Chapter 2
LCS Concepts

Abstract

This chapter aims to provide an appreciation for core concepts that separate LCSs from other techniques. In particular, we provide insight into why they work, and how they are conceptually unique. It is hoped that the reader will appreciate that LCSs represent a machine learning concept, rather than a single technique. We consider the answers to questions such as (1) Rules/classifiers - What are they? Plus, how can they be represented and evaluated? (2) Why do we evolve a population of rules rather than a single rule as a solution? (3) What is the importance of cooperation and competition among classifiers? (4) How does an LCS interact with problems to find and generalise useful patterns? (5) What problem properties should be considered when deciding whether to apply an LCS? and (6) What are the general advantages and disadvantages of LCSs? The functional cycle and how to begin implementing an LCS are covered in the next chapter.

LCSs are a wondrous way to address interesting problems. LCSs are computer programs that attempt to build a piecewise model capturing patterns inherent in the data they experience. The concept of LCSs started to form in the mid-1970s with John Holland's work on adaptation. The first implementation, 'CS-1' by Holland and Reitman in 1978, was very different to to modern LCS algorithms. Note that CS-1 stands for 'Cognitive System One', rather than Classifier System One, which hints at the original purpose of LCSs.

LCSs are one of the earliest artificial cognitive systems drawing inspiration from a number of fields; see Figure 2.1. The early work was ambitious and broad leading to many paths being taken to develop the concept over the subsequent 40 years. Coupled with the fact that replicating cognition is in itself a difficult problem this led to the field being affectionately termed 'a quagmire' with a lack of widespread adoption.

R.J. Urbanowicz and W.N. Browne, *Introduction to Learning Classifier Systems*, SpringerBriefs in Intelligent Systems, DOI 10.1007/978-3-662-55007-6_2

"LCSs are a quagmire - a glorious, wondrous and inventing quagmire, but a quagmire nonetheless" D. Goldberg 1992.

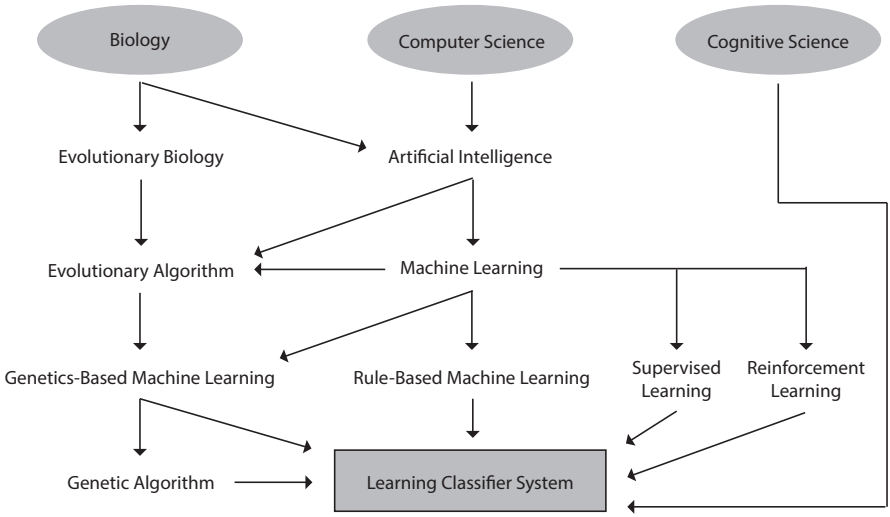

Fig. 2.1 Major influences on the LCS concept. Other influences exist, such as unsupervised learning, that can be incorporated into LCSs

However, the field of LCSs is no longer a quagmire. Research on LCSs has clarified understanding, produced algorithmic descriptions, determined 'sweet spots' for parameters and delivered understandable 'out of the box' source code. This textbook reveals the boardwalk across the swamp so you will be able to proficiently implement and apply LCSs. The first sections of this chapter are organised to reflect the concepts behind the term 'learning classifier system' itself.

2.1 Learning

Learning is valued by humans as it enhances our abilities to solve problems and adapt to our environment. Much work has been conducted in research fields such as education, psychology and neuroscience into how humans learn. With the advent of computers, humans have been interested in how artificial 'agents' learn, either learning to solve problems of value that humans find difficult to solve or for the intellectual curiosity of how natural/artificial learning could be achieved: 'Learning' has a very useful definition for our purposes "Learning is constructing or modifying representations of what is being experienced" Michalski et al., 1986.

The first step toward LCS learning is exposure to domain experience in the form of data. This can be through recorded past experiences (termed offline learning) or interactive with current events (termed online learning). Online learning includes

embedding (embodiment) in robotic systems that may directly act to manipulate their environment. Offline learning often accesses data from a database. To illustrate learning in LCSs this book will rely on offline learning examples (until Chapter 4). Domain experience can also be 'physical' as in the embodiment of a robotic system that may directly act to manipulate its environment or 'virtual' as in a software program receiving data.

Learning is often guided by feedback to improve the reward returned through environmental interaction. It can also be latent where only state-action-state patterns are learned (i.e. if we are in state 'X' and take some action, how do we expect the state to change?). Noise and dynamics within the data may impact learning ability, but research shows that LCSs can tolerate relatively poor quality data and still learn.

At a minimum, in order for artificial learning to occur, it is necessary to have data containing generalisable patterns, or 'signal'. Consider the adage, 'If garbage in Then garbage out' that governs machine intelligence'.

If *there are no knowledge-based patterns in the data*, Then *LCSs will not learn.*

2.1.1 Modeling with a Ruleset

In an LCS, the learned rules are grouped together in a set referred to as the population. This rule population can be interpreted as a model for the data. This reliance on multiple rules to encapsulate patterns in the domain differentiates LCS from other standard versions of AI techniques (such as genetic algorithms, decision trees or artificial neural networks) that build a single entity (e.g. solution, tree or network) that describes the learned knowledge.

