The XCSF Classifier System in Java

Martin V. Butz, University of Würzburg, Germany, butz@psychologie.uni-wuerzburg.de

After many inquiries by various researchers in the field, I finally got around to completely restructure my XCS Java code released in 2000 (!) and was able to make a new version available. Due the apparent lack of publicly available real-valued XCS implementations and the struggle of many to implement it from scratch, I invested the time and put out a real-valued XCS implementation for function approximation. The code is available from Martin Pelikan's MEDAL lab webpage (http://medal.cs.umsl.edu/files/XCSFJava1.1.zip). It includes the Java API, which should give a general idea of how the code works and how classes communicate with each other. Moreover, there is a short MEDAL report available that gives further code details and also points out what needs to be done to adjust or enhance the code for your individual research needs (http://medal.cs.umsl.edu/files/2007008.pdf). This note is meant to spread the good news and to give a short idea of what the code is capable of.

Code Features

The XCSFJava1.1 code includes most XCSF features published so far. Classifiers consist only of conditions and predictions. Thus, the implementation does not support problems in which more than two classes need to be distinguished — although the necessary enhancements are rather easy to accomplish. Nonetheless, the code can be applied to approximate Boolean functions with two problem classes. That is, it can be tested on the multiplexer problem as well as on any other imaginable binary classification problem — instead of determining an action for classification, though, the XCSF implementation determines only if a problem instance yields high payoff (belonging to class one) or low payoff (class zero). Certainly, though, XCSFJava1.1 is best suited for real-valued function approximation problems. In the real-valued domain, predictions can be optionally generated as constant predictions or linear predictions (linearly dependent on the input), which in turn can be either updated by the (simpler) delta rule or by recursive least squares techniques. XCSFJava1.1 supports hyper-rectangular conditions as well as hyper-spheroid and hyper-ellipsoidal conditions. Besides the different options for the classifier structure, the code also supports classifier population compaction mechanisms [1]. The compaction mechanisms include condensation, in which the GA is applied without mutating or recombining the offspring, closest classifier matching, in which a certain fixed number of classifiers closest to in the problem input match, and an optional greedy compaction algorithm, which does a prioritized sweep over the population typically eliminating more than 90% of the classifiers in the population while only marginally decreasing approximation accuracy. The above options can be conveniently specified in a parameter file — of which a basic version is provided with the code package. The parameters also allow the specification of other setup parameters, including other XCS learning parameters, performance monitoring, and optional visualisations of the learning process. The last feature is the mentioned performance monitoring using visualizations. 2D and 3D visualization of respective problem input space dimensions are supported for hyper-rectangular conditions and most hyper-ellipsoidal condition types. The code shows the classifiers evolving in the problem input space, visualizing the (scaled) location, size, orientation, and fitness of the classifier conditions. Java 3D is used for this feature in 3D input spaces.

