations for parameter values based on the information gathered. More information on the experiment environment and methodology can be found in Section 5.3 (Methodology).

6.1 Parameter I - Environment iterations

The iteration parameter (I) specifies the maximum number of learning iterations before the complete evaluation of the algorithm commences. During a single iteration, a single training instance is retrieved from the training data set and the LCS performs a learning iteration over it. The learning loop continues until a termination condition is met – in the implementation devised for this project, a deliberate choice was made to end the learning loop when the number of learning iterations reaches the limit specified in the configuration file, rather than the training accuracy reaching a certain threshold. This decision allows the algorithm to evolve a larger amount of accurate, environment-representative rules, as well as enables the conduction of a fair test and comparison between different benchmark result for identical amount of iterations.

The following results (Table 6.1-6.6) demonstrate the relationship between value of learning operations, algorithm accuracy and time elapsed.

It is evident that the number of iterations and the time it takes to complete the learning cycle are directly proportional – doubling the number of iterations doubles the execution time for the cycle. This is true for all benchmark problems and is not affected by the algorithm type. Contrarily, results demonstrate that the number of rules generated is related to the algorithm being utilised and the problem environment being mapped. Note: population rule compaction has been deactivated for easier demonstration of this effect.

Iterations	%correct	Rules	Time (s)
10000	47.29	992	74.29
20000	46.53	993	209.19
40000	47.17	992	382.63
80000	46.46	993	724.86
160000	46.83	989	1623.95
320000	45.85	991	2944.32
640000	45.65	995	5879.7
1280000	45.85	995	11479.33
2000000	45.74	994	17935.94

Table 6.1: 11-bit Parity - XCS

Iterations	%correct	Rules	Time (s)
10000	46.66	993	139.88
20000	45.52	997	392.26
40000	46.92	995	748.41
80000	46.14	993	1511.95
160000	45.94	995	2070.82
320000	46.39	993	6119.20
640000	46.12	992	11889.61
1280000	45.96	995	23898.11
2000000	45.85	995	35831.59

Table 6.2: 11-bit Parity - UCS

Iterations	%correct	Rules	Time (s)
10000	85.61	984	72.89
20000	87.24	978	134.51
40000	88.21	973	343.09
80000	90.27	969	509.25
160000	89.62	957	1372.99
320000	88.82	972	2692.54
640000	88.94	971	5223.53
1280000	89.19	969	10603.76
2000000	89.55	968	16557.24

Table 6.3: 11-bit Multiplexer - XCS

Iterations	%correct	Rules	Time (s)
10000	81.87	974	134.35
20000	84.66	964	392.76
40000	81.39	973	694.98
80000	84.42	967	1373.99
160000	86.62	966	2693.75
320000	85.34	967	6725.14
640000	83.84	970	11027.44
1280000	83.94	982	31167.85
2000000	82.32	986	47262.68

Table 6.4: 11-bit Multiplexer - UCS

Iterations	%correct	Rules	Time (s)
10000	96.93	568	49.87
20000	97.63	468	80.44
40000	97.94	462	141.76
80000	98.15	450	249.85
160000	97.95	447	483.59
320000	98.11	442	921.04
640000	98.13	441	1851.29
1280000	98.06	442	3778.48
2000000	98.19	437	7368.04

Table 6.5: 9-bit Position - XCS

Iterations	%correct	Rules	Time (s)
10000	97.02	812	122.58
20000	97.09	815	226.69
40000	97.21	816	449.01
80000	97.19	820	907.03
160000	97.29	824	1868.14
320000	97.65	826	3164.74
640000	98	835	6239.05
1280000	97.84	836	9,991.28
2000000	97.92	839	14,674.24

Table 6.6: 11-bit Position - UCS

For a semi-pattern problem, such as the Parity problem, results demonstrate a constantly high rule count that is close to the specified population size - this can be explained by the fact that rules for such type of problems are not very easily optimisable and therefore the rule generality is barely affected. This in turn results in large amount of rules being inaccurate and inexperienced, what has a direct effect on the algorithm accuracy - results show that it does not increase, but rather remains largely the same (within acceptable error bounds) or decreases.

On the other hand, for problems containing recognisable and generalisable patterns, e.g. Position or Multiplexer, the increase in the number of generations causes a decrease in the amount of rules generated, indicating that the population is becoming more optimised and generalised. Increase in iterations also has a positive effect on the accuracy of the algorithm, showing slow increase, but increase nonetheless. This is supported by experiments conducted by [Urbanowicz and Moore, 2015], who during their experimentation with the ExSTraCS 2.0 LCS on a Multiplexer problem recorded that an increase in the number of iterations yields a small improvement in testing accuracy and a large increase in rule generality, in turn causing a decrease of the macro population size.

A0	A1	R0	R1	R2	R3	phenotype	Fitness	Accuracy	Numerosity	CorrectCount	MatchCount
0	**1**	#	**1**	#	#	**1**	1.0	1.0	16	938	938
#	**0**	**0**	#	**0**	#	**0**	1.0	1.0	12	753	753
#	#	#	**0**	**0**	**0**	**0**	0.161	0.83	2	688	826
0	#	**1**	**1**	#	#	**1**	1.0	1.0	10	762	762
#	**1**	**0**	**0**	**1**	#	**0**	0.059	0.75	1	110	146

Table 6.7: Rule Comparison in 6-bit Multiplexer: after 10,000 iterations (UCS)

A0	A1	R0	R1	R2	R3	phenotype	Fitness	Accuracy	Numerosity	CorrectCount	MatchCount
0	**1**	#	**1**	#	#	**1**	1.0	1.0	19	274365	274365
#	**0**	**0**	#	**0**	#	**0**	1.0	1.0	19	234964	234964
#	#	#	**0**	**0**	**0**	**0**	1.0	1.0	19	196066	196066
0	#	**1**	**1**	#	#	**1**	1.0	1.0	14	235267	235267
#	**1**	**0**	**0**	**1**	#	**0**	1.0	1.0	13	5414	5414

Table 6.8: Rule Comparison in 6-bit Multiplexer: after 2,000,000 iterations (UCS)

Tables 6.7 and 6.8 demonstrate the effect of iteration increase for XCS on a 6-bit Multiplexer problem. Two extremes were used for the iteration values, and 5 random rules from each generated population were compared. As expected, increase in number of iterations has a positive effect on rule fitness, rule accuracy and rule numerosity - with more iterations and as a consequence more rules generated, it is expected that numerosity will increase. With the increase of numerosity rules, the algorithm accuracy should therefore also increase for higher iteration values in comparison to lower iteration values.

Nonetheless, the accuracy and rule count will eventually reach a stagnation point and will no longer dramatically change - this is dependent on the data being learned and on the extent to which it can be broken down into self-contained patterns. Once the algorithm passes this "stagnation point", i.e. the peak value at which the algorithm accuracy is maximum and rule count is minimum, an increase in the number of iterations will not yield any improvements to the results - as can be seen from the Multiplexer and Position tables above, in most cases this appears to be between 40,000-80,000 iterations. The challenge is therefore to find this peak value and consider it as the largest possible viable value for I.

Unfortunately, the value to which parameter I should be set is heavily dependent on the data being learned and there is no way to theoretically determine what that exact value should be. The common rule evident for all LCS and all problem domains is to set the number of iterations as high as possible and no less than the size of the unique instances in the training data set - importantly, with a consideration for the computation time (what value of computation time is considered acceptable will depend on the domain in which the LCS is being utilised). To give a sense of range to which the parameter I could be set, XCS was able to solve the 20-bit Multiplexer problem in about 50000 iterations [Iqbal et al., 2013] - it is therefore possible to conclude that for well-defined, pattern-separable problems, acceptable results may be achieved with the number of iterations being set between **10,000-50,000** cycles. For larger and the more complex problems, a larger number of iterations must be considered and set.

6.2 Parameter N - Population size

The maximum population size parameter ($\boldsymbol{N}$) specifies the allowed maximum LCS population size before the deletion mechanism kicks in to delete excessive rules to maintain this specified amount. This parameter specifically refers to the micro-population size (the sum of numerosity's (copies) of unique classifiers), rather than the macro-population (simply a count of the unique classifiers without consideration for any copies). Numerosity is an important metric to be considered within the population, as it allows to separate rules by quality, with higher-numerosity rules being of higher value than lower-numerosity rules – this separation is key to avoiding a situation when a good rule is deleted from the population by chance, reducing the capability of the algorithm to learn.

This is usually one of the first parameters to be tweaked in an LCS and has an important influence on the result obtained by the system.

The following results (Table 6.9-6.14) demonstrate the relationship between the population size, algorithm accuracy, rules generated and time elapsed. Results suggest several important patterns.

For the Parity problem, used as an example of a problem with low generality, it is evident that with the increase of population the algorithm accuracy is decreasing in a negative exponential curve pattern. This is true for both supervised and unsupervised algorithms. This is closely related to the logarithmic increase of generated rules as the population size increases (population compression has been turned off). Usually, more rules in the population should improve the algorithm accuracy, but since the problem being learned has low to no generality, the rules generated have a low level of accuracy and experience – this in turn results in the decrease of the algorithm accuracy. Incorporating population compression (subsuming overly specific classifiers and then eliminating inexperienced and inaccurate classifiers) has further proven the point above, with all rules within the population being deleted due to none of them being able to exceed the specified deletion experience and error thresholds. The computation time is also closely related to the rule count – the more rules generated, the longer it will take for the algorithm to run – without yielding any benefits, as seen above. It is quite possible that since LCS is not the correct tool to be used as a prediction machine for such problems, other approaches for such category of problems should be considered.