A rule population is not to be confused with an ensemble of models (e.g. random forests) that integrates the knowledge of a group of single entity models. Rather, each rule is 'context dependent', i.e. only relevant to a subset of the problem space. A consequence of this is that no single LCS rule may constitute a complete model.

An LCS forms and improves its ruleset model through experience with the data. During this learning phase it increases the worth of good rules, generates plausibly better rules, and removes poor rules. Once learning is completed the performance may be confirmed using unseen data. If learning is deemed successful, then the rules can be used in two main ways.

- The population can be used to predict the class of newly encountered input.
- The rules can be analysed to understand the learned knowledge.

This book expands upon LCS learning in the remaining chapters.

2.2 Classifier

Generically, the term 'classifier' can refer to any means to assign input data instances to some 'class' among a given number of available classes. It is common practice to know (or decide) the number of available classes prior to learning. For instance in the multiplexer problem, only two classes, i.e. '0' and '1', are available. The class of a given data instance can more generally be referred to as the dependent variable, outcome, endpoint, phenotype or action. These terms may be used interchangeably, but 'action' is most often used in the LCS literature. As with most machine learning methods, LCS seeks to predict a single action, which will have multiple class labels (e.g. the action might be 'diagnosis', and the class labels might be 'sick' vs. 'healthy'). If multiple actions need to be learned, then a single LCS may be assigned to each task.

The term 'classifier' has a specific meaning in the LCS field, as we will see. Unfortunately, having the term 'classifier' in the name of the field incorrectly suggests that LCS algorithms are limited to classification tasks, i.e. tasks with a discrete number of classes as the action of the model. While this book tends to focus on classification problems, it is important to note that some LCSs have been adapted to regression problems that have a continuous-valued action. One LCS-based definition of a 'classifier' offered by Stewart Wilson reflects how LCSs may be applied to continuous-valued actions: "a computational structure that computes a function of its input within a subspace selected by its conditions." This will be explored more fully later when we consider how inputs are mapped to outputs by an LCS.

2.2.1 Rules

Rules are the fundamental building blocks of an LCS model. Rules follow a standard 'If ... Then ... ' statement format that is essential to LCSs. This has similarities to production rules in computer science, so the name 'rule' is used. The learned patterns are represented in the form of 'If *this* Then *that*' rules. This could be thought of as sets of antecedents and sets of consequents, but the flexibility of LCS means that these are not always strictly sets. The pattern could link *condition* to *action*, *state* to *action*, *features* to *class* or many other variations (explored in Chapter 4). For convenience we will refer to a rule's *condition* and *action* in this text, but other variants may be substituted as applicable. Rules can thus be interpreted using the following expression:

'If *condition(s)* Then *action*'

Notice how rules are context dependent, meaning that the link to the action relies on satisfaction of the condition. For each feature or encoded digit in the data, the condition can either specify a value or apply a wild card symbol (covered in Section 2.2.3.1). Each possible value of a given feature can be referred to as an *allele*, taken from the evolutionary-genetics-inspired roots of LCS. For convenience, it is typical

to refer to the antecedents part of the rule simply as the 'condition', in the singular, even if it includes more than one specified allele (generally one per feature).

The 'If ... Then ... ' form of LCS rules also leads them to be human readable - termed 'transparent'. The use of context dependent expressions, plus the transparency again separates LCSs from other techniques, e.g. those that use networks of trees to represent the model of the data. The purpose of an LCS is to produce rules that encapsulate the patterns within the data in an interpretable manner.

2.2.1.1 Rule Worth

An 'If ... Then ... ' rule may be valid syntactically, but we need to verify its *worth* (i.e. value) for it to be useful. A valid rule can quite easily encode meaningless relationships and information. Interestingly, the majority of syntactically valid rules are likely to encapsulate incorrect knowledge - consider all the possible arrangements of rules where the condition represents two numbers to sum together to produce the answer as the action, e.g. '3,3 : 6' is correct, but '3,3 : 5' and '3,3 : 7' are equally valid despite being incorrect. Interaction with the domain (in this case reporting 'correct' and 'incorrect') enables experience to determine the worth of a rule.

Evolutionary algorithms that produce single model entities refer to the worth of a model as its '*fitness*', due to analogies with biological fitness. For the same reason, LCS uses the term 'fitness' to refer to the worth of a rule, since most LCSs utilise an evolutionary rule discovery mechanism. Assigning fitness at the rule, rather than at the model level does lead to a few complications given that rule fitness generally reflects the worth of the rule with respect to the problem subspace the rule's condition applies to, rather than the problem space as a whole. Rule fitness is central to the operation of an LCS. There are many ways it can be calculated (see Section 4.3), each having advantages and disadvantages in a given problem domain (see Section 4.5). Instantaneous, filtered, or long-term fitness values may be employed.

A simple and effective fitness measure relevant to supervised learning is the number of correct data classifications (action of the rule being the same as the known action from the data) divided by the total number of times the rule (condition) has matched the input data; see Equation 2.1. This is also known as the long-term accuracy of the *i*th classifier.

$$Fitness_i = |C_i|/|M_i| \tag{2.1}$$

Alternatively, '*error*' informs us how often the classifier failed to model the data correctly. This may be calculated as one minus accuracy; see Equation 2.2. Considering that fitness is not always pure accuracy, including the error statistic can be useful as it indicates the negative consequences of invoking the rule. As will be seen later, fitness can be relative to other rules, shared amongst rules, recency-weighted, reward-based and so on, which may all be different to cumulative accuracy.

$$Error_i = 1 - |C_i|/|M_i| \tag{2.2}$$

Fitness embodies the past success of the rule in modelling the data it covers, indicates the quality of the knowledge held in the rule, and predicts the likelihood of the class (action) arising from future similar state input examples. It can relate to external (e.g. the prediction of feedback from the environment) and/or internal (e.g. overall contribution of the rule to the system) effects.

