UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

A metaheuristic perspective on learning classifier systems

Michael Heider, David Pätzel, Helena Stegherr, Jörg Hähner

Angaben zur Veröffentlichung / Publication details:

Heider, Michael, David Pätzel, Helena Stegherr, and Jörg Hähner. 2023. "A metaheuristic perspective on learning classifier systems." In *Metaheuristics for machine learning: new advances and tools*, edited by Mansour Eddaly, Bassem Jarboui, and Patrick Siarry, 73–98. Singapore: Springer. https://doi.org/10.1007/978-981-19-3888-7_3.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Chapter 3
A Metaheuristic Perspective on Learning Classifier Systems

Michael Heider, David Pätzel, Helena Stegherr, and Jörg Hähner

Abstract Within this book chapter we summarize Learning Classifier Systems (LCSs), a family of rule-based learning systems with a more than forty-year-long research history, and differentiate LCSs from related approaches like Genetic Programming, Decision Trees, Mixture of Experts as well as Bagging and Boosting. LCS training constructs a finite collection of if-then rules. While the conclusion (the then-part) of each rule is based on a problem-dependent local submodel (e. g.linear regression), the individual conditions and, by extension, the global model structure, are optimized using a—typically evolutionary—metaheuristic. This makes the employed metaheuristic a central part of learning and indispensable for successful training. While most traditional LCSs solely rely on Genetic Algorithms, in this chapter, we also explore systems that employ different metaheuristics while still being similar to LCSs. Furthermore, we discuss the different problems that metaheuristics are solving when applied in this context, for example, discrete or real valued input domains, optimization of individual rule conditions or entire sets of rule conditions, and fitness functions that support nicheing or control bloating. We ascertain that, despite the optimizer being essential, it has been investigated less directly than the learning components so far. Thus, overall, we provide an analysis of LCSs with a focus on the used metaheuristics and present existing solutions as well as current challenges to other practioners in the fields of metaheuristic ML and rule-based learning.

3.1 Motivation and Structure

In a book about metaheuristics and machine learning (ML), Learning Classifier Systems (LCSs) have to be mentioned as they were among the first metaheuristics-based ML approaches.

M. Heider (✉) · D. Pätzel · H. Stegherr · J. Hähner
University of Augsburg, Augsburg, Germany
e-mail: michael.heider@uni-a.de

Originally, LCSs stem from approaches to utilize genetic algorithms (GAs) for interactive tasks [39, 82]. While they were developed somewhat independently of reinforcement learning (RL) methods in the beginning, the two fields merged later. LCSs build sets of if-then rules and there are variants for all of the major ML tasks. Although some researchers refer to LCSs as *evolutionary rule-based machine learning (RBML)* methods, we would like to stress that—from our perspective—evolutionary optimization is not required but actually any metaheuristic may be employed.

In this chapter, we provide an in-depth view at LCSs from the metaheuristics side. To our knowledge, this has not been done yet, despite the employed metaheuristic playing a central role in these methods. Indeed, neither the metaheuristics used in LCSs nor their operators have received a notable amount of research attention so far. With this chapter, we want to raise awareness of this fact and, by organizing existing research, provide inspiration for new developments in both areas, metaheuristics and LCSs, and thus lay the ground work that researchers can then build upon to more easily remedy this deficiency.

To this end, we start with a short introduction to the field of LCSs, explaining the overall approach and discussing the learning and optimization tasks that these systems try to solve (Sect. 3.2). We then describe the relation of LCSs to other similar ML approaches to highlight similarities and differences and thus deepen the reader's understanding (Sect. 3.3). Next, we elaborate on the role of metaheuristics in LCSs, the types of systems that different design choices result in (in the process proposing a new classification instead of the known Michigan/Pittsburgh one), and approaches to solution representation, metaheuristic operators and fitness functions (Sect. 3.4). Finally, we look at recent ML approaches that employ metaheuristics and that were developed somewhat independently of LCSs but that rather closely resemble what we would call an LCS (Sect. 3.5); some of the techniques from these fields may be relevant for LCS as well.

3.2 What Are Learning Classifier Systems?

While there is no official definition of what constitutes an LCS, the consensus [26, 76] seems to be that an LCS is an ML algorithm that creates an ensemble model consisting of a set of simpler *submodels*, each of which models a certain part of the problem space. At that, the problem space is not necessarily split into disjoint sets—submodel responsibilities may overlap. This is achieved by pairing each submodel with a *matching function* that, for each possible input, states whether that input is modelled by the submodel. Although overlapping responsibilities introduce some additional complexity, they are considered a feature (e. g. the resulting overall model can be expected to be more smooth than one based on disjoint splits). In areas of overlap, the predictions of the submodels that model the input are *mixed*, usually by employing a weighted average. The combination of a matching function and a submodel can be seen as an *if-then rule* (or just *rule*, in short) which is also

the technical term in the general field of RBML. Out of historic reasons, in LCS literature, a rule together with its corresponding mixing weight (and possibly further bookkeeping parameters) is called *a classifier*—hence the name *Learning Classifier Systems*. In this chapter, we try to avoid the slightly overloaded term *classifier* and rely more on the term *rule* which we use to denote a tuple consisting of a matching function, a submodel as well as a mixing weight and bookkeeping parameters such as how often the rule was correct. Just like existing LCS literature, we further use the term *condition* to refer to the genotypic representation of a matching function (i. e. a certain matching function is the phenotype of a certain condition).

Most of the LCSs currently popular in the field were developed in an ad-hoc manner [26]. This means that the systems' mechanisms were not derived from a set of central, formally specified, assumptions about the learning task (e. g. the available data); instead, in particular in very early LCSs and presumably due to their role as proof of concept for the ideas regarding artificial evolution and genetics, methods that were to be used were selected first and then combined so certain kinds of inputs could be dealt with [39]. Later LCSs then built on those early systems: For example, XCS [82] is a simplification (and improvement) of ZCS, which on the other hand is a simplification of Holland's original framework [81].

3.2.1 Learning Tasks

LCSs are a family of ML methods. At the time of this writing and aside from pre- and postprocessing, task primitives found in ML can be very roughly divided into *supervised learning* (SL), *unsupervised learning* and *reinforcement learning* (which is sometimes also called *sequential decision making*) [12]. While there are LCSs that perform unsupervised learning [see e. g. 90], these play a rather minor role in recent years. Most currently researched LCSs are meant for supervised learning (SL) or RL [59, 60].

The very first LCSs focused solely on RL tasks [39, 81, 82], even though that term was not being used in the beginning (RL and early LCS frameworks were developed somewhat independently). However, over the years it has been seen that those LCSs, while performing competitively on SL tasks such as regression and classification (discussed in the next paragraph), do *not* on non-trivial RL tasks [9, 69]. In hindsight it can be said that this was probably the cause of more and more researchers falling back to analyzing what are called in LCS literature *single-step problems*, which are RL tasks with an episode length of one. Since, in many of the considered tasks, there is a finite set of reward levels, these kinds of studies often actually investigate classification tasks. Besides, many of the tasks considered are deterministic which means that the corresponding classification tasks are additionally noise-free. Nevertheless, several existing LCSs *are* capable of solving more or less difficult RL tasks; for example, anticipatory classifier systems [19, 55] whose rules also model the environment's behaviour.

The larger part of LCS research is concerned with SL and, in particular, *classification* and *regression* tasks. At the time of this writing, the probably most commonly used (and certainly most investigated) LCSs for these kinds of tasks are almost exclusively XCS derivatives [59, 60, 76] with a notable exception being *BioHEL* [see e. g. 7, 29]. XCS [82] is one of those aforementioned early systems that originally were meant for RL but that, over the years, have been used more and more exclusively for classification [59, 60]. In spite of that, XCS still has several RL-specific mechanisms that are not directly useful or even detrimental for performing classification; this gave rise to the *UCS* derivative [11, 78] which explicitely focuses on supervised classification tasks.