N	%correct	Rules	Time (s)
500	47.2	497	37.74
1000	47.35	993	74.56
2000	44.37	1974	138.59
4000	43.24	3924	237.87
8000	39.71	7672	314.40
16000	39.63	9343	298.71
32000	39.56	9377	299.92
50000	39.36	9389	300.05

Table 6.9: 11-bit Parity - XCS

N	%correct	Rules	Time (s)
500	47.74	499	102.08
1000	45.54	996	207.15
2000	44.01	1982	389.74
4000	40.85	3923	712.26
8000	36.84	7699	1233.19
16000	31.48	14862	1591.70
32000	30.70	17317	1474.86
50000	30.75	17326	1428.44

Table 6.10: 11-bit Parity - UCS

N	%correct	Rules	Time (s)
500	71.39	497	51.49
1000	85.71	981	73.02
2000	97.64	1850	179.40
4000	99.6	3550	313.86
8000	99.18	7207	433.95
16000	97.56	13956	515.21
32000	98.41	14015	641.03
50000	97.89	14463	639.47

Table 6.11: 11-bit Multiplexer - XCS

N	%correct	Rules	Time (s)
500	69.67	498	133.26
1000	82.26	980	136.03
2000	93.20	1879	257.68
4000	96.48	3583	466.23
8000	97.20	6965	760.04
16000	97.17	13679	891.78
32000	97.39	15030	867.05
50000	97.34	14922	857.67

Table 6.12: 11-bit Multiplexer - UCS

N	%correct	Rules	Time (s)
500	96.12	236	34.15
1000	96.8	563	67.49
2000	97.55	1067	114.58
4000	97.88	2224	201.07
8000	97.56	3098	199.61
16000	97.75	4335	198.91
32000	97.6	4190	176.87
50000	97.73	4356	176.35

Table 6.13: 9-bit Position - XCS

N	%correct	Rules	Time (s)
500	95.51	411	65.44
1000	96.98	806	119.51
2000	97.06	1625	238.39
4000	97.42	3390	434.44
8000	97.51	6472	599.64
16000	97.61	7206	570.36
32000	97.63	7513	574.34
50000	97.74	7723	571.15

Table 6.14: 11-bit Position - UCS

N	%correct	Rules	Time (s)
500	51.12	0	48.62
1000	59.79	13	93.75
2000	87.94	84	168.22
4000	96.6	177	296.36
8000	96.3	176	372.72
16000	96.26	173	343.26
32000	96.09	176	336.58
50000	95.82	185	320.82

Table 6.15: 11-bit Multiplexer - XCS - Compact

N	%correct	Rules	Time (s)
500	57.21	0	42.19
1000	59.11	16	91.67
2000	66.35	86	187.70
4000	77.00	190	294.09
8000	85.33	295	390.96
16000	82.32	252	346.93
32000	82.49	249	321.95
50000	81.94	252	319.43

Table 6.16: 11-bit Position - XCS - Compact

A different pattern is evident whilst looking at problems with adequate levels of generality, namely the Position and Multiplexer problems. Regardless of the algorithm used, an increase in the population size has yielded a dramatic increase in accuracy, following a logarithmic pattern (Position problem appears constant due to the rise in accuracy taking place when population size is smaller than 500, which is outside the range of boundaries being investigated in this project). Like problems with low generality, increase in population size has yielded a logarithmic increase in the rules generated and, consequentially, the computation time for the algorithm – but this time due to the rules being able to better generalise the aforementioned benchmark problems, increase in rule count has a direct positive contribution to the accuracy of the algorithm. Using rule compression has also supported the above finding, as can be seen in the Table 6.15 and 6.16.

The results gained allow to make several conclusions:

1. Parameter N plays a vital role in a learning classifier system and it is crucial to get the value correct for the system to yield adequate results.

2. If N is set too low, the LCS will not be able to build up and generate a good model due to not being able to maintain good classifiers. This will result in low accuracy of prediction of new instances.

3. If N is set too high, unnecessary computational resources will be consumed without any benefit. This is not only inefficient but can also lead to longer time for the algorithm to converge to an optimal solution.

The challenge therefore is to find such value of N that it is as low as possible, but at the same time provides adequate accuracy. This idea has been summarised well in [Butz and Wilson, 2001] as:

> *"The population size, N, should be large enough so that, starting from an empty population, covering occurs only at the very beginning of a run."*

Values above suggest that population size in the range of **[1000, 4000]** will yield adequate accuracy for majority of classification tasks, with smaller problems employing the lower values of that range and more large and complex problems employing the upper value of the range. This conclusion agrees with the proposals regarding optimal population size in [Urbanowicz and Browne, 2017] and in [Butz, 2004]. Note that if population compression is to be utilised, the population size will need to be set on the generous side of the proposed range.

6.3 Parameter $P\#$ - "Don't care" probability

The parameter ($\boldsymbol{P\#}$) specifies the probability of inserting a "don't care", also known as "wild card" or simply "hashtag" (denoted as #), symbol into a classifier that has been generated during the covering stage. In contrast, parameter p_spec can be utilised instead of this parameter, where p_spec specifies the probability of inserting an attribute rather than #. Essentially, the relationship between the two is $P\# = 1 - p\_spec$. Both approaches are valid and have been utilised in literature. The common theme is that during the covering operation, a rule that perfectly matches the environment is taken as base and is generalised by substituting a condition component with a wild card symbol with probability of $P\#$. Alternatively, if p_spec is used, the process works similarly by taking a fully generalised (containing all #) rule as base and "specifying" correct component values with probability of p_spec.

If population compression is enabled, it is especially important to get the value of $P\#$ correct, as it can dramatically influence the accuracy of the algorithm. For example, when applying the two extreme ranges for $P\#$ to 11-bit Multiplexer Problem for 10000 iterations and population of N=4000 with population compression enabled, a significant difference in accuracy can be observed.

P#	%correct	Rules	Time (s)
0	**84.33**	**69**	**316.36**
1	**96.01**	**189**	**264.30**

Table 6.17: P# Extremes Comparison

The tables below (Table 6.18-6.23) demonstrate the relationship between $P\#$, algorithm accuracy, rules generated and time elapsed - without population compression: The effect of $P\#$ on algorithm accuracy has been studied prior and the results gained support the commonly agreed theory. The setting of $P\#$ is very closely related to the problem to which the LCS is being applied. For problems that can not be easily generalised, such as the Parity problem, $P\#$ must be set to zero; for problems that contain high levels of generality (Multiplexer / Position) or for data sets containing large amounts of redundancy, $P\#$ must be set high. Results gathered compliment the above conclusion: as can be seen, $P\#$ has limited effect on the Parity problem which is limited by its nature of not being able to be accurately generalisable. More reflective of $P\#$ influence are results gained for Multiplexer and Position problems – with the increase of value of $P\#$, the accuracy of the algorithm increased, and the run time decreased – this is due to more "good" rules being generated, as is proven when population compression is turned on.

P#	%correct	Rules	Time (s)
0	46.31	995	74.15
0.1	46.23	994	73.92
0.2	46.36	992	73.91
0.3	45.6	992	74.7
0.4	45.86	993	75.51
0.5	45.76	992	74.94
0.6	46.75	991	73.21
0.7	46.46	992	73.64
0.8	46.88	993	72.6
0.9	46.29	992	71.75
0.99	46.84	992	72.93

Table 6.18: 11-bit Parity - XCS

P#	%correct	Rules	Time (s)
0	46.06	995	131.80
0.1	46.78	993	139.31
0.2	46.4	994	140.48
0.3	45.86	995	139.97
0.4	46.31	994	140.10
0.5	45.78	993	141.95
0.6	46.28	995	141.56
0.7	46.62	993	138.49
0.8	46.50	995	136.63
0.9	45.85	995	134.10
0.99	46.58	995	137.76

Table 6.19: 11-bit Parity - UCS

P#	%correct	Rules	Time (s)
0	83.12	982	73.25
0.1	83.09	989	73.18
0.2	85.16	985	72.73
0.3	84.84	983	71.46
0.4	85.92	982	71.41
0.5	85.58	983	72.00
0.6	85.98	986	73.64
0.7	86.00	983	71.26
0.8	84.98	988	70.20
0.9	85.81	981	70.02
0.99	85.64	982	69.50

Table 6.20: 11-bit Multiplexer - XCS

P#	%correct	Rules	Time (s)
0	81.35	978	190.54
0.1	81.99	981	189.49
0.2	82.16	982	199.94
0.3	81.90	978	197.32
0.4	82.26	977	194.94
0.5	81.48	977	195.04
0.6	82.35	976	201.00
0.7	81.78	980	200.14
0.8	83.10	978	193.74
0.9	83.18	980	188.59
0.99	83.14	979	200.49

Table 6.21: 11-bit Multiplexer - UCS

P#	%correct	Rules	Time (s)
0	97.21	569	64.32
0.1	97.15	572	64.35
0.2	97.18	568	64.12
0.3	97.23	567	63.59
0.4	97.15	533	59.65
0.5	97.46	515	62.53
0.6	97.40	506	55.92
0.7	97.46	515	53.76
0.8	97.63	531	49.68
0.9	97.43	546	49.09
0.99	97.45	547	48.77

Table 6.22: 9-bit Position - XCS

P#	%correct	Rules	Time (s)
0	97.02	782	83.99
0.1	97.09	823	83.26
0.2	97.12	806	82.69
0.3	97.02	834	83.09
0.4	97.19	825	83.61
0.5	96.98	830	83.04
0.6	96.97	831	82.99
0.7	97.06	842	82.01
0.8	97.08	839	83.15
0.9	97.09	832	85.04
0.99	96.88	836	100.69

Table 6.23: 11-bit Position - UCS

The tables below (Table 6.24-6.25) demonstrate the relationship between $P\#$, algorithm accuracy, rules generated, and time elapsed with population compression being turned on for the Multiplexer problem.

