Missouri Estimation of Distribution Algorithms Laboratory

Documentation of XCSFJava 1.1 plus Visualization

Martin V. Butz

MEDAL Report No. 2007008

July 2007

Abstract

This report gives an overview of the XCSFJava 1.1 code, available from the web. The document specifies where to get the code and how to compile and run the code. Moreover, the document specifies the features of the code. In short, XCSFJava 1.1 is an XCSF classifier system implementation that can be used at least for the following purposes: (1) testing and evaluating XCSF on various function approximation problems, including binary and real-valued problems, (2) visualizing the evolutionary process in XCSF in 2D or 3D, (3) enhancing XCSF's capabilities, such as adding new types of classifier conditions or predictions.

Keywords

Learning classifier systems, LCS, XCS, XCSF, Java, implementation, function approximation, evolutionary computation, visualization.

Missouri Estimation of Distribution Algorithms Laboratory (MEDAL)
Department of Mathematics and Computer Science
University of Missouri–St. Louis
One University Blvd., St. Louis, MO 63121
E-mail: medal@cs.umsl.edu
WWW: http://medal.cs.umsl.edu/

Documentation of XCSFJava 1.1 plus Visualization

Martin V. Butz
Dept. of Psychology (Cognitive Psychology III)
University of Würzburg
Röntgenring 11, Würzburg, Germany 97070
butz@psychologie.uni-wuerzburg.de

Abstract

This report gives an overview of the XCSFJava 1.1 code, available from the web. The document specifies where to get the code and how to compile and run the code. Moreover, the document specifies the features of the code. In short, XCSFJava 1.1 is an XCSF classifier system implementation that can be used at least for the following purposes: (1) testing and evaluating XCSF on various function approximation problems, including binary and real-valued problems, (2) visualizing the evolutionary process in XCSF in 2D or 3D, (3) enhancing XCSF's capabilities, such as adding new types of classifier conditions or predictions.

Keywords: Learning classifier systems, LCS, XCS, XCSF, Java, implementation, function approximation, evolutionary computation, visualization.

1 Introduction

The XCS classifier system (Wilson, 1995, 1998) is a learning classifier system (Booker, 1988; Holland & Reitman, 1978; Wilson & Goldberg, 1989) that evolves its rules, the so-called classifiers, by an accuracy-based fitness approach. It has been successfully applied to boolean classification problems (Butz, Kovacs, Lanzi, & Wilson, 2004; Butz, Goldberg, & Tharakunnel, 2003; Butz, 2006; Wilson, 1995, 1998), datamining problems (Bernadó, Llorà, & Garrell, 2002; Bernadó-Mansilla & Garrell-Guiu, 2003; Butz, 2006), multistep (Markov decision) problems (Butz, 2006; Butz, Goldberg, & Lanzi, 2005; Lanzi, 1999; Lanzi, Loiacono, Wilson, & Goldberg, 2006a; Wilson, 1995) as well as real-valued function approximation problems (Butz, 2005; Butz, Lanzi, & Wilson, in press; Lanzi, Loiacono, Wilson, & Goldberg, 2006b, 2007; Wilson, 2002).

This report provides a description of the XCSFJava 1.1 implementation, which encodes XCS for function approximation (XCSF). XCSF's functionality is similar to that of XCS, except that there are no actions (or one "dummy" action) in each classifier and the predictions are often computed out of problem input values (Wilson, 2001, 2002). The implementation realizes many known XCSF features and particularly replicates the results published elsewhere (Butz et al., in press). Moreover, the implementation also supports binary classification problems, such as the multiplexer problem (cf. Butz et al., 2003, 2004; Wilson, 1998). In binary classification problems, XCSFJava 1.1 yields faster performance (number of iterations), which is because the original XCS with actions learns the solution twice: for each solution XCS evolves two classifiers, one with action "0" and another one with action "1" that specify that one of the actions is correct and the other is not. XCSF only learns if the current problem instance will yield high payoff (that is, belongs to class "1") or low payoff (that is, it belongs to class "0"). Also the runtime performance does not appear to yield significant

differences to a known C implementation (cf. XCS (+ tournament selection) classifier system implementation in C, version 1.2, `http://www.illigal.uiuc.edu/pub/src/XCS/XCS1.2.tar.Z`, Butz, 2003). The XCSFJava 1.1 code also includes the recently introduced greedy compaction algorithm as well as closest classifier matching (Butz et al., in press).