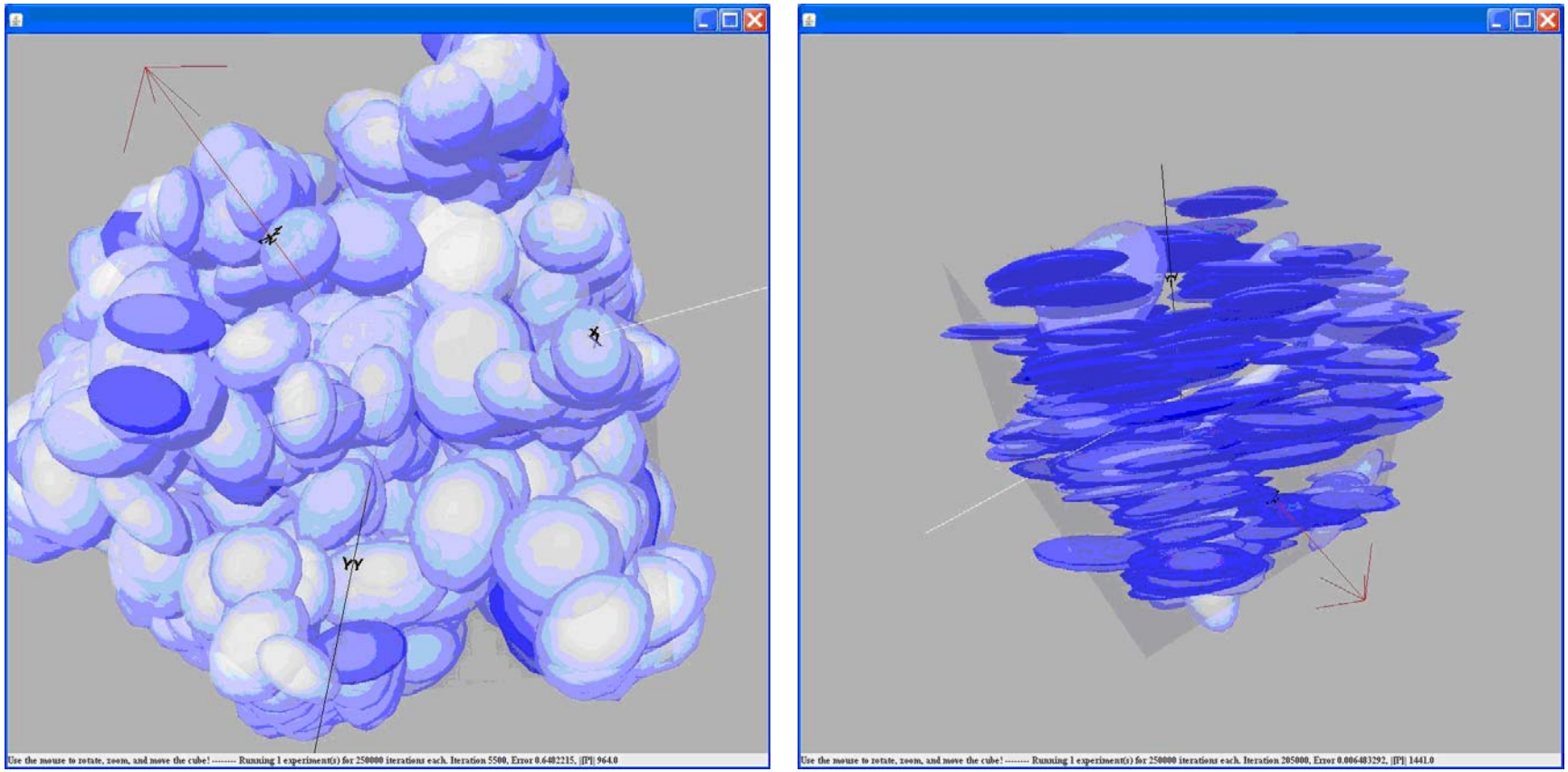

Fig. 1: A population of condition structures early and late during learning. Darker shapes indicate higher fitness. The conditions visualized are 20% of the actual size.

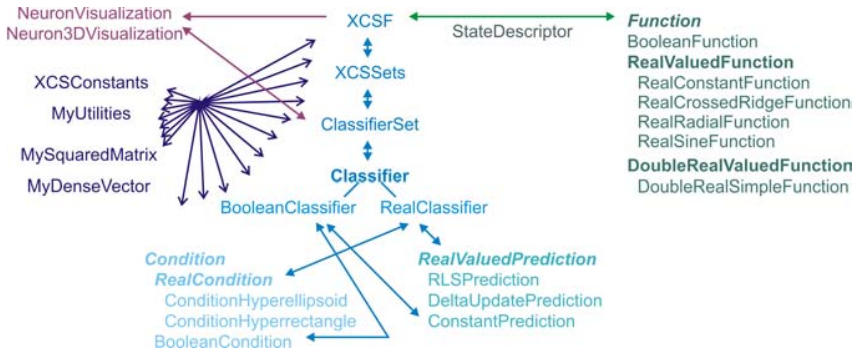

Fig. 2: Overall code structure showing the basic interactions between the classes.

It is possible to either monitor the evolutionary process step-by-step, in which case also the matching and offspring classifiers are highlighted, or to view the evolving population periodically after a specified number of learning iterations. Figure 1 shows two screenshots of the 3D visualization early and late during learning an oblique sine wave function with hyper-ellipsoidal classifier conditions and linear approximations.