P#	%correct	Rules	Time (s)
0	-	-	-
0.1	57.89	7	89.04
0.2	61.20	11	91.86
0.3	61.20	14	92.45
0.4	61.18	10	94.58
0.5	59.95	14	94.83
0.6	59.04	9	88.28
0.7	62.50	13	84.46
0.8	61.61	12	88.12
0.9	60.45	12	87.44
0.99	60.34	11	86.82

Table 6.24: 11-bit Multiplexer, N=1000

P#	%correct	Rules	Time (s)
0	70.41	30	180.39
0.1	84.35	72	161.23
0.2	86.02	81	156.03
0.3	89.50	83	162.50
0.4	88.20	86	155.06
0.5	89.79	97	158.30
0.6	89.66	83	165.48
0.7	88.05	85	156.58
0.8	88.13	87	166.32
0.9	88.70	90	160.69
0.99	88.11	88	158.08

Table 6.25: 11-bit Multiplexer, N=2000

As can be seen, with the increase of $P\#$, algorithm accuracy increases together with the rule count, and the run-time slowly decreases.

The challenge is to set $P\#$ to a value which will enable the classifier system to generate rules in the population that are not overly-general (such classifiers may maintain high fitness and succeed in certain cases, but will not work for all situations and may lead to degradation of overall performance [Karlsen and Moschoyiannis, 2018a]), but at the same time not overly-specific, since this will require more rules to provide a result that covers the entire problem map, what in turn might require an increase in parameter N, leading to unnecessary increase in algorithm running time.

For problems that are not easily accurately generalisable, $P\#$ must be set low – in the range of **[0, 0.3]**, with worst situations requiring $P\#$ to be set as low as **zero**. For problems that contain a cleanly generalisable pattern or where high generality is needed (e.g. noisy problems, containing duplicates, irrelevant entries, large number of features), $P\#$ should be set to value of the higher end of the range – from **[0.7, 0.99]**. Alternatively, in situations where determination of $P\#$ is too difficult or impossible, rule specificity limit (RSL) [Urbanowicz and Moore, 2015] may be utilised (as was described in Chapter 4).

6.4 Parameter ν - Fitness exponent

The fitness exponent parameter (ν) plays an important and direct role in calculating the updated value for the fitness parameter of the classifier.

Results captured (Tables 6.26-6.31) demonstrate the relationship between the fitness exponent, algorithm accuracy, rules generated and time elapsed.

ν	%correct	Rules	Time (s)
1	47.01	991	109.65
2	46.50	992	103.08
3	46.81	994	102.81
4	46.04	993	103.89
5	46.55	994	108.52
6	45.90	993	96.51
7	45.83	994	96.85
8	46.33	994	97.00
9	45.35	997	100.70
10	46.06	995	101.02

Table 6.26: 11-bit Parity - XCS

ν	%correct	Rules	Time (s)
1	46.98	992	200.11
2	46.37	993	188.00
3	45.62	993	192.97
4	45.58	993	196.50
5	45.93	991	196.26
6	45.92	997	197.12
7	46.97	997	213.21
8	46.39	997	205.71
9	46.52	996	205.93
10	46.58	997	206.20

Table 6.27: 11-bit Parity - UCS

ν	%correct	Rules	Time (s)
1	80.71	985	76.08
2	82.67	989	75.63
3	83.29	985	76.31
4	84.44	987	74.99
5	85.47	986	74.77
6	85.60	987	74.62
7	85.47	983	74.76
8	84.96	985	74.33
9	85.46	983	73.49
10	85.34	984	73.71

Table 6.28: 11-bit Multiplexer - XCS

ν	%correct	Rules	Time (s)
1	73.64	980	133.43
2	76.40	986	133.57
3	78.63	982	135.58
4	80.20	979	132.78
5	81.60	981	136.21
6	83.63	971	138.00
7	85.40	968	138.22
8	85.49	971	139.90
9	85.60	962	140.89
10	86.12	966	140.00

Table 6.29: 11-bit Multiplexer - UCS

ν	%correct	Rules	Time (s)
1	96.86	550	49.69
2	97.04	557	50.11
3	97.04	554	54.78
4	96.80	563	50.15
5	97.17	544	48.97
6	96.88	563	49.34
7	96.95	549	49.28
8	96.97	548	49.35
9	97.07	543	49.13
10	96.78	552	49.48

Table 6.30: 9-bit Position - XCS

ν	%correct	Rules	Time (s)
1	93.52	907	138.66
2	94.42	880	129.93
3	95.71	862	130.86
4	96.63	841	125.38
5	96.88	829	122.00
6	97.70	808	115.08
7	97.90	772	110.51
8	97.52	791	110.89
9	97.76	775	114.12
10	97.66	764	111.79

Table 6.31: 11-bit Position - UCS

As can be seen from the tables, for low-generality problems the optimal value of ν is closer to the lower range of the spectrum – best accuracy results are achieved when $\nu = 1$. Since high classifier error values are anticipated for such types of problems (low-generality or noisy), it is understandable that it is undesirable to dramatically drop the accuracy value if the error threshold is exceeded – this will have a negative impact on the overall and final accuracy of the algorithm. Setting ν low for low-generality problems is recommended for both XCS and UCS algorithms.

For high-generality problems, optimal values of ν for XCS and UCS differ. For XCS, optimal value appears to be present at value $\nu = 5$ – at this value, best balance between accuracy, rules generated and run time can be achieved. This is as well supported by experiments implementing population compression (Table 6.32). As can be seen, accuracy of the algorithm increases and reaches maximum at $\nu = 5$, and then proceeds to slowly decrease. These results complement the recommendations provided by [Urbanowicz and Browne, 2017], [Butz and Wilson, 2001] – in fact, most studies conducted rarely use any other value than $\nu = 5$ during their experiments. This confirms the practical correctness of the aforementioned conclusion.

ν	%correct	Rules	Time (s)
1	84.39	69	157.95
2	86.43	80	164.75
3	87.57	91	175.45
4	86.49	83	180.29
5	88.18	86	178.87
6	87.89	77	163.72
7	88.00	84	176.36
8	87.89	93	170.81
9	87.61	86	174.38
10	86.52	89	165.22

Table 6.32: 11-bit Multiplexer XCS Compacted

For UCS used on well-generalisable problems, results demonstrate that as the fitness exponent increases, algorithm accuracy increases, whilst the number of devised rules decreases even when rule compaction has not been turned on. This is especially dramatically visible when looking at the results obtained for the Position problem. Due to high-generalisable nature of the problems, it is admissible to allow the accuracy to drop rapidly if the error threshold is exceeded – as this will increase the rapidness with which inaccurate rules are separated from the accurate rules within the population. This conclusion is supported by suggestion to use $\nu = 10$ in [Orriols-Puig and Bernado-Mansilla, 2006]. To recapitulate, for UCS working on well-generalisable problems and data, the fitness exponent must be set high – to $\nu = \mathbf{10}$.

6.5 Parameter χ - Crossover probability

The crossover probability parameter (χ) determines the probability of applying crossover to some selected classifiers during the run of the genetic algorithm. Crossover is an operation intended to convergence the population to a local minimum or maximum in the landscape.

The following results (Table 6.33-6.38) demonstrate the relationship between the crossover probability, algorithm accuracy and rules generated:

χ	%correct	Rules	Time (s)
0.7	45.93	992	104.98
0.8	46.26	995	100.26
0.9	45.42	993	100.93

Table 6.33: 11-bit Parity - XCS

χ	%correct	Rules	Time (s)
0.7	46.14	996	201.15
0.8	46.03	994	198.80
0.9	45.66	995	191.56

Table 6.34: 11-bit Parity - UCS

χ	%correct	Rules	Time (s)
0.7	85.63	989	115.33
0.8	85.81	981	115.39
0.9	84.47	982	104.39

Table 6.35: 11-bit Multiplexer - XCS

χ	%correct	Rules	Time (s)
0.7	82.58	977	136.72
0.8	82.26	974	136.84
0.9	82.89	975	135.31

Table 6.36: 11-bit Multiplexer - UCS

χ	%correct	Rules	Time (s)
0.7	97.27	563	66.19
0.8	96.82	555	65.89
0.9	96.99	546	66.27

Table 6.37: 9-bit Position - XCS

χ	%correct	Rules	Time (s)
0.7	97.28	836	84.43
0.8	96.74	841	84.35
0.9	97.02	836	84.63

Table 6.38: 11-bit Position - UCS

As can be seen from the results, increasing the crossover probability from 0.7 to 0.9 yields very similar results, discrepancies between which can be due to the stochastic nature of the algorithm and other external factors. Essentially, a conclusion can be made that if rule compression is not used, any value in the range of **[0.7, 0.9]** will yield appropriate results.