Finally, the code also includes visualization methods in 2D and 3D. The visualizations show the evolutionary process. Particularly, it shows the evolving XCS classifier conditions colored fitness-dependently. The visualization makes it possible to actually see XCSF at work. Visualization is supported at several time scales including single evolutionary iterations, also showing matching classifiers and novel offspring classifiers, as well as longer evolutionary intervals.

This report now first specifies the code location of XCSF, the features included in the code archive, and the requirements to run the code. Next, we specify the provided parameters that allow various code modifications. Finally, we give some more details on how to implement other test functions and how to enhance and further evaluate XCSF by means of the provided XCSF code.

2 Code Introduction

The XCSFJava 1.1 code is a fundamentally re-written XCS implementation, which was derived from the XCSJava 1.0 code (Butz, 2000). While XCSJava 1.0 only worked for binary inputs, XCSFJava 1.1 handles binary inputs more effectively but can also handle real-valued problem inputs. However, the traditional action part of XCS (Wilson, 1995) is not included in XCSFJava 1.1 making it only suitable for function approximation. Nonetheless, certainly an enhanced version of the code may also include classifier actions and consequently may also be applied to multistep (Markov decision) problems.

XCSFJava 1.1 uses the matrix toolkits for Java (MTJ)[1] to handle vector and matrix multiplications. Additionally, Java3D 1.5.x[2] is used for 3D classifier visualization.

This section gives an introduction to the code including where to get the code, the basic requirements to run the code in Java, and the basic structure of the code. Note that the code includes also its own API for reference purposes as well as an executable .jar file for your convenience.

2.1 Downloading and Running the Code

The XCSFJava 1.1 code is available from the following web address:

`http://medal.cs.umsl.edu/files/XCSFJava1.1.zip`

You can unzip the code in a folder of your convenience. The code includes an XCSFJava1.1.jar file that enables you to run the code directly.

To run the code successfully, it is important that the newest Java version and Java 3D 1.5.x are installed on your system. This should enable you to use the full functionality of the XCSFJava 1.1 implementation. Moreover, a `test.xcs` file is provided that specifies XCSF and other setup parameters.

[1] Available from `http://rs.cipr.uib.no/mtj/`

[2] Java 3D is available from `https://java3d.dev.java.net/` .

2.2 Basic Code Structure

The XCSFJava 1.1 API, located in sub-folder `API`, is a convenient way to understand the code, its general structure, and the detailed purposes for each implemented class and method. Thus, this document only provides a general overview over the code structure and the parameters, classes, and interfaces involved.

The `XCSF` class specifies the main method and also implements all functions necessary to control the execution of one experiment and to monitor performance. The `XCSConstants` class contains all parameters relevant to set up a successful XCSF experiment. It also provides convenient methods to read and write the parameters from and to a file and to maintain all the parameters in a *Properties* object.

There are three major code clusters below the `XCSF` class, which are invoked directly or indirectly from the `XCSF` class and which may access parameters in the `XCSConstants` class: (1) the actual XCSF implementation; (2) the function interface; and (3) the visualization package. Additionally, some utilities are implemented.

2.2.1 XCSF Structure

The `XCSSets` class maintains classifier population and match set, delegating matching, prediction generation, classifier updating, and evolutionary procedures to the appropriate classifier set. The sets are maintained and handled in the `ClassifierSet` class. This class provides the functionality for generating match sets, potentially triggering covering, updating the match set based on the function value feedback, and executing the genetic algorithm in a classifier set.