2.2.1.2 Rules Versus Classifiers

A rule on its own is not much use as it does not tell us its confidence of correctly mapping its condition to its action (e.g. fitness) or its usefulness to the whole set of rules. With respect to LCSs, we refer to the combination of a rule with its statistics as a 'classifier'. Let's examine some common rule statistics.

The most important statistic is *fitness*, which describes the worth of the rule. Rule *error* may also be included as an alternative or complement to rule accuracy.

The *prediction* statistic, relevant to reinforcement learning LCS, predicts the value of the reward returned (i.e. feedback) from any subsequent environmental interaction. In XCS, the most popular LCS algorithm, fitness is based on the accuracy of this reward prediction (see Section 4.3.3.1).

Another key statistic is *numerosity*, which represents the number of virtual copies of each unique rule currently included in the population (see Section 3.8.1).

Other common statistics capturing rule properties include, but are not limited to

- *Lifespan* - often stored as the generation at which the classifier was first created.
- *Reproduction* - e.g. time since last rule discovery invocation.
- *Experience* - which simply collects how many times the rule's condition has matched the input data.
- *Average set size* - the average number of similar classifiers estimated by interaction with the environment.

Occasionally, statistics are kept in order to monitor the performance of the system and for academic curiosity (rather than to influence learning directly). Such statistics can include *Number of offspring*, *Parentage* (description of parents) and other variations, which are only limited by the storage capability of the computer and the time required to collect them.

2.2.1.3 Niche

The concept of a *niche* is central to LCSs, but is not a common term in other areas of EC. A niche refers to a subgroup of instances, i.e. an area of the sample space in a target problem, where the neighbouring instances share a common property. In many data models a single cluster of instance states (i.e. niche) is linked to a single class. There often arises the case where different clusters are linked to the same class. LCSs cope with this difficulty as different single rules can be assigned to each cluster. Multiple rules cooperating to model the niches in the data is an important

(in fact, core) feature of LCSs. We can think of a niche as the group of instances accurately covered by an optimal rule. Later in this chapter we will consider how rule accuracy and generality relate to rule optimality. Also note that the definition of a niche is fluid, dependent on the sample space and representation used.

2.2.2 Representation and Alphabet

The representation of the subspace that the condition of a classifier covers is central to its ability to form accurate classifiers. The set of symbols, termed the alphabet, used to encode the rules must be appropriate to the problem domain otherwise resources will be wasted with unnecessary classifiers in non-compact populations or even lack of performance as some subspaces will be incapable of being described. Fortunately, a wide range of simple, rich or expressive alphabets have been created that are known to well suit certain types of problem domain. For example, real-valued alphabets can suit real-world knowledge discovery problems, while ternary alphabets suit Boolean logic problem domains.

Over two decades ago a ternary alphabet was the most common LCS alphabet, but this has long been superseded for the majority of real-world problems that LCSs address. However, any domain where the message (environmental state) to be matched to the classifier can be represented by zeros and ones {0,1}, e.g. on/off, true/false, 0/1 and so forth, is suited to the ternary alphabet. The alphabet that is referred to as 'the ternary alphabet' consists of {0,1,#}, where the # symbol represents 'don't care', which generalises to either 0 or 1. In binary problems it happens to be equivalent to a logical OR operator such that # matches either 0 OR 1 in the environmental state. Note that in data with missing values that could be either 0 or 1, a # is often also considered a match.

It is important to realise that features, i.e. independent variables making up the state of a given data instance, can be represented directly by an alphabet (i.e. one dataset feature equals one evolvable digit in the condition), or can adopt an encoding where one feature is encoded by multiple digits. For example, in trying to learn the optimum time to open a convenience store it would be possible to adopt a six-character binary encoding for the opening time, e.g. 000111 (7 a.m.), or 001000 (8 a.m.). However there can be problems with representations that apply encodings. For instance, it is possible to produce syntactically valid rules that do not relate to the domain. An example would be using this six-bit binary encoding for time; although the string '111111' in the condition is valid, it does not represent a state found in the problem (i.e. there is no 64th hour of the day). Conversely, if the representation is not flexible enough there may be states in the environment that cannot be described by the conditions. For example, if a four-bit binary number is chosen to represent the hours in a day only 16 possible hours can be encoded).

There is a link between how the rules are represented and how they are expressed, i.e. how the encoding is realised in the environment. In the above examples, the encoding 000111 could be realised as 7 a.m. The encoding itself is termed the genotype

(again following the biological analogy). The expression of the encoding is termed the phenotype. There are multiple ways that a phenotype may be encoded by a genotype, and multiple ways a single genotype may represent different phenotypes.

Sparseness is an important consideration in selecting a representation for a problem domain. A sparse problem is one where many of the valid states do not have an associated instance (or class). In other words consider a valid rule that does not match any instance in the data. Both the size of the dataset and the complexity of the problem domain can influence sparseness. Fortunately, LCS algorithms employ mechanisms such as 'covering' to avoid and in some cases prevent rules that match nothing in the problem space from appearing in the rule population.

Also, the distance between similar genotypes (and their expressed phenotypes) in a search space is an important consideration when deciding upon the representation to use within an LCS. Again consider the convenience store opening time six-bit encodings described above (i.e. 000111 and 001000). There is a big difference (four-bits) between these condition states despite the values (i.e. 7 a.m. and 8 a.m.) being very close (one hour) in the original data. The consequences and solutions (such as using Gray encoding) are explored in more detail in Section 3.3.

