XCS Framework in Python
Game Research Lab

Andreas Schmidt

***Abstract*— In dieser Arbeit wird das Ergebnis eines Games Research Lab präsentiert. Ziel des Arbeit war die Entwicklung eines Frameworks für das eXtended Learning Classifier System (XCS) in der Programmiersprache Python. Das Framework wurde als erweiterbare Softwarebibliothek realisiert und bietet eine Implementierung eines XCS und dessen Komponenten. Des Weiteren wurden Beispiele und eine Dokumentation erarbeitet.**

I. Einleitung

Bei einem Learning Classifier System (LCS) handelt es sich um ein regelbasiertes Lernverfahren, das von dem Schöpfer des Genetischen Algorithmus [7] John Holland in den 1970er Jahren erstmals erdacht [4][6] und später formalisiert wurde [5]. Das LCS vereint die Bereiche Machine Learning, Evolutionary Computation und Reinforcement Learning, um eine Wissensbasis aufzubauen, die einen intelligenten *decision maker* modelliert. Das LCS zeichnet sich durch eine interpretierbare, regelbasierte Wissensbasis und der Fähigkeit des *Online learnings* aus. Das bedeutet, dass LCS ihr Wissen aufbauen, während sie mit ihrer Umgebung interagieren. Deshalb können mit LCS nicht nur Probleme der Klassifizierung gelöst werden, sondern auch Probleme aus den Bereichen Funktionsapproximation und Reinforcement Learning. Eine interpretierbare Wissensbasis macht LCS besonders für sicherheitskritische Anwendungen besonders interessant. So wurden LCS bereits genutzt, um den Flugverkehr zu regulieren [12] und zur Steuerung von Ampelanlagen [13][1]. Im Bereich Games fanden LCS bisher wenig Anwendung [14]. Auch ist die Anzahl an generischen Implementierungen des XCS als Framework überschaubar. Im Folgenden wird zunächst die Funktionsweise eines LCS knapp erläutert. Anschließend werden verwandte Arbeiten vorgestellt. Danach wird das Framework und dessen Aufbau präsentiert. Die Nutzbarkeit des Frameworks wird danach mit Beispielen demonstriert. Abgeschlossen wird mit einem Ausblick auf zukünftige Arbeiten.

II. LCS Funktionsweise

Im Folgenden wird der Aufbau und die Funktionsweise eines LCS knapp dargestellt. Ein LCS interagiert mit einer Umgebung, indem es über *Effektoren* Aktionen ausführt und den aktuellen Zustand über *Detektoren* wahrnimmt (vgl. Abb. 1). Die Wissensbasis eines LCS besteht aus Regeln der Form *"WENN Bedingung DANN Aktion"*. Ein Classifier kapselt eine solche Regel und besitzt weitere Attribute, die den Classifier näher beschreiben. Die Wissensbasis eines LCS besteht aus einer Menge von Classifier und wird als Population ([P]) bezeichnet. Diese Art der Population wird als Michigan-Style bezeichnet und wurde im Rahmen der