Classifiers are represented as objects of the abstract `Classifier` class, which provides the base functionality of any XCSF classifier. Actual complete implementations of a classifier are realized in the `RealClassifier` and `BooleanClassifier` classes. These two classes essentially implement all functionalities that include a particular condition structure and prediction structure. Abstract methods that need to be implemented to realize an XCSF classifier are:

```
public abstract void updatePrediction(double[] actualValue);
public abstract boolean doesMatch(StateDescriptor state);
public abstract double getMatchVote(StateDescriptor state);
public abstract void getReference(StateDescriptor state);
public abstract double[] setCurrentPrediction(StateDescriptor state);
public abstract void updatePredictionError(double[] actualValue);
public abstract boolean isMoreGeneral(Classifier cl);
public abstract void uniformCrossover(Classifier cl);
public abstract boolean applyMutation(StateDescriptor state);
public abstract boolean isIdentical(Classifier cl);
public abstract Condition getCondition();
public abstract double getGenerality();
public abstract void printClassifier();
public abstract void printClassifier(PrintStream pS);
```

The intended functionality should be either self-explanatory or you can refer to additional information given in the XCSFJava 1.1 API.

Each classifier contains a condition object, where a `RealClassifier` contains a `RealCondition` object and a `BooleanClassifier` a `BooleanCondition` object. The `RealCondition` ob-

ject is actually an interface that is implemented by the `ConditionHyperellipsoid` and `ConditionHyperrectangle` classes.

Each classifier also contains a `RealValuedPrediction` object, which is again an interface. There are currently three forms of predictions implemented: `ConstantPrediction` realizes the constant prediction representation and update, used in the XCS classifier system mainly for binary problems (Wilson, 1995); `DeltaUpdatePrediction` realizes a linear computed prediction that is updated by the Widrow-Hoff delta rule (Widrow & Hoff, 1960; Wilson, 2002); finally, `RLSPrediction` implements recursive least squares prediction updates (Haykin, 2002; Lanzi, Loiacono, Wilson, & Goldberg, 2005).

2.2.2 Test Functions

The XCSF implementation contains one `Function` object, which is an interface that requires the following methods to be implemented by an actual implementation of the interface:

```
public abstract StateDescriptor getCurrentProblemInput();
public abstract double[] getFunctValue();
public abstract int getInputLength();
public abstract int getOutputLength();
public abstract int getPredictionInputLength();
```

The first method generates a (random) problem instance using the `StateDescriptor` class that provides a convenient way to include various types of input. Currently, boolean and real-valued problem inputs are supported.

The method *getFunctValue()* returns a double array that contains all the real values associated with the last generated problem input. Note that the XCSFJava 1.1 implementation supports the prediction of multiple values. If multiple function values need to be predicted, XCSF treats those values independently, that is, each classifier generates prediction values for all function values and the classifier prediction error is determined out of the average of all single prediction errors.

Methods *getInputLength()* and *getOutputLength()* simply return input and output length. Finally, *getPredictionInputLength()* returns the input length that is used by the classifiers to generate their prediction (if applicable). This enables the system to develop classifiers whose conditions depend on one type of problem input but whose predictions depend on some other type of problem input. Class `DoubleRealSimpleFunction` provides an example of this problem type.

For the boolean case, `BooleanFunction` implements a constant boolean problem and the multiplexer problem. The following real-valued functions are implemented: `RealConstantFunction`, `RealSineFunction`, `RealCrossedRidgeFunction`, and `RealRadialFunction`. The choice of the function and optional additional function modifications are specified in the `XCSConstants` class (cf. the API).

2.2.3 Visualization

Two classes contain the current visualization package. Class `NeuronVisualization` visualizes the classifiers in 2D showing their distribution, spatial coverage, and current fitness. The class, when instantiated, simply takes the first two dimension of a `RealClassifier` to locate the classifier in 2D space. The space covered by a condition is visualized scaled by parameter *relativeVisualizedConditionSize* specified in `XCSConstants`. Stretch and orientation of an ellipsoid are not considered in the case of general hyperellipsoids without explicit rotation.