Code Structure

Figure 2 gives a general idea of the overall functioning of the code. The `XCSF` class serves as the main executive — generating experimental runs, maintaining performance statistics, and iteratively executing the main learning iterations. `XCSSets` maintains population and current match sets and delegates learning and classification steps to the appropriate classifier sets. `ClassifierSet` implements a classifier set and all XCS-relevant operations on sets including updating, evolutionary component, and classifier additions and deletions. The interface `Classifier` maintains one classifier structure — dependent on the chosen problem a `BooleanClassifier` or a `RealClassifier`, where a Boolean classifier consists of a `BooleanCondition` class object and a `ConstantPrediction` class object whereas a real-valued classifier consists of a `RealCondition` interface object, which can be either of class `CondtionHyperrectangle` or `ConditionHyperellipsoid`, and a `RealValuedPrediction` interface object, which can be of class `ConstantPrediction`, `DeltaUpdatePrediction`, or `RLSPrediction`. Additionally, the `Function` interface realizes the communication with the targeted function approximation problem where problem instances are coded as `StateDescriptor` objects. As current test functions, the `BooleanFunction` class implements a constant Boolean function and the multiplexer function whereas the `RealValuedFunction` implementations can generate a `RealConstantFunction`, a `RealCrossedRidgeFunction`, a `ReadRadialFunction`, or a `RealSineFunction` class object. Multiple parameters allow the further modification of the selected test function's size and complexity. Class `XCSConstants` provides static access to all XCS relevant constants and also supports reading and writing the constants. `MyUtilities` encodes some utilities such as sorting and the determination of a fixed number of closest classifiers. Also the utilized random number generator is located here. Finally, `MySquaredMatrix` and `MyDenseVector` are used as fundamental data structures within various parts of the code.

Final Remarks

The code is publicly available and meant for academic use. While there is no warranty for the code's correctness, it has been intensively tested and evaluated in the provided problem domains confirming its validity. I strongly hope that it is useful for students and teachers alike to get a kick-start in the successful application of XCS to their problem at hand as well as for successful future research studies. Especially the provided visualization options should help to quickly gain a deeper understanding of the general functioning of XCS. Thus, I can only wish everybody good luck with using the code, have fun, and if there are any questions or if bugs are spotted or if you simply like or hate the code for whatever reason, then let me know!

Bibliography

[1] M. V. Butz, P. L. Lanzi, S. W. Wilson. *Function Approximation with XCS: Hyperellipsoidal Conditions, Recursive Least Squares, and Compaction* IEEE Transactions on Evolutionary Computation. (in press).

About the author

Martin V. Butz received his PhD in computer science at the University of Illinois at Urbana-Champaign in October 2004 under the supervision of David E. Goldberg. His thesis "Rule-based evolutionary online learning systems: Learning Bounds, Classification, and Prediction" puts forward a modular, facet-wise system analysis for Learning Classifier Systems (LCSs) and analyzes and enhances the XCS classifier system. Until September 2007, Butz was working at the University of Würzburg at the Department of Cognitive Psychology III on the interdisciplinary cognitive systems project "MindRACES: From reactive to anticipatory cognitive embodied systems". In October 2007 he founded his own cognitive systems laboratory: "Cognitive Bodyspaces: Learning and Behavior" (COBOSLAB), funded by the German research foundation under the Emmy Noether framework. Butz is the co-founder of the "Anticipatory Behavior in Adaptive Learning Systems (ABiALS)" workshop series and is currently also co-organizing the "International Workshop on Learning Classifier Systems" (IWLCS) series. Butz has published and co-edited four books on learning classifier systems and anticipatory systems. Currently, he is focusing on the design of highly flexible and adaptive cognitive system architectures, based on recent research insights gained in cognitive psychology, behavioral neuroscience, evolutionary biology, and adaptive behavior.

Homepage: http://www.psychologie.uni-wuerzburg.de/i3pages/butz
COBOSLAB: http://www.coboslab.psychologie.uni-wuerzburg.de
Email: mbutz@psychologie.uni-wuerzburg.de