LogFileAnalyzer for Learning Classifier Systems

– Version 1.1 –

Clemens Gersbacher Holger Prothmann*
holger.prothmann@kit.edu

August 29, 2008

The LogFileAnalyzer supports the analysis of Learning Classifier System (LCS) experiments. LCSs are rule-based evolutionary learning systems that code their knowledge in a population of rules (called *classifiers*). The classifier population evolves and changes constantly. New classifiers are created, bad classifiers are deleted, and existing classifiers are modified and updated. The LogFileAnalyzer can help to better understand this dynamic process by visualizing the classifier sets in tables and histograms.

Contents

*Karlsruhe Institute of Technology (KIT), Univ. Karlsruhe (TH), Institute AIFB, 76128 Karlsruhe, Germany

1 Introduction

The LogFileAnalyzer is a program that supports the analysis of Learning Classifier System (LCS) experiments. LCSs are rule-based evolutionary learning systems [1] that code their knowledge in a population of rules (called *classifiers*). A classifier cl consists of a condition $cl.c$, an action $cl.a$, and an evaluation that contains (usually among other values) a reward prediction $cl.r$. Given that condition $cl.c$ is satisfied and action $cl.a$ is executed, the classifier cl predicts a reward $cl.r$. In an LCS experiment, the classifier population evolves and changes constantly. New classifiers are created, bad classifiers are deleted, and existing classifiers are modified and updated.

The LogFileAnalyzer can help to better understand this dynamic process by visualizing classifier sets in tables and diagrams:

- For each iteration in an LCS experiment, the classifier population, the match set, and the action set can be displayed in separate tables. The tables can be sorted using any classifier property as criterion. Therefore, classifiers of special interest (e. g. those having a high prediction) can easily be determined for further investigation. The development of classifier sets over time can be tracked iterationwise or by directly addressing the sets of an iteration by its number.

- Classifier sets can be visualized using histograms. Histograms are created automatically for each classifier property and visualize the distribution of classifiers with respect to the displayed property. Filters can be applied to each histogram, additionally charts can be saved and printed.

- Single classifiers can be compared to all other classifiers in a set. Diagrams display the classifier's properties relative to the minimum and maximum values of the containing set.

Figure 1 shows a screenshot of the LogFileAnalyzer.

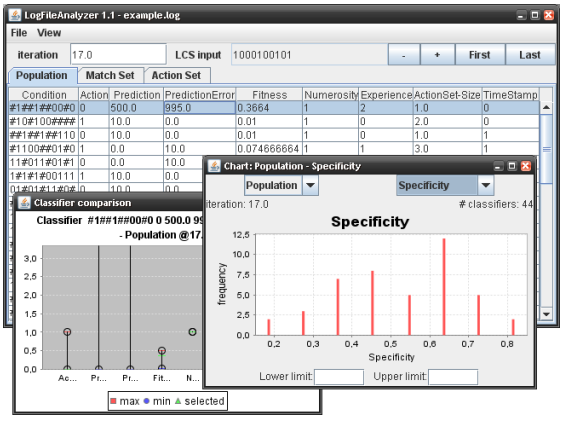

Figure 1: A screenshot of the LogFileAnalyzer

2 Getting started

The LogFileAnalyzer is written in Java. To install and start the program, follow these steps:

- Uncompress the contents of `lcsgui_v1-1.zip` to a local folder.
- Check the system requirements:
  - A Java Runtime Environment (JRE 6 or later) is needed to run the program. The latest JRE is available from [3].
  - To create histograms, the LogFileAnalyzer relies on JFreeChart which is available from [2]. Download and uncompress the JFreeChart package and copy the contents of its `/lib`-folder to the `/lib/ext`-folder of your JRE. Without JFreeChart, the LogFileAnalyzer will run, but no histograms can be created.
- Start the LogFileAnalyzer by double clicking `lcsgui_v1-1.jar`.
- Open the `example.log` to check the functionality of the program.

3 The LogFileAnalyzer classes

This section briefly describes the most important classes of the LogFileAnalyzer divided by packages.