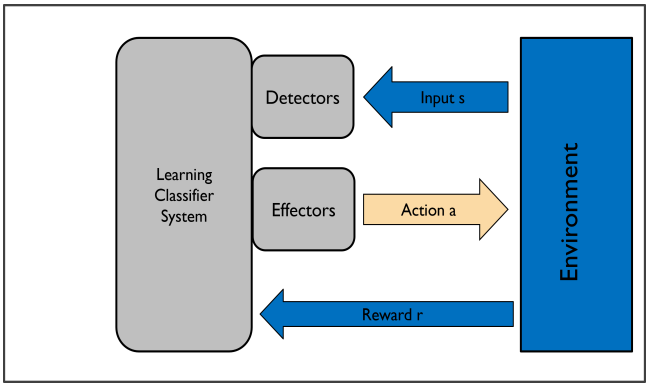


Fig. 1: Die Interaktion eines LCS mit dessen Umgebung

Arbeit verwendet. Die Bedingung eines Classifier gibt an, in welchen Zuständen die Regel des Classifier gilt. Es gibt verschiedene Arten die Bedingung eines Classifier zu kodieren. Die Population bildet eine der Hauptkomponenten des LCS. Insgesamt gibt es vier Hauptkomponenten, die sich in allen LCS Modellen wiederfinden [8] (vgl. Abb. 2). Die Interaktion mit der Umgebung und der Population wird von der Performance Component übernommen. Die Performance Component führt führt eine Iteration des Algorithmus aus und wählt eine auszuführende Aktion aus. Die Aktion führt zu einem numerischen Feedback, welches von der Credit Assignment Component genutzt wird, um die Wissensbasis zu aktualisieren und gute Classifier zu identifizieren und zu bestärken. Neue Classifier werden von der Discovery Component erzeugt und der Population hinzugefügt. Standardmäßig wird dies durch einen Genetischen Algorithmus umgesetzt. Die populärste Variante des LCS ist das XCS, welches 1995 von Wilson vorgestellt wurde [16]. Das XCS basiert auf Hollands standardisiertem Framework, zeichnet sich aber durch Änderungen an diesem aus. Ein wichtige Änderung ist, dass die Fitness eines Classifier nun nicht mehr von der erwarteten Belohnung, sondern auf der Genauigkeit der Vorhersage basiert.

III. Verwandte Arbeiten

Als Grundlage für die Implementierung diente die algorithmische Beschreibung des XCS von Butz & Wilson [2]. Es existieren mehrere Implementierungen eines XCS in unterschiedlichen Programmiersprachen. In Python gibt es das Paket *XCS* [9] von Entwickler hosford42, welches ebenfalls versucht ein XCS Framework aufzubauen. Dem XCS Paket von hosford42 mangelt es an Modularität, um XCS Komponenten auszutauschen und eigenes Verhalten zu

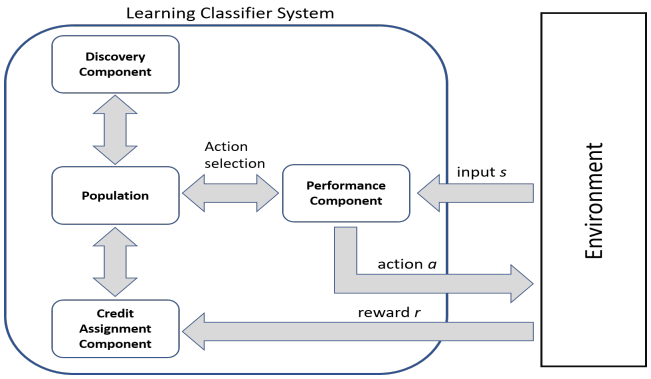


Fig. 2: Die Komponenten eines LCS

definieren. So sind zum Beispiel Operationen wie *mutate* des Genetischen Algorithmus teil der XCS Entität. Ein weiterer Unterschied besteht in der Einbindung der Hyperparameter eines XCS. In der XCS Implementierung von hosford42 werden die Hyperparameter durch Attribute der XCS Entität abgebildet. Eine weitere Python Implementierung liefert Nugroho Fredivianus auf GitHub [10]. Daneben existieren noch weitere Implementierungen in C++ und Java. Es handelt sich hierbei nicht um ein gewöhnliches XCS, sondern ein XCS mit einem besonderen Prozess für das Kombinieren von Regeln [3]. Der Fokus dieser Implementierung liegt deshalb nicht auf der Entwicklung eines Frameworks. Im Rahmen dieser Arbeit waren jedoch die dort verfügbaren Beispiele hilfreich.