In 2002, Wilson developed *XCSF* [84], an XCS derivative for function approximation. XCSF has been further improved up to quite recently, resulting in competitiveness, at least for input spaces with low dimensionality [68, 70]. Besides XCSF, there are other lesser known LCS-based regressors as welldrugowitsch2008b. Note that one application of regression is approximating action-value functions in RL problems [72]—which circles back to the early goals of LCS research.

Overall, it can be said that LCSs are a versatile framework that can be used for all different kinds of learning problems.

3.2.2 LCS Models

Most models built by LCSs are of a form similar to

$$\sum_{k=1}^{K} m_k(x)\gamma_k \hat{f}_k(\theta_k, x) \tag{3.1}$$

where

- x is the input for which a prediction is to be made,
- K is the overall number of rules in the model,
- $m_k(x)$ is the output of the matching function of rule k for input x; typically, $m_k(x) = 1$ iff rule k is responsible for input x (i. e. if its submodel models that input), and $m_k(x) = 0$ otherwise,
- γ_k is the mixing weight of rule k; mixing weights usually fulfill $\sum_{k=1}^{K} m_k(x)\gamma_k = 1$ and $0 \leqslant \gamma_k \leqslant 1$ (this can easily be achieved using normalization, which is standard for almost all LCSs),
- $\hat{f}_k(\theta_k, x)$ is the output of the submodel of rule k (with parameters θ_k) for input x.

Note that although the sum goes over all rules, by multiplying with $m_k(x)$ only the rules that match the considered input x contribute to the result (for non-matching rules, the respective summand is 0). Further, instead of binary matching, that is, $m_k(x)$ being a function with codomain $\{0, 1\}$, *matching by degree* can be an option: $m_k(x)$ then has codomain $[0, 1]$ and may correspondingly take on any value between

0 and 1. However, since all of the more prominent LCSs use binary matching, we assume the same for the remainder of this chapter—unless we explicitly write otherwise.

3.2.3 Overall Algorithmic Structure of an LCS

In order to build such an ensemble model, an LCS performs two main tasks, each of which can be seen as having two subtasks:

Model selection consists of

(I) choosing an appropriate number of rules K for the task at hand and
(II) assigning each rule a part of the input space (i. e. choosing matching functions $\{m_k(\cdot)\}_{k=1}^{K}$).

It is important to note once more, that, in LCSs, the input space is not split into disjoint partitions but that instead an input may be modelled by several submodels. The set of matching functions is also called the model's *model structure*.

Model fitting is concerned with selecting the best-performing submodels and combining them optimally, given a fixed model structure. It usually consists of

(III) fitting each submodel $\hat{f}_k(\theta_k, \cdot)$ to the data its rule is responsible for by adjusting its parameters θ_k. It is often beneficial to do this such that each submodel is trained independently of the others, since, that way, upon getting new training examples or changing the model structure (model selection is typically done iteratively and alternatingly with model fitting), only partial retraining is necessary [26].
(IV) fitting the mixing weights $\{\gamma_k\}_{k=1}^{K}$ (see the introduction to Sect. 3.2) such that areas where the responsibilities of several rules overlap are dealt with optimally.

We now shortly discuss the four subtasks with a focus on whether it is common that metaheuristics come into play.

The number of rules K is the main indicator for the overall model's complexity and with that, its expressiveness. A larger number of rules increases the expressiveness at the cost of an increased training effort and a decrease in interpretability. Also, if the number of rules is too large, then overfitting can be expected (extreme case: more submodels than there are data points in the training data) resulting in worse generalization capabilities. On the other hand, if the number of rules is too small, the overall model may not be able to sufficiently capture the patterns in the available

data. Choosing an appropriate number of rules (subtask (I)) is thus an optimization problem.

Based on the chosen number of rules, the next open task is to decide which submodel is responsible for modelling which parts of the problem space (subtask (II)). *This is the task that all existing LCSs use metaheuristics for*. Since the quality of a solution candidate to this subtask highly depends on the number of rules (for example, if there are fewer rules, then, in order to properly cover the problem space, at least one of them has to be responsible for a larger part of it than if there were more rules), subtasks (I) and (II) are often solved together or using a combination of a metaheuristic for (II) and a simple heuristic for (I). For instance, the latter approach has been taken for XCS [82] and its main derivatives, where the set of matching functions is optimized using GAs [11, 82].

How exactly the individual submodels are fit to the data that their respective rule's matching function matches (subtask (III)) highly depends on the type of submodel. For example, for linear submodels, methods such as least squares may be employed [26]. Several well-known LCSs (most prominently, most XCS derivatives) perform online learning which means that training data is seen instance-by-instance and that predictions can be performed during learning. This demands that the submodels support this kind of learning as well; for instance, in the original XCS, the submodels' parameters are updated incrementally using a gradient-based update [82]. Overall, fitting a submodel is also an optimization problem (e. g. minimization of expected risk on the data matched by the corresponding rule) that is typically not approached with metaheuristics.

Given fixed sets of matching functions and (fitted) submodels, the final open question (subtask (IV)) is how submodel predictions should be combined into a single prediction. As mentioned before, an LCS's submodels are allowed to overlap, and for areas where that is the case, predictions of several submodels have to be combined sensibly. A straightforward approach which is used by several popular LCSs is to compute a quality metric for each rule (e. g. an accuracy score on the training data that the rule matches) and use a weight proportional to that metric's output in a weighted average. This is, for example, how XCS does it [82]. While there has been some research regarding this task [26], the larger part of LCS research more or less neglected it and never really explored different methods [26]. In general, computing mixing weights given fixed matching functions and submodels is another optimization problem; just like for each individual submodel in subtask (III), here, for the whole model, the expected risk is to be minimized. While there are cases where provably optimal solutions can be found, their computation is often rather expensive and cheap heuristics based on rule properties or metrics (such as the one of XCS) can be a better choice [26]. To our knowledge, there have not yet been any approaches of using metaheuristics for subtask (IV).

This concludes this first, rather high-level discussion of the four subtasks. Section 3.4 discusses in more detail how subtasks (I) and (II) are approached metaheuristically by existing LCSs as well as the challenges encountered in doing so.

3.2.4 LCSs in Metaheuristics Literature

After introducing LCSs from the ML perspective, we now shortly discuss their presentation in existing metaheuristics literature. Overall, it can be said, that, if mentioned at all, they are only discussed from a rather limited viewpoint with a focus on older systems. In addition, recent work often does not include LCSs at all. In the following, we summarize the main aspects we found—note how this contrasts the more general description of these systems we give in this chapter.

LCSs are considered as a special kind of production system [34, 80] or as methods for genetics-based machine learning (GBML) [62], evolutionary reinforcement learning (ERL) or evolutionary algorithms for reinforcement learnings (EARLs) [80], or policy optimization [48]. It is typically assumed that LCSs utilize a GA and build a set of if-then rules. Fitness is calculated based on individual rule statistics such as accuracy or strength [8, 62]. While it is unclear whether evolutionary algorithms (EAs) are the only approach qualified to be used in an LCS [62, 80], the importance of the solution representation and the applied metaheuristic operators are quite clear. In particular, the operators for the generational replacement and the mutation need to be adapted for usage in LCSs [34]. Still, GAs in LCSs are said to be similar to the ones applied to search and optimization problems [34].

3.3 ML Systems Similar to LCSs

LCS models, as presented in the previous section, have some similarities with other well-known ML frameworks. We now shortly review them informally[1] and the differences to LCSs in order to better highlight the LCS approach—especially in light of the rather loose definition of what constitutes an LCS and the many existing variants. To this end, we now differentiate LCSs from decision trees (DTs), mixture of experts (MoE) systems, genetic programming (GP) as well as the ensemble learning techniques *bagging* and *boosting*.