3.1 Main class and data storage (Package de.dfg.oc.logfileanalyzer)

The main package contains classes necessary for starting the program and storing the classifier sets.

LogFileAnalyzer The `LogFileAnalyzer` is the program's main class. It determines how log-files are imported and how histograms are created. Furthermore, it defines the column names used in the program's tables. A `LogFileAnalyzer` is a singleton, its instance can be obtained by using the static `getInstance()`-method of this class. To adapt the program to your needs, use the `setDataImporter()`-, `setHistograms()`-, and `setColumnNames()`-methods. To run the program, call its `startLogFileAnalyzer()`-method.

DataImporterInterface Classes that read LCS log-files need to implement this interface by providing the `getNextDataElement()`-method specified here. The method should read a log-file and return the classifier sets for one iteration.

DefaultDataImporter The `DefaultDataImporter` is an example class that implements the `DataImporterInterface`. It is used to read the `example.log` provided with the LogFileAnalyzer.

DataElement A `DataElement` stores all classifier sets (i. e. population, match set, and action set) of one iteration. Each set is stored in a `DefaultTableModel` that can be directly displayed in the user interface. Furthermore, a `DataElement` contains references to the `DataElement`s of the previous and next iteration.

DataMemory This class stores the `DataElement`s of all iterations of an experiment. It relies on the `DataImporterInterface` to read complete log-files and provides a method to search for `DataElement`s by their iteration.

3.2 Histograms (Package de.dfg.oc.logfileanalyzer.histograms)

The `histograms`-subpackage contains classes necessary to create histograms.

AbstractHistogram Inherit from this abstract class when creating new histograms. The `calculateHistogramData()`-method has to provide the data displayed in the histogram.

UniveralHistogram The `UniversalHistogram`-class extends `AbstractHistogram`. It can be used to create a histogram for any table column by simply passing the column name to its constructor.

3.3 Graphical User Interface (Package de.dfg.oc.logfileanalyzer.gui)

The `gui`-subpackage contains classes defining the graphical user interface of the program.

TableFrame This class provides the main window containing the menu, buttons to step through the iterations, and tables displaying the classifier sets.

ChartFrame This class provides a frame containing the histogram chart.

4 Modifications to the LogFileAnalyzer

4.1 Create your own import method

To import your own log-files, you need to create your own class that implements the `DataImporterInterface`. The class needs to define the method `getNextDataElement()` which returns a `DataElement` containing the classifier sets belonging to the next unprocessed iteration. A `BufferedReader` is provided as parameter to access the log-file. See Listing 1 for a brief example implementation or compare the `DefaultDataImporter`-class for details.

Listing 1: Create a `DataImporter`

```
class MyDataImporter implements DataImporterInterface {
  public DataElement getNextDataElement(BufferedReader _bR) {
    // Create a DataElement to store classifier sets...
    DataElement newElement = new DataElement();
    // Obtain table to store classifier population...
```

```
    DefaultTableModel pop = newElement.getPopulation();
    try {
      // Read classifiers from log-file...
      String currentLine = _bR.readLine();
      while (currentLine != null) {
        // Split classifier (condition, action, prediction)...
        String[] clParts = currentLine.split("␣");
        // Add classifier to set...
        pop.addRow(clParts);
        // Read next classifier...
        currentLine = _bR.readLine();
      }
    } catch (Exception e) {
      // Error handling...
    }
    // Add set to DataElement...
    newElement.setPopulation(pop);
    // Repeat for match and action set and finally...
    return newElement;
  }
}
```

To use your newly created class, find the `main()`-method of the `LogFileAnalyzer`-class, create an instance of your `DataImporter` and pass it to the `setDataImporter()`-method of the `LogFileAnalyzer` (see Listing 2).

Listing 2: Use your `DataImporter`

```
public static void main(String[] args) {
  LogFileAnalyzer lfa = LogFileAnalyzer.getInstance();
  // Use your DataImporter...
  lfa.setDataImporter(new MyDataImporter());
  lfa.startLogFileAnalyzer();
}
```

4.2 Change the name of table columns

Column names can be changed from the `LogFileAnalyzer`'s `main()`-method. Simply define your names and pass them to the `setHeaders()`-method (see Listing 3).