The following results (Table 6.39-6.40) demonstrate the relationship between the crossover probability, algorithm accuracy and rules generated, after rule compression:

χ	%correct	Rules
0.7	88.53	84
0.8	87.24	79
0.9	85.5	77

Table 6.39: 11-bit Multiplexer - XCS

χ	%correct	Rules
0.7	71.88	86
0.8	70.90	83
0.9	64.50	60

Table 6.40: 11-bit Position - XCS

As can be seen for both Multiplexer and Position problems, increase in crossover probability results in decrease of the algorithm accuracy by a considerable amount - best results are achieved when $\chi = 0.7$. The number of rules is also decreasing – this can be explained by the fact that by increasing crossover probability the algorithm is encouraged to generate rules that will be more generalised, so more rules will be subsumed during the compression stage. The caveat is then that more general rules do not necessarily guarantee a better resulting algorithm accuracy. To conclude: if population compression is to be used, the recommended value for χ is **0.7**.

6.6 Parameter μ - Mutation probability

The mutation probability parameter ($\boldsymbol{\mu}$), in some sources also known as the mutation rate parameter, determines the probability of mutation for every attribute of a classifier generated by the genetic algorithm.

For this project, a range of values from [0.2, 0.5] was investigated for this parameter, as per recommendations in [Urbanowicz and Browne, 2017]. Although apparently not supported in practice, for the sake of consistency, the range recommended by [Urbanowicz and Browne, 2017] was investigated in more thorough detail.

The following results (Table 6.41-6.46) demonstrate the relationship between the mutation probability, algorithm accuracy, rules generated and time elapsed:

μ	%correct	Rules	Time (s)
0.2	47.31	949	72.20
0.3	47.00	980	74.24
0.4	46.01	994	73.70
0.5	45.77	996	76.89

Table 6.41: 11-bit Parity - XCS

μ	%correct	Rules	Time (s)
0.2	45.99	967	172.55
0.3	46.17	990	181.18
0.4	45.20	994	200.81
0.5	45.99	997	201.31

Table 6.42: 11-bit Parity - UCS

μ	%correct	Rules	Time (s)
0.2	99.14	732	62.79
0.3	92.24	943	69.72
0.4	85.81	978	71.57
0.5	81.35	992	79.01

Table 6.43: 11-bit Multiplexer - XCS

μ	%correct	Rules	Time (s)
0.2	99.51	723	141.22
0.3	94.19	902	181.50
0.4	82.07	980	191.51
0.5	76.42	993	200.94

Table 6.44: 11-bit Multiplexer - UCS

μ	%correct	Rules	Time (s)
0.2	97.7	446	54.74
0.3	97.48	493	60.26
0.4	97.27	576	66.25
0.5	96.23	649	70.71

Table 6.45: 9-bit Position - XCS

μ	%correct	Rules	Time (s)
0.2	98.82	783	76.17
0.3	98.20	813	79.31
0.4	96.95	834	83.57
0.5	94.09	872	93.70

Table 6.46: 11-bit Position - UCS

As can be seen, for all three problems for both UCS and XCS the optimal value is at 0.2; as the value of μ increases, the accuracy decreases, rule count becomes larger and the computation time increases as well.

For the 11-bit Multiplexer problem, both when utilising population compression and without it, the optimum value for the mutation probability appears to be 0.05, resulting in a prediction accuracy of 100%, what is much better than what was achieved when looking at the [0.2, 0.5] range. For the 9-bit Position problem, similar dynamic is present.

The results appear to be describing a relationship following the formula mentioned in [Deb, 2011]: mutation probability must be set as

$$\mu = \frac{1}{n}, \tag{6.1}$$

where n is the number of variables – in LCS, each state in a condition can be perceived as a variable.

Therefore, one can conclude that the mutation probability parameter is very domain specific, and no single value can be recommended; rather, mutation probability must be calculated using formula suggested in [Deb, 2011]. This agrees both with the results recorded, practical implementations of other researchers (e.g. [Karlsen and Moschoyiannis, 2018a]), and the available literature on the topic.

6.7 Other Parameters

This section will touch upon other parameters and values to which they must be set.

Fitness fall-off ($\boldsymbol{\alpha}$) – [Urbanowicz and Browne, 2017] states that parameter α must be set to **0.1** and should never be changed. In majority of literature and research, α is rarely different to the value above – complexity of calculating the classifiers fitness is minimised by keeping α constant, leaving ε_0 and ν parameters to be used for adjustment.

Learning rate ($\boldsymbol{\beta}$) – denotes the learning rate of the algorithm. Recall Section 3.4.1 for a description of how the learning rate is used within the LCS to update classifier parameters. This parameter is quite straight-forward to set and is dependent on the nature of the problem being solved; $\beta = \mathbf{0.1}$ must be used for problems where noise is present, data is unbalanced, complexity is high, whilst $\beta = \mathbf{0.2}$ must be used for clean straight-forward problems. Setting values on the lower scale will simply result in a slow convergence rate for the system. On the other hand, exceeding the suggested maximum value of **0.2** is uncommon, as high learning rates usually are an alarm that the problem being solved is too simple to be solved using LCS, so other techniques should be considered.

Accuracy threshold ($\boldsymbol{\epsilon_0}$) – also known in some literature as the *error threshold*, this value identifies the limit under which the classifier error ϵ must be for that classifier to be able to subsume other classifiers. A quick experiment was conducted to compare suggestion in [Butz and Wilson, 2001] (as the original and authoritative source for XCS) and suggestion for ϵ_0 in [Urbanowicz and Browne, 2017] to determine what value will be more suitable for XCS on 11-bit Multiplexer problem. The results can be seen below (Table 6.47), averaged over 5 runs each:

ϵ_0	%correct	Rules	Time (s)
0.01	99.17	76	87.22
10	99.12	82	83.68

Table 6.47: 11-bit Multiplexer - XCS

As can be seen, value suggested by [Urbanowicz and Browne, 2017] yields a marginally better accuracy (close to insignificant difference) in marginally slower computation time, but better (lower) rule count.

To conclude – ϵ_0 must be set in accordance with the problem being learned. For clean problems, ϵ_0 must be set low, in the range **[0, 0.01]** – for complex problems and problems that are expected to yield a high amount of error for each classifier, ϵ_0 must be set higher. On the other hand, if the parameters of the problems are not known before hand, it is best to utilise the suggestion given by [Butz and Wilson, 2001] as a starting point. Utilising an adaptive error threshold is not common and has not been observed in other works, so more research needs to be conducted to determine its effectiveness.

Chapter 7

Conclusion and Future Work

7.1 Optimal Parameters - Summary

This section will provide a condensed summary of recommendations for every investigated parameter optimal value.

Parameter I - Environment iterations

For problems where the training instance has a length of smaller or equal to 20 bits, acceptable results may be achieved with the number of iterations being set between **10,000-50,000** cycles. For larger and the more complex problems, a larger number of iterations must be considered and set.

Parameter N - Population size

Population size in the range of **[1000, 4000]** will yield adequate accuracy for majority of classification tasks, with smaller problems employing the lower values of that range and more large and complex problems employing the upper value of the range. If population compression is to be utilised, the population size will need to be set on the generous side of the proposed range.

Parameter $P\#$ - "Don't care" probability

For problems that are not easily accurately generalisable, P# must be set low – in the range of **[0, 0.3]**, with worst situations requiring P# to be set as low as **zero**. For problems that contain a cleanly generalisable pattern or where high generality is needed, P# should be set to value of the higher end of the range – from **[0.7, 0.99]**. Alternatively, in situations where determination of P# is too difficult or impossible, rule specificity limit (RSL) [Urbanowicz and Moore, 2015] may be utilised.

Parameter ν - Fitness exponent

Setting ν low $(\boldsymbol{\nu = 1})$ for low-generality problems is recommended for both XCS and UCS algorithms. For high-generality problems, optimal values of ν for XCS and UCS differ. For XCS, optimal value appears to be present at value $\nu = 5$, whilst for UCS working on well-generalisable problems and data, the fitness exponent must be set high – to $\boldsymbol{\nu = 10}$.

Parameter χ - Crossover probability

If rule compression is not used, any value in the range of **[0.7, 0.9]** will yield appropriate results; else, if population compression is to be used, the recommended value for χ is **0.7**.

Parameter μ - Mutation probability

The mutation probability parameter is very domain specific, and no single value can be recommended; rather, mutation probability must be calculated using formula suggested in [Deb, 2011]:

$$\mu = \frac{1}{n}, \tag{7.1}$$

where n is the number of variables – in LCS, each state in a condition can be perceived as a variable.