3.3.1 Decision Trees

Decision trees are amongst the most widely known rule-based systems. They are often applied to both regression and classification tasks and can be either induced from data automatedly or manually constructed by domain experts. A typical DT divides the input space into a set of disjoint regions; these are created by repeatedly splitting the input space axis-parallelly. The resulting hierarchy defines

[1] At the time of this writing, a more formal treatment of the differences is under active investigation.

a tree structure of which each path from the root to a leaf node corresponds to one region. For each such region, a submodel is fitted (and, by convention, stored in the corresponding leaf); for example, DTs for classification often simply use a constant function corresponding to the majority class of the training data lying in that region. Where to split the input space (i. e. in which dimension and at what value and at what level of the tree) poses an optimization problem that can be solved in many ways. Famous (and widely used) traditional algorithms to construct DTs include CART [15] and C4.5 [61]. Besides these and other related methods, metaheuristics for tree building have been investigated over the years as well: For example, [87] use ant colony optimization (ACO) to construct trees similarly to CART, [86] propose an EA-based hyperheuristic to design heuristics that then construct DTs and [89] introduce a GA that optimizes a population of DTs based on both their accuracy and size.

There is a direct correspondence between LCSs and DTs: Just like a path from the DT root to a leaf, a rule in an LCS specifies a region and a submodel. The only conceptual difference is that in LCSs, regions are allowed to overlap whereas they are not in classical DT. A DT can thus be transformed into an LCS model straightforwardly without losing information. The other direction can be done as well by extending an LCS model by a new region for each overlap of regions and then splitting regions further such that a proper hierachy among them is achieved. Since regions in LCS models may be placed arbitrarily, it can be expected that a rather large number of regions (and correspondingly, tree nodes) has to be added in order to get a proper DT out of an LCS model. Finally, note, that there are fuzzy DT approaches [10, 42] whose predictive models are very close if not the same as LCS models that use matching by degree—although the way that the model is fitted is fundamentally different.

3.3.2 Mixture of Experts

Mixture of experts is a research direction that developed independently of LCSs [41, 44, 85]. Nevertheless, the resulting models share a lot of similarity: The only differences are that, in MoE, usually,

1. a probabilistic view is taken which provides prediction distributions instead of the point estimates that LCSs models return,
2. submodels are not trained independently,
3. submodels are not localized using matching functions,
4. mixing weights are not constant but depend on the input.

This means that an MoE is more expressive than an LCS[2] since in the latter, localization (matching) essentially decides whether the submodel's output is multiplied by 0 or by a constant mixing weight whereas in MoE more than these two values can occur when mixing. On the other hand, LCSs are inherently more interpretable since binary decisions as the one mentioned before are much more comprehensible than decisions on somewhat arbitrary values. Note that MoE-like models whose mixing weights do not depend at all on the input are typically called *unconditional mixture models* [12]. Therefore, typical LCSs can be seen as in-between unconditional mixtures and MoE models in terms of both interpretability and expressiveness. Finally, independent submodel training inherently results in a slightly worse model performance in areas of submodel overlap [26]; thus an MoE can be expected to outperform an equivalent LCS. However, independent submodel training has two major advantages that may actually lead to an improved performance given the same amount of compute: Model structure search is much more efficient (e. g. when changing a single matching function, only the corresponding rule and no others have to be refitted) and, for some forms of submodels, no local optima during submodel fitting arise [26].

There are currently two MoE-inspired formulations of LCSs, one by Drugowitsch [26] and one by [88]. Drugowitsch [26] creates an LCS for regression and classification by adding matching to the standard MoE model. The resulting fully Bayesian probabilistic model is fitted using variational Bayesian inference and agnostic of the employed model structure search (Drugowitsch explores GAs as well as markov chain monte carlo (MCMC) methods). Other than in typical MoEs, the submodels are trained independently (like in an LCS), resulting in the dis-/advantages discussed in the previous paragraph. The resulting model is probabilistic, that is, it provides for any possible input a probability distribution over all possible outputs; to our knowledge, no other LCS is currently able to do this.

[88] take a less general approach and closely model UCS, an LCS for classification, using an MoE. They are able to provide a simpler training routine than Drugowitsch since their system is only capable of dealing with binary inputs (just like the original UCS) and they always model all possible rules. The latter makes training infeasible in high-dimensional space which is why, in a follow-up paper [27], they extend their system with a GA for performing model selection. They also add iterative learning to their model [27] which is not supported by Drugowitsch's system as of yet.

[2] At least as long as the LCS uses the typical binary matching functions. A matching-by-degree LCS can actually be more expressive than a comparable MoE.

3.3.3 Bagging and Boosting

The two most well-known ensemble learning techniques are bagging [14] and boosting [32].

Bagging [14] means generating several bootstrap data sets from the training data and then training on each of them one (weak) learner. In order to perform a prediction for a certain input, the predictions of all the available weak learners for that input are averaged (regression) or combined using majority voting (classification). This procedure has been shown to decrease prediction error if the weak learners have instabilities; for example, this is the case for DTs, neural networks (NNs) and RBML [24]. On first glance, the set of weak learners is somewhat akin to the set of submodels in LCSs. However, there are several major differences: Other than in bagging, in LCSs,

- no bootstrapping is performed to assign data points to the submodels; instead a good split of the data set is *learned* by performing model selection.
- the learned splits do not only influence training but also prediction.
- submodel predictions are mixed based on some quality measure and not using an unweighted average.

Boosting (e. g. the prominent AdaBoost algorithm [32]) trains a set of weak learners (submodels) sequentially and changes the error function used in training after each trained submodel based on the performance of the previous submodels. To perform predictions, the submodels are combined using weighted averaging (regression) or weighted majority voting (classification). LCSs differ from boosting by

- submodels being localized, they do not model all inputs.[3]
- submodels being trained independently of each other.
- directly optimizing the function that combines submodels. While Boosting's repeatedly changed error function is somewhat reminiscent to the interaction of matching and mixing in LCSs, matching and mixing are optimized more explicitly.

3.3.4 Genetic Programming

Another well-known symbolic method applicable to many different ML tasks is genetic programming [45], the evolution of syntax trees that represent functions or programs. When compared to LCSs, GP approaches typically have more degrees of

[3] If matching by degree is used (which is not the case for any of the promiment LCSs), they are softly localized. This means that inputs are somewhat weighted—which goes more in the same direction as boosting.

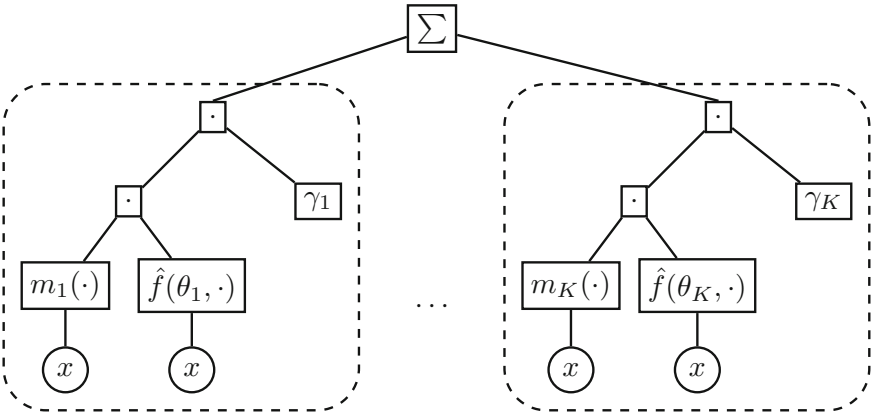

Fig. 3.1 Direct transformation of an LCS model (cf. Eq. (3.1)) to a GP tree. We summarized the nested summation of the submodel outputs as a single large sum. Input nodes are circular

freedom: While an LCS model can be transformed directly into a GP syntax tree (e. g. see Fig. 3.1), the other direction is not that straightforward since GP trees in general do not need to adhere to the general form of an LCS model. Other than in LCSs, there is no explicit generation of submodels in GP.

There are several approaches that use GP techniques in LCSs. For example, [40] use small GP trees (so-called *code fragments*) as conditions to more efficiently explore model structure space by reusing building blocks from previous training on subproblems.

3.4 The Role of Metaheuristics in LCSs

This section discusses the role of metaheuristics in LCS learning more in depth. Section 3.2 already established that metaheuristics are almost always used to solve the task of model selection, that is, subtasks (II) (choosing matching functions) and (I) (choosing an appropriate number of rules), where subtask (I) is sometimes approached heuristically. We now first introduce the options for the general structure of the metaheuristic process and then continue with analysing the used representations, operators and fitness functions.