IV. Aufbau Framework

Grundsätzlich wurde das Framework objektorientiert entwickelt und mit Unit Tests getestet. Es wurde eine auf Komponenten basierende Architektur gewählt. Dadurch konnten die einzelnen Komponenten des XCS als austauschbare Softwarekomponenten durch Komposition umgesetzt werden. Insbesondere können damit eigene Komponenten verwendet werden. In Abbildung 3 ist der Aufbau des Frameworks vereinfacht als UML Klassendiagramm dargestellt. Der Aufbau wurde vorab geplant und wurde nur geringfügig während der Implementierung geändert. Top-Down lässt sich der Aufbau wie folgt zusammenfassen. Die Entität XCS besitzt vier Komponenten, welche durch Interfaces definiert sind. Die Komponenten werden bei der Durchführung einer XCS Iteration verwendet. Diese Softwarekomponenten entsprechen dabei den Komponenten eines LCS. Eine weitere Komponente stellt die *ICoveringComponent* dar, welche für die Covering Operation zuständig ist. Die ICoveringComponent wird von der PerformanceComponent genutzt, falls es für einen gegebenen Input keinen Classifier gibt, dessen Bedingung mit dem Input erfüllt wird. Als standardmäßige Implementierung der IDiscoveryComponent wird ein Genetischer Algorithmus verwendet. Die Population eines XCS setzt sich aus einer Menge von Classifier zusammen. Ein Classifier besitzt durch Komposition eine Condition und eine Action. Eine Action ist generisch und kann daher jeden Datentyp annehmen. Eine Condition besteht aus einer Menge von *ISymbols*. Ein ISymbol ist eine Abstraktion eines beliebigen Symbols und definiert eine Methode, die angibt, ob das ISymbol zu einem gegebenem Wert passt. Eine besondere Rolle nimmt das WildCardSymbol ein, welches zu allen Werten passt. Es wird durch das Hashtag # repräsentiert. Ein einfaches Symbol kapselt einen einzelnen Wert. Dies kann eine Zahl oder ein Buchstabe sein. Eine Spezialisierung des ISymbol stellt das IBoundSymbol dar, welches Symbole repräsentiert, die einen beschränkten Bereich abdecken. Ein Beispiel für einen beschränkten Bereich wäre das Intervall $0.0 - 1.0$ oder die Buchstaben $a - d$. Diese Formulierung der Symbole erlaubt es beliebig komplexe Conditions zu formulieren. Die Implementierung von reellwertigen Symbolen basiert auf den Publikationen von Wilson [17], sowie Stone & Bull [15]. Das Framework bietet zwei Möglichkeiten für die Repräsentation von reellwertigen Symbolen. Bei der Center-Spread Variante wird ein Mittelpunkt und ein maximaler Spread um diesen definiert. Bei der Ordered-Bound Variante wird explizit eine untere und obere Grenze definiert. Die Wahl der Variante hat Auswirkungen auf den Genetischen Algorithmus und die Covering Operation.

Ein XCS besitzt eine Vielzahl an Hyperparametern. Die Unterbringung dieser stellte eine besondere Herausforderung dar. Die Wahl fiel auf eine objektorientierte Implementierung der Parameter. Klassen gruppieren eine Menge von Hyperparametern, die zusammengehörig sind. Zum Beispiel gibt es die Klasse *PopulationConstants* für Parameter bezüglich Populationen. In dieser Klasse sind die für eine Population relevanten Parameter, wie zum Beispiel maximale Größe, gruppiert und werden den jeweils benötigten Objekten übergeben (Dependency Inversion). Dies bedeutet, dass eine Population ein Objekt vom Typ *PopulationConstants* benötigt. Damit werden die Vorteile der OOP gewahrt.

V. Beispiele

Die Beispiele sind teil des Frameworks und können nach vorheriger Installation des Frameworks ausprobiert werden.

A. Multiplexer

Ein n-bit Multiplexer ist ein Klassifizierungsproblem und damit Single-Step. In diesem Beispiel wurde der 6-bit Multiplexer mit ganzen und reellwertigen Zahlen verwendet. Bei dem 6-bit Multiplexer mit ganzen Zahlen erhält das XCS einen Input bestehend aus sechs Einträgen $[n_0, ...n_5]$ mit $n \in 0, 1$. Die korrekte Antwort $a \in 0, 1$ wird wie folgt bestimmt. Die ersten zwei Bits des Inputs werden als Adresse verwendet, um auf eine der vier verbleibenden Bits zu zeigen. Sei als Input das Array $X = [0, 1, 1, 0, 1, 0]$ gegeben, dann formen die ersten beiden Einträge die Binärzahl 01. Damit lautet die korrekte Antwort $a = X[2 + 1] = X[3] = 0$. Das XCS erreicht nach etwa 2500 Beispielen fast durchgehend eine Genauigkeit von 100% mit einer maximalen Größe der Population von 300, die leer initialisiert wird. Die Werte der meisten Hyperparameter entsprachen den vorgeschlagenen Werten aus [2].