Parameter α - Fitness fall-off

Fitness fall-off in single-step problems must be set to **0.1**.

Parameter β - Learning rate

β = **0.1** must be used for problems where noise is present, data is unbalanced, complexity is high, whilst β = **0.2** must be used for clean, straight-forward problems.

Parameter ϵ_0 - Accuracy threshold

ϵ_0 must be set in accordance with the problem being learned. For clean problems, ϵ_0 must be set low, in the range **[0, 0.01]** – for complex problems and problems that are expected to yield a high amount of error for each classifier, ϵ_0 must be set higher.

7.2 Future Work

Future work in the area of LCS run parameters could include looking into automating the setting of run parameters in accordance with the problem that is being given to the LCS for learning. At least several run parameters, e.g. the error threshold, mutation probability, "don't care" probability may be set automatically upon the analysis of the given input problem - this will greatly simplify the process of specifying run parameters, narrowing the range of parameters that must be specified manually by the algorithm operator. An interesting result of such investigation could potentially be a new LCS variant that would be self-adapting in terms of run parameters to different types of learning problems.

7.3 Project Evaluation

This section will provide evaluation of the project, as well as how well the project achieved the initially set out objectives.

7.3.1 Overall Project Evaluation

This project has achieved all the objectives that have been set out in Section 1.3 and therefore it can be concluded that this project has been a success:

1. The project objective to provide a comprehensive overview of how Michigan-style Learning Classifier Systems operate has been achieved with this report, with Chapter 3 providing an in-depth overview of Learning Classifier Systems.

2. The project objective to provide an overview of the importance of setting the correct LCS run parameters has also been achieved using this report. Specifically Chapter 4 provides an in-depth overview and background research of the Learning Classifier Systems run parameters, as well as the result analysis puts the values for every run parameter into the context of the overall problem, focusing on explaining how the parameter value corresponds to the problem that is being solved.

3. The project objective to provide a software implementation of XCS and UCS algorithms in Python has been successfully fulfilled via the source code that is accompanying this report. Both UCS and XCS implementations have been successfully devised and utilised to gather data for analysis.

4. The main project objective to provide a comprehensive analysis of the acquired results to determine the optimal run parameter values has been successfully fulfilled in Chapter 6 of this report. A final overview of the optimal parameter values has been provided in Section 7.1. Effect of run parameter values on the resulting model prediction capability, size and algorithm run time has been recorded and analysed in Chapter 6.

7.3.2 Implementation Evaluation

Implementation Strengths

Implementation strengths include:

1. *Simple implementation* - the implementation of both UCS and XCS algorithms is easy to follow and understand, making it easy both for the final users and for developers who would want to use/debug/extend the source code.

2. *Simple output interpretability* - log and console output allows to easily see what the system is calculating and how it is progressing. The output is clear and allows to visualise the work of every component of the LCS.

3. *Model K-Fold Cross-Validation* - k-fold Cross-Validation has been incorporated to evaluate the resulting LCS model predictive capabilities.

4. *Wide adaptability* - the solution can be used successfully to work with a wide range of data, from discrete to continuous, clean to noisy, complete and incomplete.

5. *Benchmark Problem Generation* - scripts have been created to effortlessly generate training data for solving the most commonly used benchmark problems in the LCS industry.

6. *Using a configuration file to supply run parameters* - supplying run parameters is easy and intuitive, can be done simply through the global configuration file without any need to access and change hard-coded run parameter values directly in the source code.

7. *XCS Support* - finally, support for the XCS LCS algorithm variation has been added, extending the scope for usage of the original implementation. Being built on the basis of the UCS implementation, the XCS implementation benefits from all the positives that the UCS implementation originally had.

Implementation Weaknesses

Although the original implementation weaknesses laid out in Section 5.2 have been rectified (XCS implementation has been created, model validation through K-Fold Cross-Validation has been introduced), the primary weakness of the final implementation is its speed - calculations are carried out on the CPU, what might take a significant amount of time, especially for larger and more complex problems. This is not helped by the fact that Python by itself is a slow language, being the slowest among other possible implementation languages, such as C/C++ and Java. As part of future work, the implementation could be enhanced by having the calculations be done on the GPU rather than on the CPU (for example, using pyCUDA[1] or Numba[2]), or have the parallelism be improved within the implementation using the "multiprocessing"[3] process-based threading interface. Such techniques should significantly decrease the model training time.

7.3.3 Confidence in Results

Several factors allow to conclude that the results gained can be trusted. Firstly, the carried out experiments were conducted with utmost care for experimental fairness - load on both systems was kept minimal during the experiments, results for a single benchmark were gained on only one machine, and not both (i.e. Desktop-PC was used for XCS 11-bit Multiplexer population size, whilst Dell was used for UCS 11-bit Multiplexer population size, but not both systems on one benchmark). For each different value for an LCS parameter being changed (every row in the tables in Chapter 6), the experiment was run 10 times in order to compromise with the stochastic nature of LCS and minimise the chance of a fluke - around 4000 combined XCS and UCS runs were required to gather the data presented in Chapter 6. Finally, confidence in results stems from the fact that the results confirm

[1] `https://developer.nvidia.com/pycuda`
[2] `http://numba.pydata.org/`
[3] `https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing`

the general suggestions that can be found in literature - although such suggestions were not taken into account during experimentation in order to avoid involuntary bias, it is evident that the results support the existing theory.

7.3.4 Personal Evaluation

From a personal point of view, this has been a project that I have immensely benefited from whilst I was working on it during my Final Year. This has been my first in-depth exposure to genetic algorithms and machine learning as a discipline outside of the taught University programme, and this was my first encounter with Learning Classifier Systems in general - due to this, I had to learn a lot during this project within both disciplines in order to succeed in achieving the project goals. I had to develop, and have developed, my research skills greatly, becoming more accustomed to finding and reading scientific papers/literature, extracting the useful information and following references to other researchers in order to find the answers and knowledge that I required in order to fulfil the project goals - the bibliography accompanying this report is evidence of the amount of papers, website pages, books, code bases, and articles I had to read and comprehend in order to become knowledgeable in the topic of Learning Classifier Systems.

This has also been a major project for me in terms of the amount of code needed to be developed in order to gather results - this has arguably been the largest Python project that I have yet developed to date. Working on this code base has helped me improve my programming skills with Python, as well as allowed me to learn working in a new development environment - PyCharm. I am content with my effort, considering that the project goals have been met and that the result of my work is an LCS suite capable of running both UCS and XCS LCS algorithms.

Despite being warned that this is a difficult project to undertake, I am content with how I was able to handle the project, organising my time and effort effectively - I was able to dedicate sufficient time for research in order to gain the understanding of how Learning Classifier Systems work and how they operate, what role the run parameters play in the LCS algorithm and what effect they have on the overall system performance; then, I dedicated sufficient time to implement the XCS LCS variation, making an educated decision on what code base to use as the implementation foundation and then spending enough time to write a piece of software capable in achieving the goal of allowing me to gather data for analysis; I have then dedicated sufficient time for the analysis of acquired data, and finally, sufficient time to write this report describing and explaining the results that I have acquired and the conclusions that can be drawn from them. During the entire process, I learned something new every day and developed both as a software developer, as a researcher and as a IT-industry professional.

If I was to start my project again, or if I had an extra month, I would dedicate more time on improving my implementation speed - integrating calculations on GPU capability would be a good start, and would save me and future users of my implementation a great amount of time when training on large and complex data sets.

7.3.5 Final Statement

Overall, this project has been a success and was able to provide useful results. The result of this project has been the creation of guidelines, explaining how to achieve the optimal value of LCS run parameters given the type and complexity of the problem being solved, as well as a software suite implemented in Python containing source code for running UCS and XCS LCS algorithms on various types of data and providing information on the predictive capability of the resulting model. The result of this project has also been this report that provides a general overview of Learning Classifier Systems and may serve as an introductory reading material to anyone interested in finding more about Learning Classifier Systems and their place in the modern world.