Class `Neuron3DVisualization` displays classifiers in 3D, given classifier conditions cover at least a 3D input space. As in the 2D case, potential additional dimensions are ignored by the class. Conditions are represented as spheres that cover a certain percentage of the space (specified in the *relativeVisualizedConditionSize* parameter in the `XCSConstants` class).

2.2.4 Utilities

A couple of additional utility classes were implemented. Class `MyUtilities` implements object sorting using quicksort and the determination of the k best objects, which is used for closest classifier matching. Moreover, `MyUtilities` supports the generation of uniform random numbers between zero and one, specified in *uniRand()*, as well as the generation of normally generated numbers, specified in *normRand()*. Additionally, the classes `MyDenseVector` and `MySquaredMatrix` are extensions of the classes `DenseVector` and `DenseMatrix`, respectively, providing convenient methods to instantiate and use vectors and matrices in the XCSF implementation.

2.3 Generated Output

By default, XCSF writes its learning progress to standard output as well as to a file, which is specified as the (sole) input parameter. If no input parameter is specified, output is written to the file `test.res`. The output contains the following performance information:

1. Learning Iteration Step
2. Average absolute error over the last *averageExploitTrials* iterations
3. Number of distinct classifiers (macro classifiers) in population (at the specified iteration step)
4. Population size (sum of numerosities)
5. Average number of distinct classifiers in the last *averageExploitTrials* match sets
6. Average match set size in the last *averageExploitTrials* match sets
7. Average prediction error in population
8. Average fitness in population
9. Average generality in population
10. Average experience in population
11. Average set size estimate in population
12. Average time stamp in population
13. Additionally, individual errors if multiple prediction values are generated.

At the end of each run, XCSFJava 1.1 still generates a `test.avd` file that contains parameter settings and all the above listed performance measures averaged over all executed experiments. Each average is additionally followed by its (unbiased) standard deviation estimate.

Visualization output is generated only if the *doVisualization* flag is set to *true* in the `XCSConstants` class and a real-valued problem is executed. If selected and real-valued classifiers are generated, a window opens in which the classifier population is visualized in 2D or 3D, dependent

on if the problem input contains only two or at least three dimensions, respectively. Moreover, the generated images can be saved by setting the *doOnlineScreenshots* flag to *true*. In this case, the screenshots of the window that contains the visualization are taken. The frequency of visualization updates and taken screenshots coincides and is specified in the *updateVisualizationSteps* parameter.

3 Parameter Modifications

All relevant XCSFJava 1.1 parameters are included in the `XCSConstants` class. The API gives details to each actual parameter. Thus, we will not specify all the parameters in this document but rather specify how to modify them.

Generally, there are two ways to modify the parameters: First, the XCSConstants file may be modified and recompiled. In this case the user should make sure that there is no `test.xcs` file contained in the folder in which XCSFJava 1.1 is started, since `test.xcs` may overwrite parameters set in `XCSConstants` (unless a startup parameter is given, which can be used to specify the name of a different parameter file). Second, and more conveniently, the `test.xcs` file allows the modification of parameter values without the need for recompilation. At the beginning of a run, XCSF checks for this parameter file in the folder the program is started in. If the file does not exist, the process throws an exception and continues with default parameters. If the file exists, XCSFJava 1.1 treats the file as a Java Properties file that contains tuples of parameter names and values on each line. If the name corresponds to a name of an actual XCSFJava parameter, the default value is replaced by the one specified in the file. The XCSFJava 1.1 package includes a `test.xcs` file that specifies default parameter values at your convenience.

4 Adding New Code

The object-oriented implementation of XCSFJava 1.1 should provide a convenient way to add your own pieces of code to test other functions or to enhance the current XCSF capabilities themselves. We now give a short overview of how this might be accomplished.