Verwendet man reellwertige Zahlen, dann wird zusätzlich ein Parameter θ definiert, der als Schwellwert zum Runden

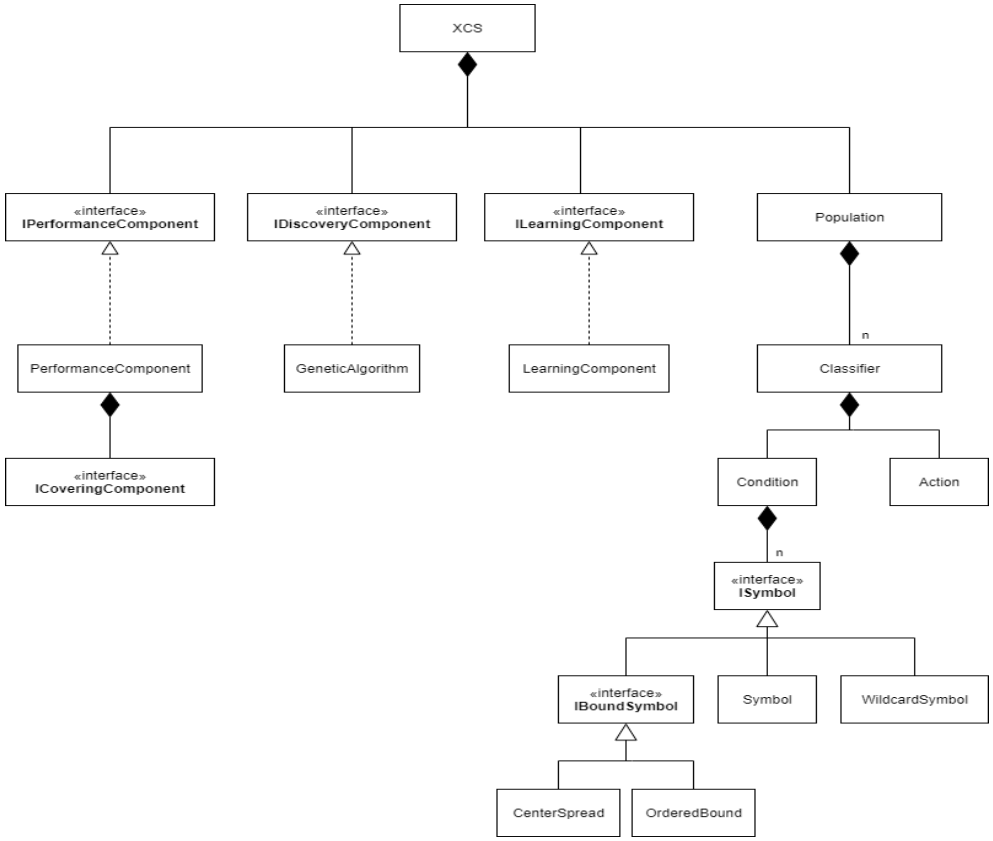
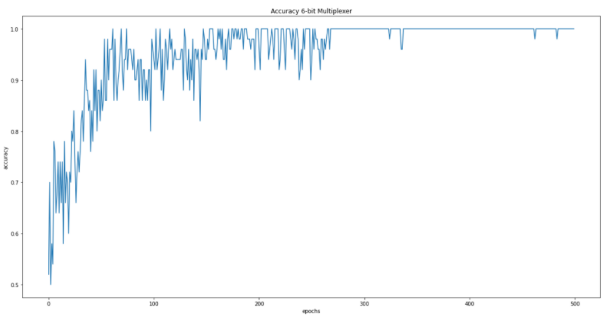


Fig. 3: Vereinfachte UML Darstellung des Frameworks

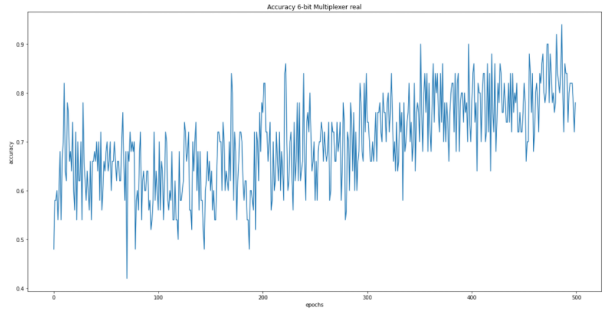
verwendet wird. Die korrekte Antwort für einen reellwertigen Eintrag x ist 0, falls $x < \theta$, sonst 1. In diesem Beispiel war $\theta = 0.5$ und der Wertebereich wurde auf das Intervall $[0.0, 1.0]$ beschränkt. Die maximale Größe der Population betrug 700. Für die Repräsentation der Symbole wurde CenterSpread verwendet. Gegenüber der binären Variante, erreicht das XCS hier keine 100% Genauigkeit. Eine Optimierung der Hyperparameter hat nicht stattgefunden, sodass Potential besteht das Ergebnis zu verbessern.