Appendix A

Generating Data for Benchmarks Scripts

This appendix contains the scripts to generate data for Parity, Multiplexer, Decoder and Position benchmarks.

```
#set condition length
bits = 11
def generate_complete_parity_data(myfile, bits):
    """ Attempts to generate a complete non-redundant parity dataset."""
    num_bits = sanity_check(bits)
    if num_bits is None:
        print 'Problem_Parity: ERROR - Specified binary string bits is smaller than
     or 0'
    else:
        try:
            fp = open('Demo_Datasets/' + myfile, 'w')

            # Make File Header
            for i in range(num_bits):
                # Bits
                fp.write('B_' + str(i) + '\t')
            # Class
            fp.write('Class' + '\n')

            # Generating all range of bits
            for i in range(2 ** num_bits):
                binary_str = bin(i)
                string_array = binary_str.split('b')
                binary = string_array[1]

                while len(binary) < num_bits:
                    binary = '0' + binary

                # Starting counter of "1" bits
                one_count = 0
                for j in binary:
                    if int(j) == 1:
                        one_count += 1

                # Determining instance class
                if one_count % 2 == 0:
                    output = 0
                else:
                    output = 1

                for j in range(num_bits):
                    fp.write(binary[j] + '\t')

                # Write output to file
                fp.write(str(output) + '\n')
            fp.close()
            print 'Problem_Parity: Dataset Generation Complete'
        except:
```

```
            print 'ERROR - Cannot generate data for binary problem'

#sanity check condition length input
def sanity_check(bits):
    if bits > 0:
        return bits
    return None

#randomize generated data set
def randomize():
    with open("../XCS/Demo_Datasets/"+str(bits)+"Parity_Data_Complete.txt", 'r') as
     source:
        data = [(random.random(), line) for line in source]
    data[1:] = sorted(data[1:])
    with open("../XCS/Demo_Datasets/"+str(bits)+"Parity_Data_Complete_Randomized.
    txt", 'w') as target:
        for _, line in data:
            target.write(line)

if __name__ == '__main__':
    generate_complete_parity_data(str(bits)+"Parity_Data_Complete.txt", bits)
    randomize()
```

Listing A.1: Code snippet to generate a complete 11-bit Parity data set

```
bits = 11
instances = 10


def randomize():
    with open("../UCS/Demo_Datasets/" + str(bits) + "Multiplexer_Data_Complete.txt"
    , 'r') as source:
        data = [(random.random(), line) for line in source]
    data[1:] = sorted(data[1:])
    with open("../UCS/Demo_Datasets/" + str(bits) + "
    Multiplexer_Data_Complete_Randomized.txt", 'w') as target:
        for _, line in data:
            target.write(line)


def generate_multiplexer_data(myfile, num_bits, instances):
    """ """
    print("Problem_Multiplexer: Generate multiplexer dataset with " + str(instances
    ) + " instances.")
    first = solve_equation(num_bits)
    if first is None:
        print("Problem_Multiplexer: ERROR - The multiplexer takes # of bits as
    3,6,11,20,37,70,135,264")

    else:
        fp = open("Demo_Datasets/" + myfile, "w")
        # Make File Header
        for i in range(first):
            fp.write('A_' + str(i) + "\t")  # Address Bits

        for i in range(num_bits - first):
            fp.write('R_' + str(i) + "\t")  # Register Bits
        fp.write("Class" + "\n")  # State found at Register Bit

        for i in range(instances):
            instance = generate_multiplexer_instance(num_bits)
            for j in instance[0]:
                fp.write(str(j) + "\t")
            fp.write(str(instance[1]) + "\n")


def generate_multiplexer_instance(num_bits):
    """ """
    first = solve_equation(num_bits)
    if first is None:
        print("Problem_Multiplexer: ERROR - The multiplexer takes # of bits as
    3,6,11,20,37,70,135,264")

```

```
    else:
        condition = []
        # Generate random boolean string
        for i in range(num_bits):
            condition.append(str(random.randint(0, 1)))

        gates = ""

        for j in range(first):
            gates += condition[j]

        gates_decimal = int(gates, 2)
        output = condition[first + gates_decimal]

        return [condition, output]


def generate_complete_multiplexer_data(myfile, num_bits):
    """ Attempts to generate a complete non-redundant multiplexer dataset.  Ability to generate the entire dataset is computationally limited.
    We had success generating up to the complete 20-multiplexer dataset"""

    print("Problem_Multiplexer: Attempting to generate multiplexer dataset")
    first = solve_equation(num_bits)

    if first is None:
        print("Problem_Multiplexer: ERROR - The multiplexer takes # of bits as 3,6,11,20,37,70,135,264")
    else:
        try:
            fp = open("Demo_Datasets/" + myfile, "w")
            # Make File Header
            for i in range(first):
                fp.write('A_' + str(i) + "\t")  # Address Bits

            for i in range(num_bits - first):
                fp.write('R_' + str(i) + "\t")  # Register Bits
            fp.write("Class" + "\n")  # State found at Register Bit

            for i in range(2 ** num_bits):
                binary_str = bin(i)
                string_array = binary_str.split('b')
                binary = string_array[1]

                while len(binary) < num_bits:
                    binary = "0" + binary

                gates = ""
                for j in range(first):
                    gates += binary[j]

                gates_decimal = int(gates, 2)
                output = binary[first + gates_decimal]

                # fp.write(str(i)+"\t")
                for j in binary:
                    fp.write(j + "\t")
                fp.write(output + "\n")

            fp.close()
            print("Problem_Multiplexer: Dataset Generation Complete")

        except:
            print(
                "ERROR - Cannot generate all data instances for specified multiplexer due to computational limitations")


def solve_equation(num_bits):
    for i in range(1000):
        if i + 2 ** i == num_bits:
            return i
    return None
```

```
114
115
116 if __name__ == '__main__':
117
118     generate_complete_multiplexer_data(str(bits) + "Multiplexer_Data_Complete.txt",
        bits)  # 3,6,11,20,37
119     randomize()
120     # generate_multiplexer_data("Multiplexer_Data.txt", bits, instances)
```

Listing A.2: Code snippet to generate a complete 11-bit Multiplexer data set

```
1  bits = 11
2  instances = 10
3
4
5  def randomize():
6      with open("../UCS/Demo_Datasets/" + str(bits) + "Decoder_Data_Complete.txt", 'r
       ') as source:
7          data = [(random.random(), line) for line in source]
8      data[1:] = sorted(data[1:])
9      with open("../UCS/Demo_Datasets/" + str(bits) + "
       Decoder_Data_Complete_Randomized.txt", 'w') as target:
10         for _, line in data:
11             target.write(line)
12
13
14 def generate_decoder_data(myfile, bits, instances):
15     """ """
16     print("Problem_Decoder: Attempting to Generate decoder dataset with " + str(
       instances) + " instances.")
17     num_bits = sanity_check(bits)
18
19     if num_bits is None:
20         print("Problem_Decoder: ERROR - Specified binary string bits is smaller
       than or 0")
21     else:
22         fp = open("Demo_Datasets/" + myfile, "w")
23         # Make File Header
24         for i in range(num_bits):
25             fp.write('B_' + str(i) + "\t")  # Bits
26         fp.write("Class" + "\n")  # Class
27
28         for i in range(instances):
29             instance = generate_decoder_instance(bits)
30             for j in instance[0]:
31                 fp.write(str(j) + "\t")
32             fp.write(str(instance[1]) + "\n")
33
34         fp.close()
35         print("Problem_Decoder: File Generated")
36
37
38 def generate_decoder_instance(bits):
39     """ """
40     num_bits = sanity_check(bits)
41     if num_bits is None:
42         print("Problem_Decoder: ERROR - Specified binary string size is smaller
       than or 0")
43     else:
44         condition = ""
45
46         # Generate random boolean string
47         for i in range(bits):
48             condition = condition + (str(random.randint(0, 1)))
49
50         output = int(condition, base=2)
51
52         return [condition, output]
53
54
55 def generate_complete_decoder_data(myfile, bits):
56     """ Attempts to generate a complete non-redundant decoder dataset."""
57
```

```
    print("Problem_Decoder: Attempting to generate complete decoder dataset")
    num_bits = sanity_check(bits)

    if num_bits is None:
        print("Problem_Decoder: ERROR - Specified binary string bits is smaller than or 0")
    else:
        try:
            fp = open("Demo_Datasets/" + myfile, "w")
            # Make File Header
            for i in range(num_bits):
                fp.write('B_' + str(i) + "\t")  # Bits
            fp.write("Class" + "\n")  # Class

            for i in range(2 ** num_bits):
                binary_str = bin(i)
                string_array = binary_str.split('b')
                binary = string_array[1]

                while len(binary) < num_bits:
                    binary = "0" + binary

                output = int(binary, base=2)

                for j in range(num_bits):
                    fp.write(binary[j] + "\t")

                fp.write(str(output) + "\n")

            fp.close()
            print("Problem_Decoder: Dataset Generation Complete")

        except:
            print(
                "ERROR - Cannot generate all data instances for specified binary due to computational limitations")


def sanity_check(input_bits):
    if input_bits > 0:
        return input_bits
    return None


if __name__ == '__main__':
    # generate_decoder_data(str(bits)+"-"+str(instances)+"Decoder_Data.txt", bits, instances)
    generate_complete_decoder_data(str(bits) + "Decoder_Data_Complete.txt", bits)
    randomize()
```

Listing A.3: Code snippet to generate a complete 11-bit Decoder data set

```
bits = 9
instances = 10


def randomize():
    with open("../UCS/Demo_Datasets/" + str(bits) + "Position_Data_Complete.txt", 'r') as source:
        data = [(random.random(), line) for line in source]
    data[1:] = sorted(data[1:])
    with open("../UCS/Demo_Datasets/" + str(bits) + "Position_Data_Complete_Randomized.txt", 'w') as target:
        for _, line in data:
            target.write(line)


def generate_position_data(myfile, bits, instances):
    """ """
    print("Problem_Position: Attempting to Generate position dataset with " + str(instances) + " instances.")
    num_bits = sanity_check(bits)