3.4.1 Four Types of LCSs

In the early years of GBML, there emerged two "schools" with which many of the investigated systems have later been associated: the Michigan and Pittsburgh

approaches [22]. The classical definitions of these two terms are [22, 30]:

Pittsburgh-style systems are methods using a population-based metaheuristic (e. g. a GA) at the level of complete solutions to the learning tasks. They thus hold a *population of rule sets* whose conditions they diversify and intensify; the operators of the metaheuristic operate at the level of entire sets of conditions.

Michigan-style systems on the other hand consider a single solution (a single rule set) as a population on whose conditions a metaheuristic operates. The operators of the employed metaheuristic *operate on individual conditions*.

We deem this differentiation as slightly problematic. For one, there are many systems that cannot quite be sorted into one of the two classes (see, for example, the systems marked as "hybrid" in the list in [77]). Secondly, the two terms focus on population-based approaches. However, what about an approach that uses a metaheuristic such as simulated annealing (SA) for model structure search and gradient-based local optimization for model fitting? Since that metaheuristic works at the level of complete solutions to the problem, this may be seen as a Pittsburgh-style system. On the other hand, only a single solution is considered at any time and individual rules of that solution are changed by the metaheuristic—therefore it could also be called a Michigan-style system.

We thus propose a different distinction based on two major design decisions that greatly influence how exactly a certain LCS, or, more general, an RBML algorithm works:

Training data processing *Online* algorithms update their model after each data point whereas *batch* algorithms process the entire available data at once [12, p. 143].

Model structure search LCSs differ greatly if a single model structure is considered at a time or more than one. We thus introduce the differentiation between *single-model* and *multi-model* LCSs.

These two dimensions define four base types of RBML algorithms (online single-model, online multi-model, batch single-model and batch multi-model systems). The proposed terms also allow for gradual classification (e. g. mini-batch approaches lie somewhere between batch and online methods) and thus resolve the issue of having a category of "hybrid" systems that hold everything that does not fall into one of the categories. Finally, the terms also have a meaning outside of the RBML community which increases transparency and eases entry to the field.

Table 3.1 gives a quick overview of several well-known LCSs and how they relate to the two design decisions. Note that according to our definitions, most systems that were said to be Michigan-style belong to the class of *online single-solution systems* whereas most systems that were said to be Pittsburgh-style belong to the class of *batch multi-solution systems*. XCS and its derivatives [11, 82, 84] are online learners using a typical Michigan-style metaheuristic; they therefore fall into the category of online single-model methods. BioHEL [7] on the other hand performs batch learning but does only ever consider a single set of rules. GAssist [6, 29] is a classical Pittsburgh-style system that considers more than one set of rules at any

Table 3.1 Examples for the different types of LCS algorithms

	Online	Batch
Single-model	XCS(F) [82, 84], UCS [11]	BioHEL [7]
Multi-model		GAssist [6]

time and performs batch learning. Note that there currently is, to our knowledge, no well-known online multi-model LCSs.

Each of the classes has some unique advantages and disadvantages. Dividing model selection and model fitting is easier to achieve for batch algorithms; this makes these systems often simpler and more straightforward to analyse formally than online systems [26]. Particularly, existing population-based single-model systems (e. g. XCS, where the GA operates on individual rules) are often hard to analyse formally due to the rules both competing for a place in the population but also cooperating to form a good global model.

It can be argued that the overall training structure of batch multi-solution systems has a comparably high computational cost since for each model structure considered, the submodels are fitted until convergence—whereas online single-solution systems can get away with fitting the submodels iteratively until convergence only once. However, online single-solution systems perform fitting steps alternatingly with model selection steps which makes the comparison slightly unfair as this can be expected to amount to significantly more fitting steps required until convergence than if the model structure were held fixed. Indeed, given the enormous amount of training examples provided regularly to many state-of-the-art Michigan-style LCSs [47, 53, 68], the comparison may actually not be as unfavourable as it looks at first glance. However, this hypothesis has to our knowledge not yet been investigated in depth.

3.4.2 Metaheuristic Solution Representation

Metaheuristics optimize the set of matching functions and, at that, operate on the set of conditions (useful representations of matching functions, cf. Sect. 3.2).

Early LCSs were designed mostly for binary input domains and these systems are still among the most-used and researched ones [77]. Matching functions for these domains are typically represented by ternary strings, that is, binary strings extended by an additional symbol #, the so-called *wild card*, that represents any of the other two options and thus enables generalization. An example for the representation of a matching function $m : \{0, 1\}^5 \rightarrow \{0, 1\}$ is

$$(1, 1, 0, 1, \#). \tag{3.2}$$

It assigns 1 to (matches) the inputs $(1, 1, 0, 1, 0)$ and $(1, 1, 0, 1, 1)$ and 0 to any other inputs [76, 82].

For real-valued or mixed integer problem domains, many different representations have been proposed. Among the simplest that are commonly applied are hypercuboid [83] and hyperellipsoid [18] conditions. An example for a hypercuboid condition for a matching function $m : \mathbb{R}^3 \to \{0, 1\}$ is the 3-dimensional interval $[l, u)$ (with $l, u \in \mathbb{R}^3$) which can be seen as a tuple

$$(l_1, u_1, l_2, u_2, l_3, u_3). \tag{3.3}$$

It matches all $x = (x_1, x_2, x_3) \in \mathbb{R}^3$ that fulfill[4] $l \leqslant x < u$:

$$m(x) = \begin{cases} 1, & l \leqslant x < u \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

Aside from the mentioned ones, several more complex function families have been proposed, e. g. neural networks [17] and GP-like code fragment graphs [40] (also referred to in Sect. 3.3.4).

In general, any representation is possible including composites or combinations of other representations as long as appropriate operators can be defined. It has to be kept in mind, however, that more complex representations often lead to more complex operators being required as well as that both the size and the topology of the search space is directly influenced by the representation (e. g. there are functions $[0, 1]^5 \to \{0, 1\}$ that cannot be represented by above-introduced representation of ternary strings of length 5). Furthermore, too simplified or restricted encodings of the feature space can result in parts of the matching functions space being inacessible. Variable length representations may alleviate some of these problems but can render the operator design more difficult [76].

3.4.3 Metaheuristic Operators

Metaheuristic operators need to not only be compatible with the chosen form of conditions but also perform well in optimizing the corresponding sets of matching functions. This can be a challenging task for ML practitioners without a strong metaheuristic background, in turn causing the same operators to be used repeatedly, regardless of whether they may actually be suboptimal.

An important aspect in LCSs is *when* the metaheuristic is invoked and whether it operates on the entire set of conditions or on a subset. For instance, in XCS [82], for each input provided to the system, the GA applies its operators only to the conditions of a subset of rules; namely to the ones that matched the input seen last and of those only the ones that also proposed the action taken. Due to the dependence on the

[4] This is equivalent to $(l_1 \leqslant x_1 < u_1) \wedge (l_2 \leqslant x_2 < u_2) \wedge (l_3 \leqslant x_3 < u_3)$.

input introduced by only considering matching and used rules, the GA may operate on a different subset in the very next iteration.

The initial set of conditions is usually created at random, possibly slightly directed by requiring the corresponding matching functions to match certain inputs (matching functions that do not match any of the training data may not be that useful since their merit cannot be estimated properly) [76, 82].

For the generation of new individuals from existing ones, existing LCSs use both recombination as well as mutation operators [76, 77]. Recombination operators in single-solution systems may exchange condition attributes (e. g. hypercuboid boundaries) whereas, in multi-solution systems, they probably should also include an option to recombine sets of conditions in a meaningful way (e. g. exchanging entire conditions between sets of condition).