B. CartPole

Bei CartPole handelt es sich um ein Multi-Step Problem, bei dem das XCS lernen muss einen Klotz, an dem ein Stock befestigt ist, zu balancieren. CartPole ist teil von OpenAI Gym [11] und kann in Python als Modul installiert werden. Der Input setzt sich zusammen aus vier reellwertige Zahlen. Diese beschreiben die Position des Klotzes, dessen Geschwindigkeit, den Winkel des Stockes und dessen Geschwindigkeit. Die Geschwindigkeiten sind nicht beschränkt. Für das XCS mussten diese jedoch beschränkt werden, da das Framework nicht mit unbeschränkten Werten umgehen kann. Ziel ist es, den Stock so lange wie möglich zu balancieren. Hierfür gibt es zwei Aktionen, um den Klotz jeweils nach links oder rechts zu bewegen. Eine Episode endet, falls der Stock einen bestimmten Winkel überschreitet oder die maximale Zeit erreicht wurde. Die Bewertungsfunktion vergibt eine Belohnung, falls die gewählte Aktion den Stock näher an die neutrale Lage bringt oder der Stock sich bereits nahe



(a) Binärer 6-bit Multiplexer



(b) Reellwertiger 6-bit Multiplexer

Fig. 4: Verlauf der Genauigkeit beim 6-bit Multiplexer. Eine Epoche besteht aus zehn Beispielen.

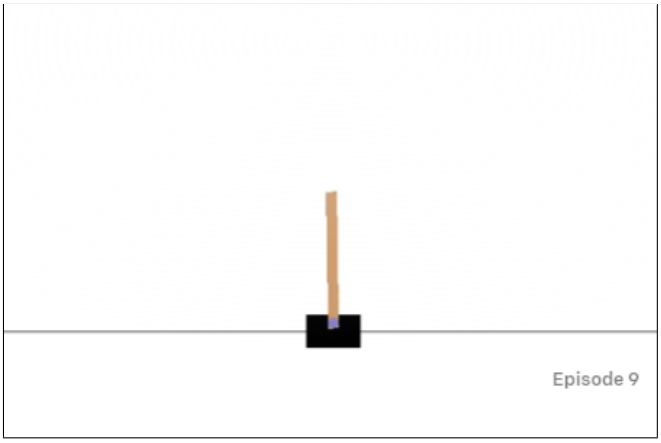


Fig. 5: Die CartPole Umgebung von OpenAI Gym

der neutralen Lage befindet. Die maximale Belohnung pro Frame beträgt 1.0. Diese Art der Bewertung führte zu einem deutlich besseren Ergebnis, als nur eine feste Belohnung pro Frame. Die maximale Dauer einer Epoche beträgt 195 Frames, sodass ein ideales Verhalten zu einer Summe von 195 pro Epoche führen würde. Das XCS trainierte für 200 Epochen mit jeweils 195 Iterationen. Nach jeder Epoche wurde die Güte des XCS ermittelt, indem das XCS für eine Epoche nur Greedy Selection für die Wahl der Aktion nutzte. Das Ergebnis ist in Abbildung 6 festgehalten. Das XCS schafft es nicht die maximale Belohnung von 195 zu erreichen. Stattdessen liegt der Höchstwert bei etwa 140. Dieser Wert wird jedoch nicht über die Dauer beibehalten. Vielmehr kommt es zu starken Schwankungen. Mögliche Gründe hierfür sind die Wahl der Hyperparameter, sowie der einfachen Bewertungsfunktion. Die Population mit dem besten Ergebnis wird als finale Population verwendet.