2.2.3 Generalisation

If a single classifier mapped to a single state in the dataset, then the system could work but not well. Each state would need its own classifier, essentially enumerating the problem, which would require an enormous population for real-world problems. Not only would this be time-consuming to execute, it would not discover any of the underlying patterns or mappings in the problem. Thus, for efficiency, ease of human readable rules and compactness of population, it is better to have multiple states being addressed by a single classifier. The concept of generality is fundamental to LCSs, i.e. the ability of a classifier to address more than one state in a single rule.

2.2.3.1 Don't Care '#' Operator

Generality in LCSs is achieved through the 'don't care' operator, which is traditionally given the symbol #. This symbol is also referred to as a 'wild card'. It is used in a condition of a classifier rule to symbolise that the LCS should not care about the value of the corresponding feature in the environment, i.e. it effectively marks redundant/irrelevant features in the particular case of the rule. Note: Alternative accurate classifiers may consider this feature important in different problem instances and the niches associated with those instances.

A don't care symbol, or equivalent form, is utilised in most LCS alphabets used to encode the environmental message into an applicable format. Its functionality is most readily understood in binary problems, where the environmental state is represented by $\{0,1\}$. Binary problems are commonly addressed in an LCS by utilising

the ternary alphabet {0, 1, #} here. # represents 0 or 1, i.e. all possible values of the state. When a state is matched to each condition in a classifier, # successfully matches 0 or 1 (cf. the OR operator in genetic programming).

In the ternary alphabet, a rule with one # matches two states, e.g. 11#1 : 1 matches both the inputs 11**1**1 and 11**0**1. 2^n states are matched for every n #s in the rule, e.g. 1001 : 0 matches one state and is completely specific, #### : 1 matches 2^4 states and is completely general. At one point in LCS history, a measure of the specificity of a rule was used to guide learning, but as the most appropriate number of #s in the rule is typically unknown a priori, this genotypic measure has fallen out of favour.

When interpreting rules at the end of training, #s enable compaction of rules and identification of important features that map state to action in a given problem.

2.2.3.2 Overgeneral Rules

It is not sufficient for a rule to match many instances of the problem if its recommended action is occasionally incorrect (in clean problems) or often incorrect (in noisy problems). Consider the case of reinforcement learning; while the obtained reward for correct predictions will be high, the reward will be inconsistent as it will occasionally effect an incorrect action, i.e. occasionally scoring no reward for the incorrect action. This inconsistency is reflected in low accuracy of prediction, albeit the average value of the prediction may be high. A rule is termed *overgeneral* when it covers more search space than is consistent with its action (i.e. too many #s in the rule). Note that when it comes to noisy problems (i.e. where we can not expect rules to make 100% accurate predictions), it can be very difficult to distinguish an overgeneral rule from an optimal rule that captures a noisy pattern. However, it is important to understand that overgeneral and overspecific rules can emerge and play a role in solving both clean and noisy problems.

2.2.3.3 Overspecific Rules

Alternatively, a rule can cover a specific area of the search space where broadening the region of the search space still requires the same action, i.e. a specified condition value can be replaced by a # without error in prediction and loss of accuracy. In binary domains, replacing a specific bit with # doubles the number of instances a rule matches if the instances are evenly distributed across the sample space. Such rules are termed *overspecific*. In clean problems, unlike overgeneral rules, overspecific rules are 100% accurate. They are useful to the system as their recommendation can be trusted, but they waste resources and often fail to identify all redundancy/irrelevancy in the domain. Given time, the rule discovery components will identify the more general version. This will displace the overspecific rule through the deletion process as the more general rule will be bred more often while each rule will have the same chance of deletion. This increased breeding opportunity for more general rules is an implicit generalisation pressure unique to LCSs, that encourages

parsimony (i.e. rule simplicity/generality). However, this can require many environmental learning iterations, which has led to the subsumption heuristic method being developed for LCSs to improve this process (see Section 3.11).

Noisy problem domains again complicate the identification of overspecific rules, since we can't expect either optimal rules or overspecific rules to be 100% accurate.

2.2.3.4 Maximally General, Accurate Rules

LCSs seek to 'optimally' form maximally general, accurate rules, i.e. not overgeneral or overspecific, while always being correct in their recommended action (again assuming a clean problem). LCSs have many complementary mechanisms (heuristics) to achieve this balance between wanting to cover as much of the domain as possible with a rule, while still producing an accurate map of input to output.

2.3 System

'Systems' are bounded entities having inputs and outputs, with a means to compute outputs from inputs. Feedback is a core concept. Interaction with the environment through feedback of the utility of the hypothesised model is essential to guide an LCS in generating its rules. Feedback may be either the best possible action to take given a state (supervised learning), the utility of the action taken (reinforcement learning) or the next state encountered (latent learning, Section 4.7.3).

Systems consist of many components, which is especially true in LCSs. Components within a single LCS include methods to match (rules to input), select/predict (an appropriate action), evaluate (determine worth) and discover (potentially better rules). LCS research has produced many variations of each component. The specific assembly of components defines the architecture for each unique LCS algorithm (think algorithmic building blocks), where some are designed for a very specific application, and others are more broadly implemented. This has increased the flexibility and applicability of LCSs at the expense of increased complexity (and impenetrability to new researchers, which this book seeks to address). Consequently, LCSs have been described as a concept rather than a single technique.

2.3.1 Interaction with Problems

In order to solve a problem, an LCS must interact with the problem domain. It must search the space of possible rules in order to map areas of the domain to separate classes. This ability to interact with the problem (e.g. the 6-bit multiplexer problem), or environment (e.g. Boolean), stems from the history of LCSs as an artificial cognitive system rather than an optimisation technique.

It is worth differentiating between the *sample space* and *search space* of a problem. The sample space is the unique instances (messages) available from the problem domain, e.g. 2^6 for the 6-bit multiplexer problem. The search space is the unique rules that can be created, which is linked to the sample space together with the chosen alphabet of the LCS; e.g. 3^6 for the 6-bit multiplexer problem when using a ternary alphabet (i.e. an alphabet with three symbols).