Optimizing matching functions poses a difficult problem if the training data is sparse, or, more generally, if there are sparsely sampled parts of the input space: On the one hand, changing a condition only leads to a detectable difference in accuracy-based fitness if that change alters the subset of the training data that is matched by the corresponding matching function. On the other hand, these differences, if occurring, can be very large (e. g. if a rule now matches three training examples while, before, it only had matched two). This means that, depending on the training data and initialization, the operators may have a *low locality* if fitness computation only takes into account accuracy statistics on the training data. As a result, areas between training data points may not be covered by solution candidates because there is no fitness signal when exploring having some rules match them. Choosing a combination of suitable operators and a fitness measure that present a consistent answer to this issue is an open problem; the common workaround is to simply rely on comparably large amounts of training data.

While multi-solution systems can explore different solution sizes rather naturally, single-solution systems require explicit mechanisms to control and optimize condition/rule set size. A popular option for population-based single-solutions (used, e. g., in XCS [82]) is a simple heuristic: There is a maximum number of rules (a hyperparameter) that, when violated by rule generation mechanisms, gets enforced by deleting rules based on roulette wheel or tournament selection. Aside from that, the *numerosity* mechanism is typically employed (also used, e. g., in XCS [82]): If there is a well-performing rule r_1 that is responsible for more inputs than another rule r_2 but the inputs matched by r_2 are already covered by r_1, then r_2 may be replaced by a copy of r_1 (typically, no real copies are used but instead a count associated with each rule in the set). The set of conditions thus contains one less unique condition.

In existing online single-solution systems, metaheuristic operators gradually change parts of conditions of the rule set in a steady-state fashion. At that, the central challenges are maintaining a healthy diversity and identifying the required niches—and keeping them in the rule set [76]. Selection is usually based on either roulette wheel or tournament selection from either the whole set of rules (esp. in earlier LCSs [77]) or subsets (e. g. in XCS [82]), the latter promoting niching. Niching is also promoted when performing rule discovery for examples that are not yet

matched by any rule in the set. The mutation operator modifies a single condition; how this occurs primarily depends on the specific representation but is typically stochastic while balancing generalization and specialization pressures. Commonly used operators include bitflip [82] and Gaussian mutation [26]. Recombination also works at the condition level and is usually a single-point, two-point or uniform crossover with encoding-dependent crossover points. The replacement of rules usually employs elitism operators. [76]

Batch multi-solution systems are more similar to other well known optimization approaches [76]. Most existing systems of this category are generational rather than steady-state. Unlike in existing online single-solution systems, parents are condition sets and not individual conditions and are selected from a population of condition sets, typically using roulette wheel or (primarily in later systems) tournament selection. The mutation operator mutates at two levels: at the level of condition sets by adding and removing rules, as well as at the level of individual conditions using a method appropriate for the rule representation, for example, Gaussian mutations of all bounds when using an interval-based condition. The recombination operator mostly exchanges rules between rule sets but can also be extended to additionally exchange parts of individual conditions [6].

3.4.4 Typical Fitness Functions

As already noted in Sect. 3.2.3, the problem that an LCS's metaheuristic tries to solve is model structure selection, that is, choosing the size of the rule set (subtask (I)) and proper matching functions (subtask (II)). The goal of this optimization task is to enable the model to be optimal after it has been fitted. At that, optimality of the model, and with it, optimality of the model selection, is typically defined slightly handwavy based on the fitness measure used for model selection. That fitness measure commonly weighs a high accuracy of overall system predictions against a low model structure complexity (i. e. number of rules) [6, 82]. However, there are also lesser-known, but more principled approaches to what an LCS is meant to learn that we cannot expand on here for the sake of brevity; for example, the one by Drugowitsch [26] based on Bayesian model selection.

The need for high accuracy and a low number of rules induces a multi-objective optimization problem with conflicting goals (the highest accuracy can consistently be achieved with a very high number of rules, e. g., one rule per training example). However, the utilized fitness functions are not always modelled explicitly multi-objectively but often use a (weighted) sum of the objectives—or even focus on only one of them. The exact fitness computation within the system strongly influences the metaheuristic to be used; therefore, we will shortly describe the different options and their implications.

Online single-model systems usually incorporate niching and thus require mechanisms for fitness sharing. In earlier LCSs for RL there was an *explicit fitness sharing* mechanism that split the reward among all rules in the same, activated

niche [39, 81]. The more generally applicable technique is *implicit fitness sharing* which is based on computing the fitness relative to the rules in the same niche and applying metaheuristic operators only within that niche [82]. The fitness functions are usually to be maximized and are often based on either rule strength [81] or rule accuracy [82]. Strength-based fitness is often used in earlier LCSs built for RL settings; it builds on the sum of RL rewards after applying the rule. Accuracy-based fitness, on the other hand, is based on the frequency of the rule's correct predictions; and, due to its increased stability much more common as of now.

In batch multi-model systems, purely accuracy-based fitness functions can result in bloating [26], that is, a significant amount of additional rules being included in the rule set that do not improve the model. To resolve this issue, multi-objective fitness functions with the second objective being the reduction of rule set size are used. They are often modelled as weighted sums of the individual objectives and thus still remaining a single-objective problem. One example for this is the utilization of the minimum description length (MDL) principle for the fitness function in BioHEL [7] and GAssist [6]. MDL is also a common strategy in optimization for feature selection problems.

3.5 Other Metaheuristics-Centric Rule-Based Machine Learning Systems

We next discuss the similarity of LCSs to other metaheuristics-focussed RBML systems, that is, other systems utilizing GAs, ACO, other metaheuristics or hybrids, and artificial immune systems (AISs). These were developed independently of LCSs but are often based on the same ideas (e. g. [39]) or on the work of [30] which summarizes genetic approaches for data mining. Furthermore, they are used to construct if-then rules, mostly applied to classification tasks, and often divided into Michigan and Pittsburgh approaches [30]. The main difference between these metaheuristics-focussed RBML systems and LCSs is that most of them do not utilize any additional bookkeeping parameters.

The list of learning systems in this section is not exhaustive, but aims at providing a broader view with some short examples. It is important to note that we restrict ourselves to systems

- that are more or less close to the definition of an LCS given in Sect. 3.2 but whose authors do not relate them to existing LCS research (or do not explicitly call them LCSs)
- that somewhat lie at the very border of the field of LCS research and may thus not be known well.

While LCSs are commonly associated with evolutionary (or more specifically genetic) algorithms, any metaheuristic can be used [62, 76]. This makes the comparison of these other metaheuristics-focussed RBML systems even more

relevant, as algorithms from both fields can profit from the respective other field's research.

3.5.1 Approaches Based on Evolutionary Algorithms

The degree of resemblance to LCSs is most obvious for approaches which also utilize EAs and of which a non-exhaustive overview is given in this section. Most of these approaches, for example the general description of GAs for data mining and rule discovery by [30], have been proposed with—if at all—only little differentiation or comparison to LCSs.

Approaches utilizing GAs for the discovery and optimization of classification rules are for example described in [1, 21, 50, 51, 79]. These are mostly subsumed under the definition of Michigan-style systems, though many of them are actually batch single-model systems. They differ from LCSs in terms of the operators used in the GA, the fitness computation, or the use of additional strategies. For example, [50] extended their RBML system to perform multi-label classification, while [21] utilize a parallel GA in their approach. A classification rule mining system using a multi-objective GA (MOGA) is presented by [35]. There are also approaches using evolutionary techniques such as co-evolution, which is applied to sets of examples, the rules being induced at the end [43]. This presents an inverse order of the process compared to the traditional LCS approach. Furthermore, other EAs can be used for rule discovery, for example a quantum-inspired differential evolution algorithm [71].

While there often is no direct relation provided between other evolutionary RBML systems and LCSs, at least some summaries describe a few of the different approaches [31] or provide experimental comparisons [74]. An exception is the combination of DTs and a GA by [64], which includes two rule inducing phases (a decision tree produces rules, the GA refines these rules) and which is simultaneously described as a Michigan-style LCS with three phases.

