Learning Classifier Systems

Victor Montiel Argaiz

MsC in Artificial Intelligence
U.N.E.D.
Madrid, Spain

victor.montielargaiz@gmail.com

May 31st, 2012

Contents

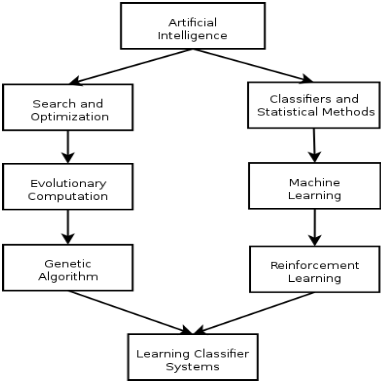
Introduction to LCS

Introduction to Learning Classifier Systems

- Learning Classifier Systems (LCS) are *machine learning* methods which use *evolutionary computation* techniques to create a set of rules or *classifiers* aimed to perform the best action given an input from the environment.
- As other *machine learning* methods, LCS are adaptive systems, i.e., they have the capacity to change and learn from previous experiences according to a *reward or credit assigment scheme*.

LCS within AI techniques

LCS make use of *Evolutionary Computing* techniques to generate new random knowledge, and *Reinforced Learning* to choose, amongst the generated knowledge, the best fitted to the environment. Because of this mixture of techniques they are also referred as *Genetic-based Machine Learning systems* [1].



Componentes of a LCS, I

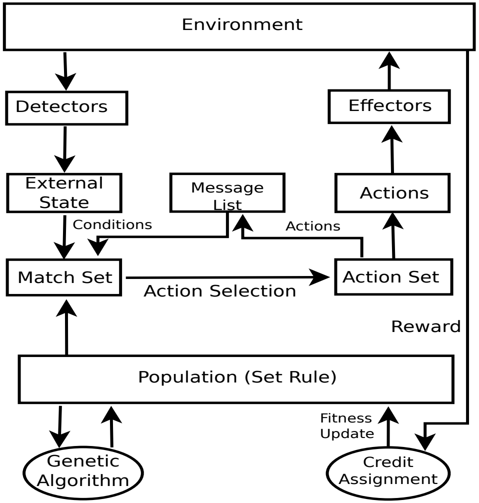
There exist many different implementations of LCS, all having some common components making possible the *discovery* of new knowledge and *learning* of the useful one:

- *Environment Interface*: Detectors to receive inputs from the environment and effectors to perform actions
- Population of *rules* or *classifiers*, containing the current knowledge, codified according to a certain genotype depending on the problem to be solved (decision trees, binary values, real values, intervals...).
- *Discovery* component, to create new rules and improve the existing ones. Implemented through Genetic Algorithms

Components of a LCS, II

- *Message List* component, to introduce, as a way of feedback, the internal state of the system as another input, being able to behave as a memory mechanism.
- *Performance* component, which selects the best rules and decides the action to perform. It is made up of the *Matching set*, the *Action set* and eventually a *prediction* component.
- *Reinforcement* component, which, given the reward obtained for an action, updates the fitness of each rule according to some credit assigment algorithm such as *bucket brigade* or *Q-Learning-alike* methods.

Components of a LCS, III



Discovery and learning within LCS

Discovery and learning are the two main modules behind LCS. The first one creates new random knowledge through rules, the second one retains the best rules.

Discovery of new rules

Rule discovery is implemented usually with Genetic Algorithms:

- N classifiers randomly initialized are codified in a genotype using a `condition:action` format (population of rules)
- Each classifier has a *fitness evaluation* according to the results obtained when applying that rule to a given input condition
- Best rules are selected to mate according to their *fitness value*, creating new individuals via *crossover* and *mutation* (creation of new knowledge)
- Best rules amongst parents and offsprings are selected to survive for the next generation, removing those less performing classifiers to keep population size constant.