4.1 New Test Functions

There are several ways to add new test functions dependent on the type and complexity of the function. Real valued functions may be added by implementing the abstract `RealValuedFunction` class. Classes `RealConstantFunction`, `RealSineFunction`, `RealCrossedRidgeFunction`, and `RealRadialFunction` provide good examples for such implementations.

Double real valued functions are problems that provide two types of input: an input for matching and one for prediction. To test XCSF on such a problem, both input sizes need to be specified. The implementation `DoubleRealSimpleFunction` provides an example for this problem type.

Regardless if an additional real-valued function or a double real-valued function is implemented, it still needs to be invoked by the XCSF code. To do so, only the *main* method in the XCSF class needs to be slightly modified: Dependent on the *functionType* parameter, which is specified in the `XCSConstants` class, the chosen function is instantiated here. Thus, you need to add the construction of the novel function in the switch statement with a new *functionType* value of your choice.

If you want to add a novel Boolean function, it is best to add it directly in the `BooleanFunction` class. The Boolean function type is chosen based on the *functionType* value that needs to be set

to a value above 99 in this case. The actual type is then determined by the value minus 100. Additionally, the *getFunctValue()* method needs to be enhanced to generate the correct function value for the new Boolean function. Usually, zero or one is returned, but also other values may be used such as those used in the layered multiplexer problem (Butz et al., 2003; Wilson, 1995)

4.2 Adding XCSF Functionalities

Besides the option to add additional test problems and evaluate the XCSF implementation based on these problems, naturally, it is also possible to modify the code as you please to incorporate new features.

4.2.1 Other Condition Types

Conditions distinguish between real-valued conditions and boolean conditions. If you intend to implement another Boolean Condition, you may either want to modify the `BooleanConditon` class itself, extend the `BooleanConditon` class, or add a new class that is chosen alternatively to the currently implemented `BooleanConditon` class. In the latter cases, the `BooleanClassifier` class will need to be modified accordingly.

To add other types of real-valued conditions it is sufficient to implement another condition class that implements the `RealConditon` interface. The methods that need to be implemented should be self-explanatory. The API of the `Condition` and `RealCondition` interfaces as well as the current implementations `ConditionHyperrectangle` and `ConditionHyperellipsoid` provide further useful bits of information.

Additionally, other conditions, such as conditions that handle nominal inputs or a mixture of inputs may be implemented. Any condition should however implement the Condition interface to ensure general compatibility.

4.2.2 Other Prediction Types

Also other prediction types may be included. To generate other real-valued prediction types, the abstract class `RealValuedPrediction` should be implemented, as long as the prediction is actually real-valued. An actual prediction instantiation is then realized in the actual classifier implementation class. `RealClassifier` provides an example for different prediction types that are invoked dependent on the `XCSConstants` parameter *predictionType*.

5 Final Comments

The code is distributed for academic purposes with absolutely no warranty of any kind, either expressed or implied, to the extent permitted by applicable state law. We are not responsible for any damage from its proper or improper use.

Feel free to use, modify and distribute the code with an appropriate acknowledgment of the source, but in all resulting publications please include the following citation:
Butz (2007). *Documentation of XCSFJava 1.1 plus Visualization.* MEDAL Report No. 20070008, Missouri Estimation of Distribution Algorithms Laboratory, University of Missouri in St. Louis, MO.

If you find any bugs in the source code, please contact the author of this report.

Acknowledgments

The author would like to thank Marc-Oliver Ochlast for taking first steps in the realization of this XCSF implementation, Kevin Reif for providing the initial visualization routines, and Oliver Herbort for supporting the implementation progress in various ways. The author would also like to thank Martin Pelikan for the useful discussions and Marjorie Kinney for proofreading. This work was supported by the European commission contract no. FP6-511931, MindRACES: From Reactive to Anticipatory Cognitive Embodied Systems.

References

Bernadó, E., Llorà, X., & Garrell, J. M. (2002). XCS and GALE: A comparative study of two learning classifier systems and six other learning algorithms on classification tasks. In P. L. Lanzi, W. Stolzmann, & S. W. Wilson (Eds.), *Advances in learning classifier systems (lnai 2321)* (p. 115-132). Berlin Heidelberg: Springer-Verlag.