3.5.2 Approaches Based on the Ant System

Another branch of approaches for (classification) rule discovery is based on the ant system, or Ant Colony Optimization (ACO), with the Ant-Miner as the most prominent representative. Their similarity to LCSs is strongly dependent on the variant of LCS and on how much the utilized metaheuristic is seen as a defining component. For example, the Ant-Miner [58] is a batch single-solution system with an overall concept similar to BioHEL [7]. Its pheromone table is similar to the attribute tracking concept in ExSTraCS [76]. Furthermore, it uses the same rule representation strategies as LCSs in general, with the exception that continuous variables are more often discretized in the Ant-Miner. Nevertheless, the ACO

algorithm [25] is quite dissimilar from GAs and the resulting RBML systems can exhibit further differences.

The Ant-Miner develops if-then rules whose condition consists of a concatenation of terms (i. e. *attribute*, *operator*, and *value*). Rules are constructed by probabilistically adding terms to an empty rule under utilization of a problem-dependent heuristic function and the ACO-typical pheromone table. Afterwards, the rule is pruned and the pheromone table is updated and the next rule is constructed. This process is repeated until the maximum population size (number of ants) is reached or the same rule has been constructed more than once. Then, the best rule is selected and the data points it matches and correctly classifies are removed from the training set. The overall algorithm is repeated until enough cases in the training set are covered by the aggregated rule set [58].

There are several extensions and variants for the Ant-Miner [3, 13], for example, different pheromone update functions or heuristic functions and adaptations to cope with continuous data [46, 56]. Furthermore, there also exists a regression rule miner based on Ant-Miner [16] and a batch multi-solution Ant-Miner variant [57]. Also, other ACO-based classifier systems have been developed simultaneously to Ant-Miner [65].

3.5.3 Approaches Based on Other Metaheuristics or Hybrids

Next to GAs and ACO, there are many more metaheuristics and hybrid algorithms that can be utilized in RBML, especially for classification rule mining [23]. They share roughly the same basic view on rules as well as a classification into Michigan- and Pittsburgh-style approaches, although the term Michigan-style often subsumes both online and batch single-model systems. Again, their similarity to LCSs depends strongly on the respective variants and the underlying definitions but a direct integration into existing LCS research is often not provided. While this section can not present these approaches exhaustively, it showcases further insights on how metaheuristics can be applied to RBML.

Particle swarm optimizations (PSOs), for example, has been used for classification rule discovery [66, 67] as well as for a regression rule miner [49]. Furthermore, the artificial chemical reaction optimization algorithm (ACROA) was used to optimize classification rules as well [2]. While these approaches all use population-based metaheuristics, it is not impossible (or infeasible) to use single-solution based optimizers, as was demonstrated in [52] where SA determines fuzzy rules for classification. This SA variant was also extensively compared to the LCSs GAssist and XCSTS.

Hybrid approaches, that is, algorithms combining two different metaheuristics to combine their benefits, are common to classification rule discovery as well. They are, again, often presented as Michigan-style systems; however, many of them perform batch single-model learning. There exist, for example, hybrids of PSO and ACO [37, 38], SA and tabu search (TS) [20], and ACO and SA [63]. Some of these

hybrids explicitly divide their rule discovery process into two phases; this is the case, for example, for the HColonies algorithm, a combination of AntMiner+ and artificial bee colony (ABC) optimization [4]. Additionally, batch multi-model hybrids are possible, as presented by an Ant-Miner-SA combination [54].

Another type of hybrid systems for classification rule mining combines not only two metaheuristics, but utilizes them in what the authors call a Michigan-style phase and a subsequent Pittsburgh-style phase. In our new classification system, these would simply fall into the batch multi-model category. For example, [73] use a combination of a GA and GP, depending on the types of attributes, in the Michigan phase to generate a pool of rules and then perform a Pittsburgh-style optimization with a GA in the second phase to evolve the best rule set from the pool. Similarly, [5] use a hybrid of ACO and a GA. AntMiner+ is used in the first phase to construct several models based on different subsets of the training data, while the GA uses these models as an initial population for optimization. This approach utilizes the smart crossover developed for Pittsburgh-style LCSs, which is an indication for at least some overlap between the two research communities.

3.5.4 Artificial Immune Systems

Artificial immune systems (AISs) are another class of algorithms inspired by biological processes and suitable for ML and optimization tasks. AISs are differentiated by the general strategies they employ, that is, clonal selection theory, immune network theory, negative selection and danger theory. [36, 62, 75]

First of all, the similarity of AIS algorithms and LCSs depends strongly on the strategy. Clonal selection– and negative selection–based AISs are more similar to evolutionary RBML systems than immune network or danger theory AISs. Furthermore, both AISs and LCSs entail many variants, depending on the learning task and implementation choices. At that, for example, solution encoding and operator choice further increases or decreases the similarity between these approaches. Finally, note that AISs research often acknowledges the similarities and differences to LCSs [28, 33].

3.6 Future Work

As demonstrated in the previous sections, LCS research has mostly focussed on the ML side and neglected its metaheuristics to some extent. Nevertheless, there still are shortcomings in terms of the learning aspect: For instance, LCSs continue to show inferior performance to many other systems when it comes to non-trivial RL, although they were originally designed for exactly this kind of task [26, 69]. Their learning performance often depends on a large number of semantically very different and non-trivially interacting system parameters [26]; this complicates

utilization of these systems and often necessitates parameter tuning. Furthermore, there are close to no formal guarantees for most of the popular LCSs, mainly due to the ad-hoc approach taken for their development [26]. As a result, the assumptions these algorithms make about the data are slighly obscure, as are the algorithms' objectives resulting in uncertainty about which learning tasks LCSs are suited for best.

To remedy these shortcomings is one important direction for future research on LCSs. However, questions relating to the metaheuristics side should not be neglected any longer either. This includes starting to utilize state-of-the-art metaheuristic knowledge and techniques for optimization, but also and probably even more understanding the relationship between the problems of LCS model fitting and LCS model selection. An open question is, for example, how the nature of the model fitting problem in LCSs relates to the nature of the corresponding model selection problem and therefore what metaheuristic to use. This begins at picking representations and operators that perform well, and translates to the usage of matching functions and submodels.

3.7 Conclusion

We presented Learning Classifier Systems (LCSs), a family of rule-based machine learning systems. They construct models of overlapping if-then rules from data. Construction of an LCS's model involves two main optimization tasks that each consist of two subtasks. To perform *model selection* LCSs employ metaheuristics to optimize the resulting model's *rules' conditions* (i. e. when a certain rule applies) and often also to determine an *appropriate number of rules* to be used. The task of *model fitting* entails optimizing each of the *rules' submodels* so it fits the data that fulfills the rule's condition; this is typically done non-metaheuristically. In situations where multiple rules' conditions match a certain input a *mixing strategy* is used to determine the model's prediction based on each rule's prediction.

Furthermore, we discussed the similarities and differences of LCSs and the related machine learning techniques of Genetic Programming, Decision Trees, Mixtures of Experts, Bagging and Boosting. We found that these techniques are in fact unique but can often be combined with LCSs or transformed into LCS models by making or dropping assumptions. However, bidirectional transformation is non-trivial.

We introduced a new nomenclature to differentiate LCSs into four distinct types that supersedes the earlier classification into Michigan-style and Pittsburgh-style systems. We categorize them based on their *training data processing* scheme into *online* and *batch* learning and based on whether *model structure search* involves a *single-model* or a *multi-model* approach.

We then focussed on how metaheuristics for model selection are commonly designed: Firstly, discussing various rule condition *representations* in relation to input domains and operators. Secondly, giving an overview about how and when

operators are usually applied, stressing issues such as locality and niching, and which operators are typically used in existing literature, especially with regards to the distinct types of LCSs. Lastly, detailing *fitness function* requirements and typical representatives.

Additionally, we summarized *other metaheuristics-centric rule-based machine learning systems*. Although these systems are not referred to by their authors as LCSs, according to the definitions presented in this chapter, they may actually qualify as such. The most well-known of these systems is probably Ant-Miner which optimizes rule sets using Ant Colony Optimization and whose research community even uses some of the terminology typical to the field of LCS such as distinguishing between Michigan-style and Pittsburgh-style systems.