Common transformation operators (crossover and mutation) are used depending on the genotype representation (binary or real numbers usually). Steady-State selection process are usually implemented.

Michigan and Pittsburgh Styles

- *Michigan-style* LCS have a population of classifiers where the GA operates at the level of individual rules, and the solution is made up of the entire population, all classifiers cooperate to provide a collective solution
- *Pittsburgh-style* LCS have a population of variable length *rule-set*, GA operates at the level of an entire *rule-set* and each *rule-set* is a potential solution. Rule-sets compete amongst them, they do not cooperate to provide a collective solution

Michigan-style LCS are by far more common because of their simplicity, lesser computational requirements and flexibility to be applied to a wider range of problems. They are more fitted for *incremental* learning. *Pittsburgh-style* systems usally employ *off-line* learning to generate the *rule-set*

Learning Methods

Learning I

Learnig is the increasing of performance of the LCS through the acquisition of new and useful knowledge based on experience of past actions.
Depending on the learning environment, learning can be:

- *Offline* or *batch*: All training examples are presented at once to the system, creating a definitive rule-set at the end of the learning process to be used later. Learning and application of the system do not coexist in time
- *Online* or *incremental*: Examples are provided one at a time, and rule-set is continously changing with each new input from the environment. Learning and application of the system coexist in time, improving the performance while it is being used

Learning II

Depending on the feedback supplied to the system, learning can be:

- *Supervised*: For each training example, the solution (correct action) is provided. More usual in *offline* learning.
- *Reinforcement*: The correct answer is not known for the system, but a numerical feedback is provided measuring the correctness of the `condition:action` rule. This feedback provides the hint to know if the rule is bad or good.

Typically, LCS use *online* and *unsupervised reinforced* learning, but this is not the unique configuration.

Learning Methods, Implicit Bucket Brigade

It is one of the first credit assigment scheme used widely by LCS community, specially in the most simple implementations.
After an action has been performed at time t, we proceed to update the fitness value s_r of each rule r according to the following steps:

1. Rules which are not in the *match set* M_t are unchanged: $\forall r \notin M_t, s'_r = s_r$
2. Rules within M_t but not in the *action set* A_t have their fitness values reduced by a mupltiplication factor $\tau \in [0,1)$, $\forall r \in M_t, r \notin A_t, s'_r = s_r \cdot \tau$
3. Rules within A_t have their fitness values reduced by a fraction $\beta \in [0,1)$, $\forall r \in A_t, s'_r = (1-\beta)s_r$

Learning Methods, Implicit Bucket Brigade

4. This proportion β of fitness substracted will be distributed to the members of previous action set A_{t-1}, reduced by a one-period discount factor γ, $\forall r \in A_{t-1}, s''_r = s'_r + \frac{\gamma \sum_{r \in A_t} \beta s_r}{|A_{t-1}|}$
5. Finally, the new reward P_t is distributed to the members of the current action set, reduced by a factor of β, $\forall r \in A_t, s'''_r = s''_r + \beta \frac{P_t}{|A_t|}$

Intuitively, we can think of the fitness update as an economic transaction. Rules will pay a fraction β of their fitness s_r for being selected. After receiving the reward P, it will be allocated within all rules in A_t, that is, they will be paid back for its work. If they get more reward than what they paid to be selected, they end up with positive balance, effectively increasing their fitness value, as a sign they are good rules.

Learning Methods, Q-Learning-alike

Inspiried in the *Q-Learning*, is one of the most important learning methods within the *Reinforcement Learning* arena. Each rule is represented by a five tuple $\langle c : a : p : \epsilon : F \rangle$, where c, a, p are the condition, action and predicted payoff, ϵ the prediction error and F a fitness for the GA.
Let $\mathcal{A}$ the set of actions in rules r within the match set M_t, upon a new action the update of parameters is as follows:

1. For each action in $\mathcal{A}$, calculate a weighted predicted payoff based on the prediction and fitness of each rule $r \in R_a$ which has a as an action. $\forall a \in \mathcal{A}, p_a = \sum_{r \in R_a} p_r \frac{F_r}{\sum_{r \in R_a} F_r}$
2. Selected the action based on this predicted action reward p_a
3. Once we have the reward R, we update the action set rules

Learning Methods, Q-Learning-alike

4. Fitness values are updated using prediction errors ϵ_r and accuracy of the rule $\kappa_r = e^{ln\alpha \cdot \frac{\epsilon_r - \epsilon_0}{\epsilon_0}}$, with a *learning rate* of β, $F'_r = F_r + \beta(\kappa'_r - F_r)$, where κ'_r is a normalized accuracy for all rules of the set $\kappa'_r = \frac{\kappa_r}{\sum\limits_{r \in R_a} \kappa_r}$
5. A Payoff $P = \gamma max\{p_a \backslash a \in \mathcal{A}\} + R$ is calculated based on reward R and maximum prediction for all actions in M_t
6. P is used to update prediction errors: $\epsilon'_r = \epsilon_r + \beta(|P - p_r| - \epsilon_r)$
7. P is also used to update the actual prediction for the rule $p'_r = p_r + \beta(P - p_r)$

Examples of LCS

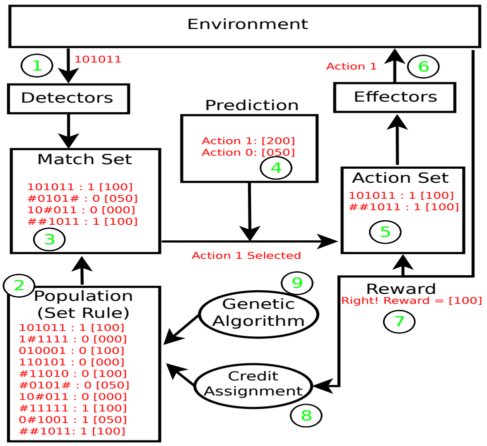
After a review of components and learning methods, some practical implementations of LCS...

An example, Minimal Classifier System (MCS)

Consider the multiplexor example on [2] using a simplified version of LCS, the MCS [3].

- Inputs are defined with 2-bits address types (`a`) and 4-bits input types (`i`):`aa iiii`
- The output (`o`) is a 1-bit type equal to the input pointed by the address bits
- Genotype of classifiers: `input:action` $\Rightarrow$ [`reward`], codified in binary as `aa iiii : o` $\Rightarrow$ [`reward`]. The character # is a wildcard matching any value.

MCS at Work, I


Environment
1
101011
Detectors
Action 1
6
Effectors
Prediction
Action 1: [200]
Action 0: [050]
4
Match Set
101011 : 1 [100]
#0101# : 0 [050]
10#011 : 0 [000]
##1011 : 1 [100]
3
Action Set
101011 : 1 [100]
##1011 : 1 [100]
5
Action 1 Selected
2
Population
(Set Rule)
101011 : 1 [100]
1#1111 : 0 [000]
010001 : 0 [100]
110101 : 0 [000]
#11010 : 0 [100]
#0101# : 0 [050]
10#011 : 0 [000]
#11111 : 1 [100]
0#1001 : 1 [050]
##1011: 1 [100]
9
Genetic
Algorithm
Reward
Right! Reward = [100]
7
Credit
Assignment
8

MCS at Work, II

Steps on the execution:

1. A new input 10 1011 is provided by the environment
2. Look up on the *population* of classifiers for those rules matching the input
3. The *match set* with those rules matching the input condition is created. In brakets we have the current fitness value for each rule
4. Each classifier might have a different action to perform 1 or 0. Predict, using fitness values for each rule, which action would return a higher reward
5. Action 1, with the highest reward, is selected

MCS at Work, III

6. Action 1 is effectively performed on the environment
7. Environment returns some kind of reward P
8. Credit Assigment unit update fitness value of the rules within the action and matching set according to the reward received and the learning scheme.
9. Genetic Algorithm run a generation, mating parents, creating new offsprings, mutating individuals and selecting the best classifiers for next round