Bernadó-Mansilla, E., & Garrell-Guiu, J. M. (2003). Accuracy-based learning classifier systems: Models, analysis, and applications to classification tasks. *Evolutionary Computation*, *11*, 209-238.

Booker, L. B. (1988). Classifier systems that learn internal world models. *Machine Learning*, *3*, 161-192.

Butz, M. V. (2000). *XCSJava 1.0: An implementation of the XCS classifier system in Java* (IlliGAL report No. 2000027). University of Illinois at Urbana-Champaign: Illinois Genetic Algorithms Laboratory.

Butz, M. V. (2003). *Documentation of XCS+TS c-code 1.2* (IlliGAL report No. 2003023). University of Illinois at Urbana-Champaign: Illinois Genetic Algorithms Laboratory.

Butz, M. V. (2005). Kernel-based, ellipsoidal conditions in the real-valued XCS classifier system. *GECCO 2005: Genetic and Evolutionary Computation Conference*, 1835-1842.

Butz, M. V. (2006). *Rule-based evolutionary online learning systems: A principled approach to LCS analysis and design*. Berlin Heidelberg: Springer-Verlag.

Butz, M. V., Goldberg, D. E., & Lanzi, P. L. (2005). Gradient descent methods in learning classifier systems: Improving XCS performance in multistep problems. *IEEE Transactions on Evolutionary Computation*, *9*, 452- 473.

Butz, M. V., Goldberg, D. E., & Tharakunnel, K. (2003). Analysis and improvement of fitness exploitation in XCS: Bounding models, tournament selection, and bilateral accuracy. *Evolutionary Computation*, *11*, 239-277.

Butz, M. V., Kovacs, T., Lanzi, P. L., & Wilson, S. W. (2004). Toward a theory of generalization and learning in XCS. *IEEE Transactions on Evolutionary Computation*, *8*, 28-46.

Butz, M. V., Lanzi, P. L., & Wilson, S. W. (in press). Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction. *IEEE Transactions on Evolutionary Computation*.

Haykin, S. (2002). *Adaptive filter theory* (4th ed.). Upper Saddle River, NJ: Prentice Hall.

Holland, J. H., & Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In D. A. Waterman & F. Hayes-Roth (Eds.), *Pattern directed inference systems* (pp. 313–329). New York: Academic Press.

Lanzi, P. L. (1999). An analysis of generalization in the XCS classifier system. *Evolutionary Computation*, *7*(2), 125-149.

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2005). *Generalization in the XCSF*

classifier system: Analysis, improvement, and extensions (IlliGAL report No. 2005012). University of Illinois at Urbana-Champaign: Illinois Genetic Algorithms Laboratory.

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2006a). Classifier prediction based on tile coding. *GECCO 2006: Genetic and Evolutionary Computation Conference*, 1497-1504.

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2006b). Prediction update algorithms for XCSF: RLS, kalman filter and gain adaptation. *GECCO 2006: Genetic and Evolutionary Computation Conference*, 1505-1512.

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2007). Generalization in the XCSF classifier system: Analysis, improvement, and extension. *Evolutionary Computation*, *15*, 133-168.

Widrow, B., & Hoff, M. (1960). Adaptive switching circuits. *Western Electronic Show and Convention*, *4*, 96-104.

Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, *3*(2), 149-175.

Wilson, S. W. (1998). Generalization in the XCS classifier system. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 665–674.

Wilson, S. W. (2001). Function approximation with a classifier system. *Proceedings of the Third Genetic and Evolutionary Computation Conference (GECCO-2001)*, 974-981.

Wilson, S. W. (2002). Classifiers that approximate functions. *Natural Computing*, *1*, 211-234.

Wilson, S. W., & Goldberg, D. E. (1989). A critical review of classifier systems. *Proceedings of the Third International Conference on Genetic Algorithms*, 244–255.