An individual *solution* within a domain can be thought of as a set of state-action rules that encapsulate the underlying properties of the problem. The task of an LCS is to autonomously identify condition-action rules to describe a problem, which is analogous to a map which shows where actions sit on conditions. LCSs can make no mathematical assumptions about the relationships between conditions and actions, so discontinuous, non-differentiable and otherwise stochastic domains can be described. LCSs are commonly set up with a separation between conditions and actions, but they can also form a direct link where conditions are used to calculate the actions (a computed model or function); see Section 4.8. Thus, although this book will refer to conditions mapping to actions, this is not always the case.

Tasks within the problem domain may be described as classification, modelling (including regression), optimisation (this can be framed as classification of solutions that return the highest value) and many other descriptions.

2.3.1.1 Environment Properties

An *environment* is the 'out there' world that may be sensed by an agent and effected upon. It is the source of data and the 'home' of the problem at hand. Considering the definition of an *agent* as observing an environment (sense), affecting the environment (act) and purposeful (goal directed), then an LCS may be considered as an agent. The environment has a defined boundary, which is often an intangible construct rather than physical input-output electronics, that inputs and outputs cross to enter and leave the agent.

Often an agent is constrained to *domains* within the environment due to its sensors, such as interacting with Boolean states only. Early LCSs' domains required that the input data could be encoded with 0s and 1s only. This resulted in a lack of precision in many domains and so real-valued encodings were created/adopted. Modern LCSs can utilise a wide variety of representations so can interact with many different types of domain.

2.3.1.2 Learning, Adaptive, and Cognitive Systems

It aids insight to consider the broad conceptual differences between learning systems, adaptive systems, and cognitive systems. A simple, albeit not rigorous, explanation follows: A basic system can have many predefined rules where the structure of each rule remains constant over time (a condition to action rule population), with only the worth being changed based on experience (similar to adjusting only the

weights on a fixed-topology artificial neural network). This is a *learning system* as it changes due to interaction with the environment during the life of the classifier, but it does not modify its representation of the environment (i.e. no rule discovery or deletion). The so-called '*sense-plan-act*' robotics cycle can fit the model of a learning system provided all patterns are already known about the world. This does not work when novel states are sensed or insufficient rules exist to plan correctly.

An *adaptive system* is one that changes (evolves in this case) its functionality (e.g. mapping between inputs and outputs) in order to better inhabit its environment. Adaptation occurs in LCSs when new rules are created in response to environmental needs - environmental inputs (e.g. no currently matching classifiers exist), feedback from the environment (e.g. sub-100% performance occurs) or chance/serendipity.

A *cognitive system* extends the sense-plan-act cycle to a '*perceive-represent-reason-learn-act*' cycle. *Perception* receives the raw sensor values, i.e. features of the domain, and focuses only on important features of the state of the environment. *Representation* encodes the features such that they can be manipulated by the system (e.g. corresponding to the condition of an LCS). *Reasoning* is important as it is not always straightforward how to map an input to an output (see Chapter 3). Furthermore, artificial systems may reason about which maps might be manipulated to form potentially better maps, hence modifying the mapping of the environment (see Section 3.9). The ability to change the structure of the knowledge is a distinguishing property between artificial intelligence techniques. *Learning* the worth of rules (see Section 3.6) and adapting to the environment through new rules (see Section 3.10) are both core to LCSs' performance. Finally, *Act*, to effect the output of the cognitive process in the environment is important, although often trivial, in common applications, e.g. reporting the class in data mining applications.

Hence, although the main task of an LCS is to learn to classify data in a domain, it is useful to acknowledge their cognitive abilities/roots as this assists in improving their core functionality. LCSs as cognitive systems, together with comments on philosophical discussions of whether an artificial system can be intelligent, are beyond the scope of this book.

Prior to leaving the cognitive heritage, there are a couple of ideas that assist in improving LCSs' performance in certain problem domains, i.e. *cheap computation* and *embodied agent*. First, 'cheap' means low-cost to the system resources, rather than low monetary value. This directs that any LCS should be matched to its environment in such a way that simple solutions result, e.g. the use of an appropriate representation scheme and selection methods to suit the task.

Second, it is noted that the goal-directed behaviour of an agent is often a single goal provided by the user, but multi-objective and goal-switching LCSs exist. This is especially true in robotic and network applications where the *embodied (situated) agent* is more concrete than other applications, say data mining domains. The interaction with the environment is crucial to the success of LCSs, as if they are not well suited then they may fail to function as expected.

2.3.1.3 Evaluating Rules

A single rule encapsulates a single If:Then pattern, but there are typically multiple patterns in a problem. Thus, LCSs have multiple rules in a population that cooperate to describe the problem. In order for the rules' fitness to be evaluated, the LCS must interact with the environment (see Figure 2.2). In certain domains the environment returns the known (best, current, optimum, ...) action that the system could have selected from the environment, e.g. turn +65 degrees. This mechanism is termed **supervised learning** (SL) when the environment supplies its *state* and known best *action*, e.g. medical risk factors and clinical diagnosis. Many practical applications, such as data mining, commonly use SL.