Overall, we find that both the general metaheuristics community, as well as the LCS community do overlap and are researching similar issues but do not interact enough. Metaheuristics researchers rarely consider the optimization task found in an LCS as an application of their algorithms and LCS researchers often stick to more basic metaheuristics and—especially in the case of those systems that are referred to by their authors as LCSs—solely consider the application of genetic or evolutionary algorithms. Contrastingly, we are certain that *interaction and cooperation* can only benefit the advancement of both fields and that many exciting research questions remain to be answered.

References

1. Basheer M. Al-Maqaleh and Hamid Shahbazkia. A Genetic Algorithm for Discovering Classification Rules in Data Mining. *International Journal of Computer Applications*, 41(18):40–44, mar 2012.
2. Bilal Alatas. A novel chemistry based metaheuristic optimization method for mining of classification rules. *Expert Systems with Applications*, 39(12):11080–11088, sep 2012.
3. Zulfiqar Ali and Waseem Shahzad. Comparative Analysis and Survey of Ant Colony Optimization based Rule Miners. *International Journal of Advanced Computer Science and Applications*, 8(1), 2017.
4. Sarab AlMuhaideb and Mohamed El Bachir Menai. HColonies: a new hybrid metaheuristic for medical data classification. *Applied Intelligence*, 41(1):282–298, feb 2014.
5. Sarab AlMuhaideb and Mohamed El Bachir Menai. A new hybrid metaheuristic for medical data classification. *International Journal of Metaheuristics*, 3(1):59, 2014.
6. Jaume Bacardit. *Pittsburgh genetics-based machine learning in the data mining era: representations, generalization, and run-time*. PhD thesis, PhD thesis, Ramon Llull University, Barcelona, 2004.
7. Jaume Bacardit and Natalio Krasnogor. *BioHEL: Bioinformatics-oriented Hierarchical Evolutionary Learning*. 2006.
8. Thomas Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. CRC Press, jan 1997.
9. Alwyn Barry. The stability of long action chains in XCS. *Soft Comput.*, 6(3–4):183–199, 2002.
10. Marco Barsacchi, Alessio Bechini, and Francesco Marcelloni. An analysis of boosted ensembles of binary fuzzy decision trees. *Expert Systems with Applications*, 154, 2020.

11. Ester Bernadó-Mansilla and Josep M. Garrell-Guiu. Accuracy-Based Learning Classifier Systems: Models, Analysis and Applications to Classification Tasks. *Evolutionary Computation*, 11(3):209–238, 09 2003.
12. Christopher M. Bishop. *Pattern recognition and machine learning, 8th Edition*. Information science and statistics. Springer, 2009.
13. Urszula Boryczka and Jan Kozak. New Algorithms for Generation Decision Trees—Ant-Miner and Its Modifications. In *Studies in Computational Intelligence*, pages 229–262. Springer Berlin Heidelberg, 2009.
14. Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, 1996.
15. Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification And Regression Trees*. Routledge, October 1984.
16. James Brookhouse and Fernando E. B. Otero. Discovering Regression Rules with Ant Colony Optimization. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, jul 2015.
17. Larry Bull and Jacob Hurst. A neural learning classifier system with self-adaptive constructivism. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 2, pages 991–997, 2003.
18. Martin V. Butz, Pier Luca Lanzi, and Stewart W. Wilson. Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction. *IEEE Transactions on Evolutionary Computation*, 12(3):355–376, 2008.
19. Martin V. Butz and Wolfgang Stolzmann. An algorithmic description of ACS2. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, pages 211–229, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
20. Ivan Chorbev, Boban Joksimoski, and Dragan Mihajlov. SA Tabu Miner: A hybrid heuristic algorithm for rule induction. *Intelligent Decision Technologies*, 6:265–271, 2012.
21. Dieferson L. Alves de Araujo, Heitor S. Lopes, and Alex A. Freitas. A parallel genetic algorithm for rule discovery in large databases. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*. IEEE, 1999.
22. Kenneth DeJong. Learning with genetic algorithms: An overview. *Mach. Learn.*, 3:121–138, 1988.
23. Clarisse Dhaenens and Laetitia Jourdan. Metaheuristics for data mining. *4OR*, 17(2):115–139, apr 2019.
24. Thomas G. Dietterich. Ensemble methods in machine learning. In *Multiple Classifier Systems*, pages 1–15, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
25. Marco Dorigo and Thomas Stützle. *Ant colony optimization*. MIT Press, Cambridge, Mass, 2004.
26. Jan Drugowitsch. *Design and Analysis of Learning Classifier Systems - A Probabilistic Approach*, volume 139 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
27. Narayanan Unny Edakunni, Gavin Brown, and Tim Kovacs. Online, GA based mixture of experts: a probabilistic model of UCS. In Natalio Krasnogor and Pier Luca Lanzi, editors, *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12–16, 2011*, pages 1267–1274, New York, NY, USA, 2011. ACM.
28. J. Doyne Farmer, Norman H. Packard, and Alan S. Perelson. The immune system, adaptation, and machine learning. *Physica D: Nonlinear Phenomena*, 22(1–3):187–204, oct 1986.
29. María A. Franco, Natalio Krasnogor, and Jaume Bacardit. GAssist vs. BioHEL: critical assessment of two paradigms of genetics-based machine learning. *Soft Comput.*, 17(6):953–981, 2013.
30. Alex A. Freitas. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer Berlin Heidelberg, 2002.
31. Alex A. Freitas. A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery. In *Natural Computing Series*, pages 819–845. Springer Berlin Heidelberg, 2003.

32. Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In Lorenza Saitta, editor, *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3–6, 1996*, pages 148–156. Morgan Kaufmann, 1996.
33. Simon M. Garrett. How Do We Evaluate Artificial Immune Systems? *Evolutionary Computation*, 13(2):145–177, jun 2005.
34. David Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Publishing Company, Reading, Mass, 1989.
35. Preeti Gupta, Tarun Kumar Sharma, Deepti Mehrotra, and Ajith Abraham. Knowledge building through optimized classification rule set generation using genetic based elitist multi objective approach. *Neural Computing and Applications*, 31(S2):845–855, may 2017.
36. Emma Hart and Jon Timmis. Application areas of AIS: The past, the present and the future. *Applied Soft Computing*, 8(1):191–201, jan 2008.
37. Nicholas Holden and Alex A. Freitas. A hybrid particle swarm/ant colony algorithm for the classification of hierarchical biological data. In *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.* IEEE, 2005.
38. Nicholas Holden and Alex A. Freitas. A hybrid PSO/ACO algorithm for discovering classification rules in data mining. *Journal of Artificial Evolution and Applications*, 2008:1–11, may 2008.
39. John H. Holland. Adaptation. *Progress in theoretical biology*, 4:263–293, 1976.
40. Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems. *IEEE Transactions on Evolutionary Computation*, 18(4):465–480, 2014.
41. Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.
42. Cezary Z. Janikow. Fuzzy decision trees: issues and methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 28(1):1–14, 1998.
43. Licheng Jiao, Jing Liu, and Weicai Zhong. An organizational coevolutionary algorithm for classification. *IEEE Transactions on Evolutionary Computation*, 10(1):67–80, feb 2006.
44. Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.
45. John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993.
46. Bo Liu, H. A. Abbas, and B. McKay. Classification rule discovery with ant colony optimization. In *IEEE/WIC International Conference on Intelligent Agent Technology, 2003. IAT 2003.* IEEE Comput. Soc, 2003.
47. Yi Liu, Will N. Browne, and Bing Xue. Absumption and subsumption based learning classifier systems. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 368–376, New York, NY, USA, 2020. Association for Computing Machinery.
48. Sean Luke. *Essentials of metaheuristics: a set of undergraduate lecture notes*. Lulu Com, 2013.
49. Bart Minnaert and David Martens. Towards a Particle Swarm Optimization-Based Regression Rule Miner. In *2012 IEEE 12th International Conference on Data Mining Workshops*. IEEE, dec 2012.
50. Thiago Zafalon Miranda, Diorge Brognara Sardinha, Mrcio Porto Basgalupp, Yaochu Jin, and Ricardo Cerri. Generation of Consistent Sets of Multi-Label Classification Rules with a Multi-Objective Evolutionary Algorithm, March 2020. https://arXiv.org/abs/2003.12526.
51. Thiago Zafalon Miranda, Diorge Brognara Sardinha, and Ricardo Cerri. Preventing the Generation of Inconsistent Sets of Classification Rules, August 2019. https://arXiv.org/abs/1908.09652.
52. Hamid Mohamadi, Jafar Habibi, Mohammad Saniee Abadeh, and Hamid Saadi. Data mining with a simulated annealing based fuzzy classification system. *Pattern Recognition*, 41(5):1824–1833, may 2008.