```

```
    if num_bits is None:
        print("Problem_Position: ERROR - Specified binary string bits is smaller than or 0")
    else:
        fp = open("Demo_Datasets/" + myfile, "w")
        # Make File Header
        for i in range(num_bits):
            fp.write('B_' + str(i) + "\t")  # Bits
        fp.write("Class" + "\n")  # Class

        for i in range(instances):
            instance = generate_position_instance(bits)
            for j in instance[0]:
                fp.write(str(j) + "\t")
            fp.write(str(instance[1]) + "\n")

        fp.close()
        print("Problem_Position: File Generated")


def generate_position_instance(bits):
    """ """
    num_bits = sanity_check(bits)
    if num_bits is None:
        print("Problem_Position: ERROR - Specified binary string size is smaller than or 0")
    else:
        condition = []
        position = 0

        # Generate random boolean string
        for i in range(bits):
            condition.append(str(random.randint(0, 1)))

        for j in range(len(condition)):
            if int(condition[j]) == 1:
                position = j
                break

        output = position

        return [condition, output]


def generate_complete_position_data(myfile, bits):
    """ Attempts to generate a complete non-redundant position dataset."""

    print("Problem_Position: Attempting to generate complete position dataset")
    num_bits = sanity_check(bits)

    if num_bits is None:
        print("Problem_Position: ERROR - Specified binary string bits is smaller than or 0")
    else:
        try:
            fp = open("Demo_Datasets/" + myfile, "w")
            # Make File Header
            for i in range(num_bits):
                fp.write('B_' + str(i) + "\t")  # Bits
            fp.write("Class" + "\n")  # Class

            for i in range(2 ** num_bits):
                binary_str = bin(i)
                string_array = binary_str.split('b')
                binary = string_array[1]

                while len(binary) < num_bits:
                    binary = "0" + binary

                bin_list = list(binary)
                output = 0

                for j in range(len(bin_list)):
```

```
                    if int(bin_list[j]) == 1:
                        output = j
                        break

                for j in range(num_bits):
                    fp.write(binary[j] + "\t")

                fp.write(str(output) + "\n")

            fp.close()
            print("Problem_Position: Dataset Generation Complete")

        except:
            print(
                "ERROR - Cannot generate all data instances for specified binary due to computational limitations")


def sanity_check(bits):
    if bits > 0:
        return bits
    return None


if __name__ == '__main__':
    # generate_position_data(str(bits)+"-"+str(instances)+"Position_Data.txt", bits, instances)
    generate_complete_position_data(str(bits) + "Position_Data_Complete.txt", bits)
    randomize()
```

Listing A.4: Code snippet to generate a complete 9-bit Position data set

Bibliography

[Arthur, 1994] Arthur, W. B. (1994). Inductive reasoning and bounded rationality. *The American Economic Review*, 84(2):406–411.

[Bacardit, 2004] Bacardit, J. (2004). *Pittsburgh genetics-based machine learning in the data mining era: representations, generalization, and run-time*. PhD thesis, Ramon Llull University, Barcelona, Catalonia, Spain.

[Bacardit et al., 2009] Bacardit, J., Burke, E. K., and Krasnogor, N. (2009). Improving the scalability of rule-based evolutionary learning. *Memetic Computing*, 1(1):55–67.

[Barry et al., 2004] Barry, A., Holmes, J., and Llorà, X. (2004). Data mining using learning classifier systems. In Bull, L., editor, *Applications of Learning Classifier Systems*, pages 15–67, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bernadó-Mansilla and Garrell-Guiu, 2003] Bernadó-Mansilla, E. and Garrell-Guiu, J. M. (2003). Accuracy-based learning classifier systems: Models, analysis and applications to classification tasks. *Evolutionary Computation*, 11(3):209–238.

[Bonarini and Basso, 1997] Bonarini, A. and Basso, F. (1997). Learning to compose fuzzy behaviors for autonomous agents. *International Journal of Approximate Reasoning*, 17(4):409 – 432.

[Booker et al., 1989] Booker, L., Goldberg, D., and Holland, J. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1):235 – 282.

[Broome, 2004] Broome, J. (2004). Half-life bot development. `http://hpb-bot.bots-united.com/hpb_bot.html`. Last Accessed: 7 April 2019.

[Browne, 2004] Browne, W. N. L. (2004). The development of an industrial learning classifier system for data-mining in a steel hop strip mill. In Bull, L., editor, *Applications of Learning Classifier Systems*, pages 223–259, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bull, 2015] Bull, L. (2015). A brief history of learning classifier systems: from cs-1 to xcs and its variants. In *Evolutionary Intelligence*, volume 8, page 55–70.

[Bull and O'Hara, 2002] Bull, L. and O'Hara, T. (2002). Accuracy-based neuro and neuro-fuzzy classifier systems. In *GECCO'02 Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 905–911, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Bunch et al., 1982] Bunch, J. B., Miller, R. D., and Wheeler, J. E. (1982). Distribution system integrated voltage and reactive power control. *IEEE Transactions on Power Apparatus and Systems*, PAS-101(2):284–289.

[Butz and Herbort, 2008] Butz, M. and Herbort, O. (2008). Context-dependent predictions and cognitive arm control with xcsf. *GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation 2008*, pages 1357–1364.

[Butz, 2004] Butz, M. V. (2004). *Rule-based evolutionary online learning systems: learning bounds, classification, and prediction.* University of Illinois, USA.

[Butz, 2015] Butz, M. V. (2015). *Learning Classifier Systems*, pages 961–981. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Butz et al., 2005] Butz, M. V., Goldberg, D. E., and Lanzi, P. L. (2005). Computational complexity of the xcs classifier system. In Bull, L. and Kovacs, T., editors, *Foundations of Learning Classifier Systems*, pages 91–125, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Butz et al., 2001] Butz, M. V., Kovacs, T., Lanzi, P. L., and Wilson, S. W. (2001). How xcs evolves accurate classifiers. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 927–934, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Butz and Wilson, 2001] Butz, M. V. and Wilson, S. W. (2001). An algorithmic description of xcs. *Soft Computing*, 6(3):144–153.

[Casillas et al., 2007] Casillas, J., Carse, B., and Bull, L. (2007). Fuzzy-xcs: A michigan genetic fuzzy system. *IEEE Transactions on Fuzzy Systems*, 15(4):536–550.

[Cavellucci and Lyra, 1997] Cavellucci, C. and Lyra, C. (1997). Minimization of energy losses in electric power distribution systems by intelligent search strategies. *International Transactions in Operational Research*, 4(1):23 – 33.

[Challet and Zhang, 1997] Challet, D. and Zhang, Y. C. (1997). Emergence of cooperation and organization in an evolutionary game. *Physica A: Statistical Mechanics and its Applications*, 246(3-4):407–418.

[Dam et al., 2008] Dam, H. H., Abbass, H. A., Lokan, C., and Yao, X. (2008). Neural-based learning classifier systems. *IEEE Transactions on Knowledge and Data Engineering*, 20(1):26–39.

[Dean, 2013] Dean, T. (2013). *Network+ Guide to Networks.* Course Technology Cengage Learning.

[Deb, 2011] Deb, K. (2011). Multi-objective optimisation using evolutionary algorithms: An introduction. In Wang, L., Ng, A. H. C., and Deb, K., editors, *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing*, pages 3–34, London. Springer London.

[Fogarty, 1988] Fogarty, T. (1988). Adapting to noise. *IFAC Proceedings Volumes*, 21(13):79 – 84. IFAC Workshop on Artificial Intelligence in Real-Time Control, Clyne Castle, Swansea, UK, 21-23 September.

[Fukuyama and Holland, 1996] Fukuyama, F. and Holland, J. H. (1996). Hidden order: How adaptation builds complexity. *Foreign Affairs*, 75(4):137.

[Goldberg, 1985] Goldberg, D. E. (1985). Genetic algorithms and rules learning in dynamic system control. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 8–15, Hillsdale, NJ, USA. L. Erlbaum Associates Inc.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing, Boston, MA, USA.

[Goldberg et al., 1992] Goldberg, D. E., Horn, J., and Deb, K. (1992). What makes a problem hard for a classifier system? *Collected Abstracts for the First International Workshop on Learning Classifier Systems (IWLCS92)*.

[Hercog and Fogarty, 2000] Hercog, L. and Fogarty, T. (2000). Xcs-based inductive intelligent multiagent system. *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 125–132.

[Hercog, 2003] Hercog, L. M. (2003). *Evolutionary and conventional reinforcement learning in multi agent systems for social simulation*. South Bank University, London, UK.

[Hercog, 2004a] Hercog, L. M. (2004a). Co-evolutionary agent self-organization for city traffic congestion modeling. *Genetic and Evolutionary Computation – GECCO 2004 Lecture Notes in Computer Science*, page 993–1004.

[Hercog, 2004b] Hercog, L. M. (2004b). Traffic balance using classifier systems in an agent based simulation. In Bull, L., editor, *Applications of Learning Classifier Systems*, pages 143–166, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

[Holland, 1992] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

[Holland and Reitman, 1977] Holland, J. H. and Reitman, J. S. (1977). Cognitive systems based on adaptive algorithms. *ACM SIGART Bulletin*, (63):49.

[Holmes, 1998] Holmes, J. (1998). Discovering risk of disease with a learning classifier system. In *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA97)*, pages 426–433. Morgan Kaufmann, Burlington, MA, USA.

[Holmes et al., 2000] Holmes, J. H., Durbin, D. R., and Winston, F. K. (2000). The learning classifier system: an evolutionary computation approach to knowledge discovery in epidemiologic surveillance. *Artificial Intelligence in Medicine*, 19(1):53–74.