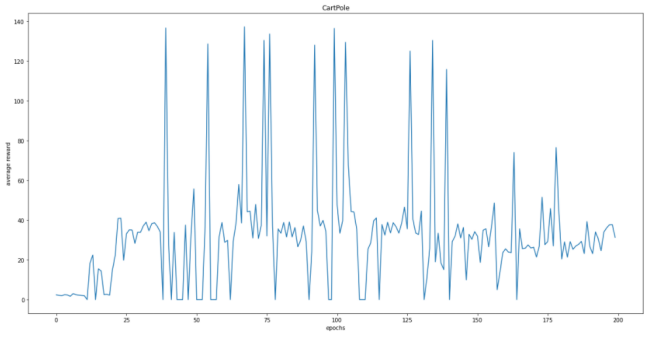


Fig. 6: Durchschnittliche Belohnung pro Epoche bei der CartPole Umgebung

VI. Zukünftige Arbeit

Anhand der Beispiele konnte gezeigt werden, dass mit dem entwickelten XCS Framework unterschiedliche Szenarien gelöst werden können. Jedoch sind die hier dargestellten Lösungen nicht optimal und bieten Raum für Verbesserungen. Eine mögliche Verbesserung wäre die Optimierung der verwendeten Hyperparameter in den Beispielen.

Das Framework implementiert das klassische XCS wie es in [2] beschrieben ist. Es wäre denkbar das Framework mit weiteren Implementierungen der Komponenten zu erweitern. Zum Beispiel könnte man den Rule Combining Prozess [3] in einer neuen Discovery Komponente realisieren oder eine neue Form des Credit Assignments in der Learning Komponente implementieren.

Eine weitere Erweiterung wäre das Hinzufügen einer neuen Art von Symbolen, deren Wertebereich nicht beschränkt ist.

Das Framework wurde für Michigan-Style LCS entwickelt. Denkbare wäre, dass man das Framework derart erweitert, dass auch Pittsburgh-Style LCS verwendet werden können.

Ein Vergleich mit anderen Frameworks anhand ausgewählter Beispiele wäre ein weiterer Ansatzpunkt für zukünftige Arbeiten.

References

[1] L. Bull, J. Sha'Aban, A. Tomlinson, J. D. Addison, and B. G. Heydecker. Towards Distributed Adaptive Control for Road Traffic Junction Signals using Learning Classifier Systems. In Larry Bull, editor, *Applications of Learning Classifier Systems*, pages 276–299. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[2] M. V. Butz and S. W. Wilson. An algorithmic description of XCS. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 6(3-4):144–153, June 2002.

[3] Nugroho Fredivianus. *Heuristic-based Genetic Operation in Classifier Systems*. PhD Thesis, Karlsruher Institut für Technologie (KIT), 2015.

[4] John Holland. *Adaptation, w: Progress in theoretical biology, red. R. Rosen, FM Snell*. New York, Plenum, 1976.

[5] John H Holland. The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. *Machine Learning, An Artificial Intelligence Approach*, 2:593–623, 1986.

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

[7] H Holland John. Adaptation in natural and artificial systems. *Ann Arbor: University of Michigan Press*, 1975.

[8] John H. Holmes, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. Learning classifier systems: New models, successful applications. *Information Processing Letters*, 82(1):23–30, April 2002.

[9] Aaron Hosford. xcs: XCS (Accuracy-based Classifier System). `http://hosford42.github.io/xcs`.

[10] Fredivianus Nugroho. nuggfr - Repositories. `https://github.com/nuggfr`.

[11] OpenAI. Gym: A toolkit for developing and comparing reinforcement learning algorithms. `https://gym.openai.com`.

[12] Viet Pham, Lam Bui, Sameer Alam, Chris Lokan, and Hussein Abbass. A Pittsburgh Multi-Objective Classifier for user preferred trajectories and flight navigation. pages 1–8, 2010.

[13] Arman Rezaee. Control of Road Traffic Using Learning Classifier System. *Journal of Applied and Computational Mathematics*, 2016:1–4, 2016.

[14] K. Shafi and H. A. Abbass. A Survey of Learning Classifier Systems in Games [Review Article]. *IEEE Computational Intelligence Magazine*, 12(1):42–55, February 2017.

[15] Christopher Stone and Larry Bull. For Real! XCS with Continuous-Valued Inputs. *Evolutionary Computation*, 11(3):299–336, September 2003.

[16] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3:149–175, 1995.

[17] 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.