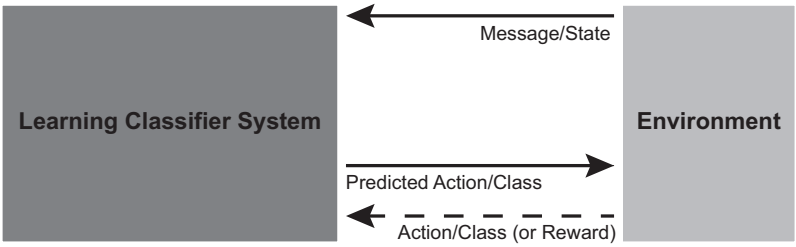

Fig. 2.2 Environmental interaction with an LCS. Note that the environmental state is often referred to as the environmental message when processed in an LCS. LCSs usually return one action to the environment at a time

Differently, this mechanism would be termed **reinforcement learning** when only the utility of the action is provided by the environment rather than the optimal ground truth, e.g. increase in light intensity for a photophilic robot. There is no explicit punishment in LCS reinforcement learning, as lack of reward is considered a sufficient signal of lack of worth of a given rule in such a situation. Originally, all LCSs used reinforcement learning as it was assumed that a 'teacher' that knew the correct and/or incorrect actions was unavailable. Nowadays, as LCSs are being applied to many real-world problems with known training data, the SL approach is gaining precedence.

2.3.2 Cooperation of Classifiers

LCSs are considered a population-based technique in that they have a population of classifiers that describe the patterns in the data.

A single rule in an LCS models a distinct part of the data (i.e. a niche). If there was only one distinct part of the domain, then only one rule would be needed, e.g. a six-dimensional binary domain $\mathbb{B}^6$ could be completely covered by ######:*Action.* However, the vast majority of domains of interest have multiple parts that require

modelling with different rules. Thus LCSs must learn a set of rules if there are multiple distinct parts of the domain.

The rules within a population *cooperate* to map the domain (see Figure 2.3). Here a six-bit problem map is visualised by the two bits on the y-axis and four bits on the x-axis. The labels on the axes represent the conditions, while the numbers in the corresponding grid cells represent the actions $\{0, 1\}$. In this case, the map represents the 6-bit MUX problem with the two address-bits on the y-axis reading left to right and the foud data bits on the x-axis reading top to bottom. Considering the bottom row, indexed by 00, it can be observed that the first eight elements of the row (in blue) represent a niche, where all instances specify the action 0, such that the 'optimal', maximally general and accurate classifier, 000### : 0 covers this niche.

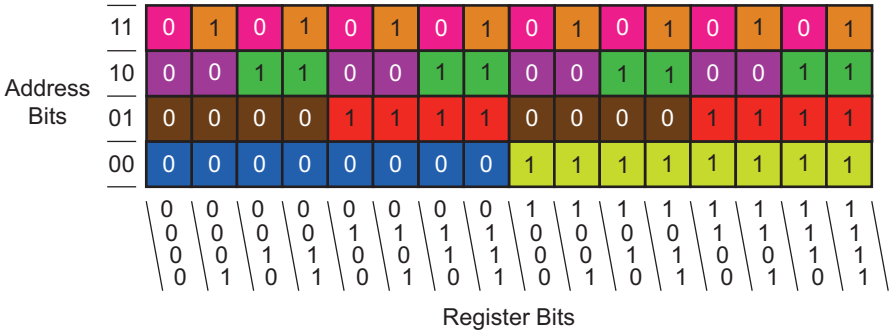

Fig. 2.3 Cooperative rules for the 6-bit multiplexer problem. Each region (same shade) represents a classifier (rule) with associated class (0 or 1) that optimally covers a niche within the domain, e.g. 000### : 0 is the rule for a niche on the bottom row. There are eight rules in this population, which are accurate and maximally general

2.3.3 Competition Between Classifiers

Ideally, there would only be one unique and correct rule for each niche of data that is linked to a specific action. Thus the population would contain the same number of rules as the domain contains distinct niches of data. However, in the majority of domains prior knowledge does not exist, so each rule must be learned. Instead of a single estimate of each unique rule, LCSs allow multiple, slightly different rules for each part of the sample space (i.e. *overlapping rules*). That is, multiple hypotheses are available in an attempt to find the optimum rule for each niche. This is one reason why we might consider LCSs to be implicit ensemble learners.

Therefore, the rules within a niche *compete* to be able to map that portion of the domain. Each rule 'covers', i.e. describes, its part of the search space. Multiple

rules can cover the same instance or cluster of instances, so they must compete to determine which best describes that data.

Implicit in the concept of overlapping rules competing with each other is that each rule should cover more than one data instance - preferably all data instances in the cluster. In the rare instance where there is only one data instance in a cluster, then a single, specific rule may be appropriate.

Competition within a niche can occur between overlapping rules that may be overgeneral, overspecific, optimal, or simply poor.

2.4 Problem Properties

Before implementing an LCS, it is worth considering how well we know the problem. If we know the map of conditions to actions (or models that generate the input-output pairs) then the problem is solved. It may still be of interest in terms of studying how the appropriate rules can be created by the technique, or discovering generalisations of the patterns. Such known problems are described as 'toy' problems, in contrast to real-world problems where the patterns, relations, noise and so forth are not known a priori (e.g. the n-bit multiplexer)

If it is possible to enumerate all the inputs to discover the outputs in a tolerable time limit, e.g. the problem is linearly separable, then there is no need for advanced techniques like LCS. This is dependent both on the cost of producing the output and the time it takes to produce all combinations. Similarly, if the problem is mathematically describable and tractable with existing algorithms, then there is little need for artificial-intelligence-based techniques, including LCSs, unless a different format of a solution is sought, or existing solution interpretability is insufficient. However, most interesting toy and real-world problems have features that are redundant, irrelevant, epistatic, heterogeneous, discontinuous, poorly correlated and so on.

2.4.1 Problem Complexity

A problem of interest is called 'complex' if simplification renders the solution useless, and non-easily decomposable if the problem cannot be simply broken into subproblems where each is solved and then easily recombined to produce the global solution. LCSs are well suited to autonomously decomposing complex problems since they rely on multiple rules. Autonomous decomposition is often more successful as humans may not know how to break up the problem appropriately, have bias, or may not be accurate at defining the boundaries between niches.