[Holmes and Sager, 2005] Holmes, J. H. and Sager, J. A. (2005). Rule discovery in epidemiologic surveillance data using epixcs: An evolutionary computation approach. *Artificial Intelligence in Medicine*, pages 444–452.

[Hosford, 2015] Hosford, A. (2015). Python software foundation - xcs 1.0.0. `https://pypi.org/project/xcs/`. Last Accessed: 8 April 2019.

[Hurst and Bull, 2003] Hurst, J. and Bull, L. (2003). A neural learning classifier system with self-adaptive constructivism for mobile robot control. *Artificial life*, 12:353–80.

[Iqbal et al., 2013] Iqbal, M., Browne, W. N., and Zhang, M. (2013). Extending learning classifier system with cyclic graphs for scalability on complex, large-scale boolean problems. *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO 13*.

[James et al., 2015] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2015). *An introduction to statistical learning: with applications in R*. Springer, New York, NY, USA.

[Karlsen, 2018] Karlsen, M. R. (2018). Multiple lcs variants. `https://gitlab.eps.surrey.ac.uk/RBMLNC/multiple-lcs-variants`. Last Accessed: 23 December 2018.

[Karlsen and Moschoyiannis, 2018a] Karlsen, M. R. and Moschoyiannis, S. (2018a). Evolution of control with learning classifier systems. *Applied Network Science*, 3(1):30.

[Karlsen and Moschoyiannis, 2018b] Karlsen, M. R. and Moschoyiannis, S. (2018b). Learning condition–action rules for personalised journey recommendations. In Benzmüller, C., Ricca, F., Parent, X., and Roman, D., editors, *Rules and Reasoning*, pages 293–301, Cham, Switzerland. Springer International Publishing.

[Kop et al., 2015] Kop, R., Toubman, A., Hoogendoorn, M., and Roessingh, J. J. (2015). Evolutionary dynamic scripting: Adaptation of expert rule bases for serious games. In Ali, M., Kwon, Y. S., Lee, C.-H., Kim, J., and Kim, Y., editors, *Current Approaches in Applied Artificial Intelligence*, pages 53–62, Cham, Switzerland. Springer International Publishing.

[Kovacs and Kerber, 2001] Kovacs, T. and Kerber, M. (2001). What makes a problem hard for xcs? In Luca Lanzi, P., Stolzmann, W., and Wilson, S. W., editors, *Advances in Learning Classifier Systems*, pages 80–99, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Koza, 1991] Koza, J. R. (1991). A hierarchical approach to learning the boolean multiplexer function. In *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, pages 171–192. Morgan Kaufmann, San Francisco, CA, USA.

[Kuhn and Johnson, 2016] Kuhn, M. and Johnson, K. (2016). *Applied predictive modeling*. Springer, New York, NY, USA.

[Lanzi, 2009] Lanzi, P. L. (2009). The xcs library. `http://xcslib.sourceforge.net/`. Last Accessed: 19 January 2019.

[Lukins, 2013] Lukins, T. C. (2013). Python learning classifier systems. `https://github.com/timlukins/pylcs`. Last Accessed: 10 March 2019.

[Mac Namee and Cunningham, 2001] Mac Namee, B. and Cunningham, P. (2001). A proposal for an agent architecture for proactive persistent non player characters.

[Nash, 1950] Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49.

[Orriols-Puig and Bernado-Mansilla, 2006] Orriols-Puig, A. and Bernado-Mansilla, E. (2006). A further look at ucs classifier system. *GECCO '06 Proceedings of the 8th annual conference on Genetic and evolutionary computation*.

[Orriols-Puig and Bernadó-Mansilla, 2008] Orriols-Puig, A. and Bernadó-Mansilla, E. (2008). Revisiting ucs: Description, fitness sharing, and comparison with xcs. In Bacardit, J., Bernadó-Mansilla, E., Butz, M. V., Kovacs, T., Llorà, X., and Takadama, K., editors, *Learning Classifier Systems*, pages 96–116, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Orriols-Puig et al., 2009] Orriols-Puig, A., Casillas, J., and Bernado-Mansilla, E. (2009). Fuzzy-ucs: A michigan-style learning fuzzy-classifier system for supervised learning. *IEEE Transactions on Evolutionary Computation*, 13(2):260–283.

[Quinlan, 1998] Quinlan, R. (1998). Data mining tools see5 and c5.0. `https://www.rulequest.com/see5-info.html`. Last Accessed: 19 March 2019.

[Robert and Guillot, 2003] Robert, G. and Guillot, A. (2003). MHiCS, A Modular And Hierarchical Classifier Systems Architecture For Bots. In *4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)*, pages 140–144, London, United Kingdom.

[Robert and Guillot, 2006] Robert, G. and Guillot, A. (2006). A motivational architecture of action selection for non-player characters in dynamic environments. *International Journal of Intelligent Games and Simulation*, 4(1):5–16.

[Robert et al., 2002] Robert, G., Portier, P., and Guillot, A. (2002). Classifier systems as 'Animat' architectures for action selection in MMORPG. In *Game-on 2002*, pages 121–125, Harrow, United Kingdom.

[Sanchez et al., 2006] Sanchez, S., Luga, H., and Duthen, Y. (2006). Learning classifier systems and behavioural animation of virtual characters. In Gratch, J., Young, M., Aylett, R., Ballin, D., and Olivier, P., editors, *Intelligent Virtual Agents*, pages 467–467, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Schuurmans and Schaeffer, 1989] Schuurmans, D. and Schaeffer, J. (1989). Representational difficulties with classifier systems. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 328–333, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Shafi and Abbass, 2017] Shafi, K. and Abbass, H. (2017). A survey of learning classifier systems in games. *IEEE Computational Intelligence Magazine*, 12:42–55.

[Small and Congdon, 2009] Small, R. and Congdon, C. (2009). Agent smith: Towards an evolutionary rule-based agent for interactive dynamic games. *IEEE Congress on Evolutionary Computation*, pages 660–666.

[Smith et al., 2000] Smith, R. E., Dike, B. A., Ravichandran, B., El-Fallah, A., and Mehra, R. K. (2000). The fighter aircraft lcs: A case of different lcs goals and techniques. In Lanzi, P. L., Stolzmann, W., and Wilson, S. W., editors, *Learning Classifier Systems*, pages 283–300, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Smith et al., 2004] Smith, R. E., El-Fallah, A., Ravichandran, B., Mehra, R. K., and Dike, B. A. (2004). The fighter aircraft lcs: A real-world, machine innovation application. In Bull, L., editor, *Applications of Learning Classifier Systems*, pages 113–142, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Smith, 1980] Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, Pittsburgh, PA, USA.

[Tamee et al., 2007] Tamee, K., Bull, L., and Pinngern, O. (2007). Towards clustering with xcs. *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO 07*.

[Urbanowicz, 2010] Urbanowicz, R. J. (2010). Software. `http://ryanurbanowicz.com/index.php/about/software/`. Last Accessed: 2 April 2019.

[Urbanowicz, 2014] Urbanowicz, R. J. (2014). elcs - educational learning classifier system. `https://sourceforge.net/projects/educationallcs/`. Last Accessed: 13 February 2019.

[Urbanowicz, 2015] Urbanowicz, R. J. (2015). Multiplexer problem. `https://ryanurbanowicz.com/index.php/resources-2/multiplexer-problem/`.

[Urbanowicz and Browne, 2017] Urbanowicz, R. J. and Browne, W. N. (2017). *Introduction to learning classifier systems*. Berlin, Germany. Springer.

[Urbanowicz and Moore, 2009] Urbanowicz, R. J. and Moore, J. H. (2009). Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, 2009.

[Urbanowicz and Moore, 2015] Urbanowicz, R. J. and Moore, J. H. (2015). Exstracs 2.0: description and evaluation of a scalable learning classifier system. *Evolutionary Intelligence*, 8(2-3):89–116.

[Vargas et al., 2004] Vargas, P. A., Filho, C. L., and Von Zuben, F. J. (2004). Application of learning classifier systems to the on-line reconfiguration of electric power distribution networks. In Bull, L., editor, *Applications of Learning Classifier Systems*, pages 260–275, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

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

[Wilson, 1998] Wilson, S. W. (1998). Generalization in the xcs classifier system.

[Wilson, 2000] Wilson, S. W. (2000). Get real! xcs with continuous-valued inputs. In Lanzi, P. L., Stolzmann, W., and Wilson, S. W., editors, *Learning Classifier Systems*, pages 209–219, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Wilson, 2001] Wilson, S. W. (2001). Mining oblique data with xcs. In Luca Lanzi, P., Stolzmann, W., and Wilson, S. W., editors, *Advances in Learning Classifier Systems*, pages 158–174, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Wilson, 2002] Wilson, S. W. (2002). Classifiers that approximate functions. *Natural Computing*, 1(2):211–234.

[Wilson and Goldberg, 1989] Wilson, S. W. and Goldberg, D. E. (1989). A critical review of classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 244–255, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Zulkifli, 2018] Zulkifli, H. (2018). Understanding learning rates and how it improves performance in deep learning. `https://bit.ly/2s4UAuK`. Last Accessed: 10 April 2019.