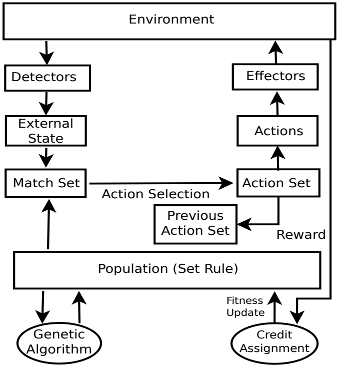
MCS at Work, IV

- MCS is just a simplified LCS, which processes iteratively just one input at a time. In general LCS have message list being able to process several messages, both from environment and feedback from other components of the system at previous time steps
- If match set is empty for a given input, GA randomly creates a new rule covering that input (*covering*), and removes another one, based on its fitness value, to keep population constant
- Offsprings (new rules) usually inherit fitness from their parents

Zeroth Level CS, ZCS

MCS machines were created as a simplified model, but never used in practice. ZCS was one of the first system used.

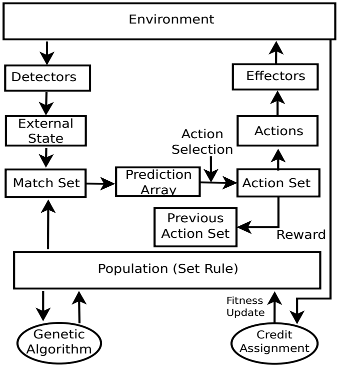
- ZCS do not implement the message list. Inputs are directly matched against the population of rules
- The credit assigment schema is usally the *implicit bucket brigade*
- There is no prediction of each action



eXtended CS, XCS

XCS are still a simplificatoin of the general LCS, and try to solve the drawbacks of the simplicity of ZCS.

- They still do not have internal message list (no memory)
- It uses predicted reward to choose the action
- *Q-learning-alike* learning methods are usually employed



Applications and Conclusions

Applications

LCS have been used on the following domains to solve a wide variety of problems:

- *Function Approximation*: Approximation of complex functions by a set of functions defined on overlapping domains (piecewise definition)
- *Classification Problems*: Pattern recognition, image analysis, medical diagnosis, data mining
- *Reinforcement Learning Problems*: Robot control and navigation, Design optimization (VLSI microchip layering), stock market trading, optimization (traffic-light control).

Drawbacks

Although LCS have been successfully applied to many problems, they have also some drawbacks:

- They do not perform well on sequential decision problems (an in general non-Markovian problems), where several actions for a given input are invoked before getting a reward
- They have a large number of components: GA, Credit Scheme, Message Lists, each one having a set of configuration parameters. A fine tuning of all components working together can be a tough task
- There is no such thing as a standard LCS, instead many different designs exist (MCS, ZCS, XCS...). The lack of rigour and homogeneity on the designs implyies a low acceptance within the machine learning field

Summary

- LCS merge stand-alone techniques, such as *Genetic Algorithm* and *Reinforcement Learning* to give another approach to the learning tools.
- The interactions between these stand-alone modules, and its individual configuration give infinite possible combinations for this learning tool, each adapted to a particular problem
- The complexity of the initial theoretic LCS schema has been solved by creating the simplified models ZCS and XCS, which have been used in practice

Bibliography

Jan Drugowitsch.
Design and Analysis of Learning Classifier Systems: A Probabilistic Approach (Studies in Computational Intelligence).
Springer Publishing Company, Incorporated, 1 edition, 2008.

A. E. Eiben and J. E. Smith.
Introduction to evolutionary computation.
Natural computing series. Springer-Verlag, 2003.

Ryan J. Urbanowicz and Jason H. Moore.
Learning classifier systems: a complete introduction, review, and roadmap.
J. Artif. Evol. App., 2009:1:1–1:25, January 2009.