We assume that the search for the optimum solution (best set of rules) is not blind, i.e. feedback exists prior to the optimum solution being evolved. That is, part solutions and partly correct rules have a measurable and meaningful worth, which the system can translate into the individual fitness of rules. Further, this feedback

value and the sensed state are not so corrupted by noise to be meaningless. Similarly, both the problem and feedback have low time variance (see Section 4.7).

Other problem characteristics can render the problem easier or more difficult for any machine learner. Important characteristics that have been investigated in the context of LCSs include the size of the search space, modality (i.e. number of classes and number of niches within a class), smoothness (continuity of genotypic to phenotypic mapping), separation of classes (overlap of niches, boundaries between classes), Hamming distance between consecutive rules, robustness of the solution, separation (features, niches), noise and linear separability of classes (or not). Below we elaborate on complexities resulting from the size of the search space, redundant and irrelevant features, and two important complicating data phenomena that LCSs are particularly well suited to addressing.

2.4.1.1 Size of Search Space

The LCS search space is the number (set) of all possible solutions in terms of the number of condition-action pairs that the rules can encode. The size of the search space can be influenced both by the available number of data instances (assuming LCS rule discovery mechanisms only allow rules that match at least one training instance to emerge) by the number of features observed in the environment and how features are encoded. For example, consider how, based on encoding, the string '100110' could represent one feature encoded as a six-bit number, two features with a three-bit encoding or six features each with Boolean states. Encoding can extend the length of the condition and thus can increase the size of the search space.

With regard to representation, there is often a trade-off between a simple alphabet that makes searching the sample space complex and a complex alphabet that makes searching the sample space simple. The larger the size of the search space, the longer it actually takes an LCS to explore the alternatives, where this often does not scale linearly. Furthermore, the number of classifiers needed to cover the domain is likely to increase. When this happens the *curse of dimensionality* becomes more of an issue, where an exponentially larger number of instances are required to identify complex, high-dimensional niches.

2.4.1.2 Redundancy and Irrelevance

Redundant features are not needed to model the problem as other features represent a better model, e.g. they give a better prediction or are more compact. While redundant features still map inputs to outputs, they are not essential to the problem. For example, when predicting height-to-weight ratio, the feature 'height in feet and inches' is equally important and would still map to the underlying pattern, but is still considered redundant provided that the feature 'height in centimetres' exists.

Irrelevant features are those that do not map to the underlying patterns in the problem. They can be removed from the environmental data without decreasing the

performance of the system as part of feature selection. Removing irrelevant features often improves a classifier's performance as the irrelevant features act as noise by creating many false patterns by chance, i.e. patterns that fit the training data but do not encapsulate true generalisable patterns in the problem applicable to unseen 'test' data. However, irrelevant features often can not be removed ahead of time, which can confound machine learning. LCSs have been shown to be effective in the face of redundant and irrelevant features, however improved performance and learning efficiency is always expected in their absence.

2.4.1.3 Epistasis

A big selling point of LCSs is the ability to handle the interaction of different features within a problem. Epistasis is a biological term that has been adapted into AI classification to describe the phenomenon where the value of one feature affects the importance of another feature, e.g. height and weight in predicting obesity. The variables are no longer independent, which breaks assumptions made in certain techniques, e.g. naïve Bayes classifiers. This affects their performance to a degree dependent on the level of epistasis. LCSs cope with epistasis well as they produce a map that is not based on such independence assumptions.

2.4.1.4 Heterogeneity

Another unique selling point of LCS is the ability to have different conditions mapped to the same action, e.g. different patterns that cause the same effect. This acknowledges that separate niches can map to one action. Thus, LCSs divide up the search space into multiple rules, rather than having to discover one representation per action, which is often a hard task due to the unrelated features needing to be combined. In contrast with the standard machine learning paradigm that seeks a single best model under the assumption of a homogeneous pattern of association, LCSs are uniquely well suited to modelling heterogeneous problem domains.

2.4.2 Applications Overview

The original intention of LCSs was to explore natural systems through evolving artificial systems, but the subsequent research in the field has focused on solving interesting problems. The field of LCSs has also been subsumed into the wider field of Evolutionary Computation. LCSs are now either tasked with doing valuable problem solving in industry/business or with exploring what problems can be solved academically. Therefore, the majority of this book will focus on how to use LCSs to solve interesting problems.

So what type of problems can LCSs learn? Fortunately, if a problem can be described in 'input-output' pairs, then LCSs can be applied. Consider optimising a function within known input bounds; we are interested in determining the parameter values that correspond to the class of the function's optimal value. Scheduling, game playing, control and so forth can all be addressed by LCSs in this manner.

More generally, as we have seen, LCSs are commonly applied to both supervised and reinforcement learning tasks, including classification, data mining, regression, function approximation, behaviour modelling, adaptive control and more. Such problems can be single- or multi-step problems where the action relies only on the current state, or potentially on previous states from the environment, respectively. With regard to classification, LCS can handle binary classes or multi-class problems as well as class imbalance. Additionally, LCS can function despite noisy data, redundant or irrelevant features, feature type (discrete, continuous or mixed), or the presence of missing data (i.e. missing feature values in training instances).

The adaptability of LCSs has been perceived as both a strength and a weakness - a 'jack of all trades, but master of none' has been a label. However, domains in which LCSs appear to outperform all other approaches have begun to emerge, including domains with complex heterogeneous patterns of association. Perhaps a better analogy for LCSs would be a 'Swiss Army knife'. There are certainly many tasks where LCSs may be better than alternatives, but we do not claim that LCSs are better than all other tools for all tasks.

LCSs seek to produce a solution containing accurate and maximally general rules forming a compact ruleset. There is cooperation between rules to map inputs to outputs, but there is also competition between rules to optimise each niche within a search space. This creates many interacting pressures with LCSs having built-in heuristics to guide this process. These heuristics seek an effective balance between overgeneralisation and overfitting, direct the population to good/promising areas of search, compensate for missing data and adjust for unbalanced data. These LCS heuristics are explored fully in the next chapter.