53. Masaya Nakata, Will N. Browne, Tomoki Hamagami, and Keiki Takadama. Theoretical XCS parameter settings of learning accurate classifiers. In Peter A. N. Bosman, editor, *Proceedings of the Genetic and Evolutionary Computation Conference 2017*, GECCO '17, pages 473–480, New York, NY, USA, 2017. ACM.
54. Bijaya Kumar Nanda and Satchidananda Dehuri. Ant Miner: A Hybrid Pittsburgh Style Classification Rule Mining Algorithm. *International Journal of Artificial Intelligence and Machine Learning*, 10(1):45–59, jan 2020.
55. Romain Orhand, Anne Jeannin-Girardon, Pierre Parrend, and Pierre Collet. PEPACS Integrating probability-enhanced predictions to acs2. GECCO '20, New York, NY, USA, 2020. Association for Computing Machinery.
56. Fernando E. B. Otero, Alex A. Freitas, and Colin G. Johnson. cAnt-Miner: An Ant Colony Classification Algorithm to Cope with Continuous Attributes. In *Ant Colony Optimization and Swarm Intelligence*, pages 48–59. Springer Berlin Heidelberg, 2008.
57. Fernando E. B. Otero, Alex A. Freitas, and Colin G. Johnson. A New Sequential Covering Strategy for Inducing Classification Rules With Ant Colony Algorithms. *IEEE Transactions on Evolutionary Computation*, 17(1):64–76, feb 2013.
58. Rafael S. Parpinelli, Heitor S. Lopes, and Alex A. Freitas. An Ant Colony Algorithm for Classification Rule Discovery. In *Data Mining*, pages 191–208. IGI Global, 2002.
59. David Pätzel, Michael Heider, and Alexander R. M. Wagner. An overview of LCS research from 2020 to 2021. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '21, pages 1648–1656, New York, NY, USA, 2021. Association for Computing Machinery.
60. David Pätzel, Anthony Stein, and Masaya Nakata. An overview of LCS research from IWLCS 2019 to 2020. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, pages 1782–1788, New York, NY, USA, 2020. Association for Computing Machinery.
61. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
62. Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors. *Handbook of Natural Computing*. Springer Berlin Heidelberg, 2012.
63. Rizauddin Saian and Ku Ruhana Ku-Mahamud. Hybrid Ant Colony Optimization and Simulated Annealing for Rule Induction. In *2011 UKSim 5th European Symposium on Computer Modeling and Simulation*. IEEE, nov 2011.
64. Bikash Kanti Sarkar, Shib Sankar Sana, and Kripasindhu Chaudhuri. A genetic algorithm-based rule extraction system. *Applied Soft Computing*, 12(1):238–254, jan 2012.
65. P. S. Shelokar, V. K. Jayaraman, and B. D. Kulkarni. An ant colony classifier system: application to some process engineering problems. *Computers & Chemical Engineering*, 28(9):1577–1584, aug 2004.
66. Tiago Sousa, Arlindo Silva, and Ana Neves. A Particle Swarm Data Miner. In *Progress in Artificial Intelligence*, pages 43–53. Springer Berlin Heidelberg, 2003.
67. Tiago Sousa, Arlindo Silva, and Ana Neves. Particle swarm based data mining algorithms for classification tasks. *Parallel Computing*, 30(5–6):767–783, may 2004.
68. Anthony Stein. *Interpolation-Assisted Evolutionary Rule-Based Machine Learning - Strategies to Counter Knowledge Gaps in XCS-Based Self-Learning Adaptive Systems*. doctoralthesis, Universität Augsburg, 2019.
69. Anthony Stein, Roland Maier, Lukas Rosenbauer, and Jörg Hähner. XCS classifier system with experience replay. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 404–413, New York, NY, USA, 2020. Association for Computing Machinery.
70. Anthony Stein, Simon Menssen, and Jörg Hähner. What about interpolation? a radial basis function approach to classifier prediction modeling in XCSF. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 537–544, New York, NY, USA, 2018. Association for Computing Machinery.

71. Haijun Su, Yupu Yang, and Liang Zhao. Classification rule discovery with DE/QDE algorithm. *Expert Systems with Applications*, 37(2):1216–1222, mar 2010.
72. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction. Second Edition*. MIT Press, Cambridge, MA, 2018.
73. K. C. Tan, Q. Yu, C. M. Heng, and T. H. Lee. Evolutionary computing for knowledge discovery in medical diagnosis. *Artificial Intelligence in Medicine*, 27(2):129–154, feb 2003.
74. Ajay Kumar Tanwani and Muddassar Farooq. Classification Potential vs. Classification Accuracy: A Comprehensive Study of Evolutionary Algorithms with Biomedical Datasets. In *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2010.
75. Jon Timmis, Paul Andrews, Nick Owens, and Ed Clark. An interdisciplinary perspective on artificial immune systems. *Evolutionary Intelligence*, 1(1):5–26, jan 2008.
76. Ryan J. Urbanowicz and Will N. Browne. *Introduction to Learning Classifier Systems*. Springer Briefs in Intelligent Systems. Springer, 2017.
77. Ryan J. Urbanowicz and Jason H. Moore. Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, 2009, 2009.
78. Ryan J. Urbanowicz and Jason H. Moore. ExSTraCS 2.0: description and evaluation of a scalable learning classifier system. *Evolutionary Intelligence*, 8(2):89–116, Sep 2015.
79. A. H. C. van Kampen, Z. Ramadan, M. Mulholland, D. B. Hibbert, and L. M. C. Buydens. Learning classification rules from an ion chromatography database using a genetic based classifier system. *Analytica Chimica Acta*, 344(1–2):1–15, may 1997.
80. Thomas Weise. *Global optimization algorithms-theory and application*. Self-Published Thomas Weise, 2009.
81. Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.
82. Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
83. Stewart W. Wilson. Get real! xcs with continuous-valued inputs. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems*, pages 209–219, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
84. Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2):211–234, 6 2002.
85. Seniha Esen Yuksel, Joseph N. Wilson, and Paul D. Gader. Twenty years of mixture of experts. *IEEE Transactions on Neural Networks and Learning Systems*, 23(8):1177–1193, 2012.
86. Rodrigo C. Barros, Márcio P. Basgalupp, André C.P.L.F. de Carvalho, and Alex A. Freitas. A hyper-heuristic evolutionary algorithm for automatically designing decision tree algorithms. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation - GECCO '12*. ACM Press, 2012.
87. Urszula Boryczka and Jan Kozak. Ant Colony Decision Trees – A New Method for Constructing Decision Trees Based on Ant Colony Optimization. In *Computational Collective Intelligence. Technologies and Applications*, pages 373–382. Springer Berlin Heidelberg, 2010.
88. Narayanan Unny Edakunni, Tim Kovacs, Gavin Brown, and James A. R. Marshall. Modeling ucs as a mixture of experts. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1187–1194, New York, NY, USA, 2009. ACM.
89. Vili Podgorelec, Matej Šprogar, and Sandi Pohorec. Evolutionary design of decision trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(2):63–82, 2012.
90. Kreangsak Tamee, Larry Bull, and Ouen Pinngern. Towards clustering with XCS. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1854–1860, New York, NY, USA, 2007. Association for Computing Machinery.