Listing 3: Change the name of table columns

```
public static void main(String[] args) {
  LogFileAnalyzer lfa = LogFileAnalyzer.getInstance();
  // Define your column names...
  String[] myColumnNames = {"Cond", "Act", "Pred"};
  lfa.setHeaders(myColumnNames);
  lfa.startLogFileAnalyzer();
}
```

4.3 Create your own histograms

To add your own histogram, you need implement a new histogram class. The class needs to extend the class `AbstractHistogram` (located in the `histogram`-package) and implement the abstract method `calculateHistogramData()`. The method receives the currently selected classifier set as parameter and should return a `Vector` containing the data that will be displayed in the histogram. Listing 4 shows an implementation of the `calculateHistogramData()`-method that calculates the specificity of classifier conditions and stores this data in a `Vector` that is returned for display. The complete class `SpecificityHistogram` is available in the `histogram`-package.

Listing 4: Create your own histograms

```
Vector<Double> calculateHistogramData(DefaultTableModel _table) {
  // Store specificity here...
  Vector<Double> dataVector = new Vector<Double>();
  // Get column id for condition...
  int column = _table.findColumn("Condition");
  // For all classifiers...
  for (int row = 0; row < _table.getRowCount(); row++) {
    // Read condition...
    String conditionString = (String)_table.getValueAt(row, column);
    // Determine number of (non-specified) bits...
    char[] conditionBits = conditionString.toCharArray();
    int numberOfBits = conditionBits.length;
    int numberOfSpecifiedBits = 0;
    for (int i = 0; i < numberOfBits; i++) {
      if (!(conditionBits[i] == '#')) {
        numberOfSpecifiedBits++;
      }
    }
    // Calculate specificity...
    double specificity = (double)numberOfSpecifiedBits
                                        / (double)numberOfBits;
    dataVector.add(specificity);
  }
  return dataVector;
}
```

Finally, you need to register your histogram class. Therefore, find the `main()`-method of the `LogFileAnalyzer`-class, create an instance of your histogram class, and pass it to the `LogFileAnalyzer`'s `addHistogram()`-method (see Listing 5).

Listing 5: Add your newly created histogram

```
public static void main(String[] args) {
  LogFileAnalyzer lfa = LogFileAnalyzer.getInstance();
  // Add your histogram...
  lfa.addHistogram(new SpecificityHistogram("Specificity"));
  lfa.startLogFileAnalyzer();
}
```

4.4 Compiling your changes

Besides JFreeChart [2], a Java Development Kit (JDK 5 or later) is required to compile your changes. The latest JDK is available from [3]. To compile, change to the `src`-folder of the LogFileAnalyzer and use the following command (`path\to\jfreechart\libs` is the folder containing the JFreeChart libraries):

```
javac -extdirs path\to\jfreechart\libs de\dfg\oc\logfileanalyzer\*.java
```

To run your code, execute `java de.dfg.oc.logfileanalyzer.LogFileAnalyzer` from the `src`-folder.

5 License and further documentation

The LogFileAnalyzer is licensed under the terms of the GNU General Public License. If you want to make further modifications to the program, please consider the JavaDoc available in the `doc`-folder.

Acknowledgement

We would like to thank Jürgen Branke and Urban Richter (both from the Karlsruhe Institute of Technology) as well as Fabian Rochner (Leibniz Universität Hannover) for their valuable suggestions. The development of the LogFileAnalyzer was financially supported by the German Research Foundation (DFG) within the priority program 1183 "Organic Computing".

References

[1] M. V. Butz. *Rule-Based Evolutionary Online Learning Systems – A Principled Approach to LCS Analysis and Design.* Springer, 2005.

[2] Object Refinery Limited. JFreeChart web page. `http://www.jfree.org/jfreechart/`.

[3] Sun Microsystems. Java web page. `http://java.sun.com/`.