Finally, it is worth considering what are the characteristics of problems worth targeting with LCSs, given they are not the most efficient method for simple problem domains. LCSs work well when there are perpetually novel events, i.e. when the required action changes due to the frequent change in environmental state. This can be accompanied by noise and/or irrelevant/redundant data as LCSs have the ability to generalise to form an underlying predictive model. The domain can have continual, including real-time, requirements for actions - noting that learned rules are effected in real time. There is no need for explicit or exactly defined goals as LCSs can function under either reinforcement or supervised learning schemes. Payoff (reinforcement from the environment) can be immediate, but also delayed where reinforcement is only obtainable through long sequences of actions. Sparseness in the search space (or payoff) is also accommodated. Applied domain characteristics often include

- Multimodality
- Multiple classes
- High dimensionality (high number of features)

- Epistasis
- Heterogeneity (environmental features of different types)

How all of these abilities of LCSs are achieved is explored in the next chapter.

2.5 Advantages

The uniqueness of LCSs stems from their rule-based approach that has the ability to divide up the problem into more easily solvable niches than in single-solution learning. This is achieved through combining the global search of Evolutionary Computation (EC) with the local optimisation of Machine Learning (ML) in a flexible framework. EC discovers new structure to the solution, while ML tunes the associated statistics and hence interaction of rules in the solution. There are many advantages to this unique core algorithmic architecture.

The output of an LCS is a set of human interpretable rules that represent a *distributed* and *generalised probabilistic* prediction model. *Distributed* in the sense that the learning resources of rules are allocated to the identified niches, which are distributed in the search space, as required. *Generalised* as a single rule covers more than one decision point (problem instance) in the niche that the rule matches. *Probabilistic* means not in the strict Bayesian sense, but rather in the way predicted actions (the model's output) are determined by the collective voting of matching rules (relevant to the input). Comparing votes for each action offers an estimate of the probability or confidence that each possible predicted action is correct.

One of the major advantages of LCSs is their applicability to all sorts of problems. The previous section reviews many of the general applications and problem characteristics/challenges to which LCSs are suited. Below we review other key advantages of LCS algorithms.

Practically, LCSs have many strengths as an EC/ML technique, particularly as rule-based machine learners. One of their major advantages is that they are flexible and adaptable in nature, allowing application to many domains with multiple types of feedback on solution progress available. The previous section reviews many of the general applications and problem characteristics/challenges to which LCSs are suited. Below we review other key advantages of LCS algorithms.

LCSs avoid having to make assumptions about the underlying patterns in the environment, such as linearity of input/output relationships or other mathematical requirements. They suit problems that are composed of subproblems due to their niche-based learning core. Importantly, LCSs will still form a simple solution (a single rule) if that is most appropriate for the problem.

The 'If ... Then ... ' format of rules enables them to be human interpretable, which for many true-valued alphabets is directly readable. Thus, experts in a given problem domain can directly verify the learned knowledge. A personal anecdote is that when utilising LCSs for knowledge discovery in a steel strip mill, the rules highlighted a centring effect of crowned work rolls, which experienced mill operators acknowledged was present but not well known.

With regard to data mining, LCS can be applied as a prediction machine, predicting output from novel input based on past experience. Towards this end, we can split the data into training/test phases, and apply cross-validation, so that independent, unseen datasets are available to evaluate the generality of the solution. LCSs are also used for knowledge discovery in data as they can identify interesting feature relationships and identify redundant/irrelevant features as a feature selection approach.

LCSs are adaptive, meaning that the rule base can acclimate to a changing environment in the case of online learning. They are implicitly ensemble learners, since predictions rely on the vote of a set of 'relevant' matching rules that can suggest different action values. They are also implicitly multi-objective given that a system with an accuracy-based fitness will evolve towards the most general as well as accurate rules thanks to the implicit generalisation pressure unique to LCSs.

2.6 Disadvantages

LCSs are not immune to disadvantages, and it is important to be familiar with those that are currently recognised. One issue is the computational expense, as evolution takes time. However, LCSs are much faster than enumeration (often impractically slow) due to the stochastic-based evolutionary search. Evolutionary search is slow compared to techniques that descend quickly to a single local (hopefully global) optimum solution, so LCSs will not win any speed races against such techniques when both can solve simple problems, but they may prove more effective in complex problems due to the broader search of EC. Importantly, the evolved solution (ruleset) operates as fast at making decisions/predictions as other solutions, which for the vast majority of problems is real-time. Furthermore, the rulebase can be continuously improved offline as the problem changes, while a snapshot of the rules can run in real time online.

Despite LCSs having both implicit and explicit generalisation pressures, they can still overfit training data like any other machine learning approach. Also while rules lend themselves to being human readable, there is an added challenge to interpreting a set or population of rules as a model, rather than a single model entity.

Additionally, like many other advanced machine learning approaches, LCSs have a number of run parameters to consider/optimise. Typically, most parameters can be left to the 'sweet spot' defaults described later in this book, however at least two critical run parameters can be difficult to optimise for a particular problem domain (i.e. the maximum rule population size and the number of learning iterations). Furthermore, LCSs are less well known even within machine learning research communities (however we hope this book helps to change that), there is limited software availability (which is why we have made educational LCS code available along with this book) and there is a relatively small body of theoretical work behind LCS algorithms, likely due to their relative algorithmic complexity as well as their stochastic nature.

http://www.springer.com/978-3-662-55006-9

Introduction to Learning Classifier Systems

Urbanowicz, R.J.; Browne, W.N.

2017, XIII, 123 p. 27 illus., 4 illus. in color., Softcover
ISBN: 978-3-662-55006-9