
JCLEC Classification Module

USER MANUAL AND DEVELOPER DOCUMENTATION

Last update on July 2, 2014.

CONTENTS

1	Introduction to JCLEC	5
2	JCLEC classification module	7
3	Documentation for users	9
3.1	Download and install	9
3.2	Compilation	18
3.3	Creating the configuration file	18
3.4	Executing the algorithms	19
3.4.1	Falco classification algorithm	21
3.4.2	Tan classification algorithm	23
3.4.3	Bojarczuk classification algorithm	25
3.5	10-fold cross-validation experimentation	27
3.6	Parameter optimization and batch processing	28
3.7	Using JCLEC classification algorithms in WEKA	29
4	Documentation for developers	31
4.1	Package structure	31
4.1.1	net.sf.jclec.problem.classification	32
4.1.2	net.fs.jclec.problem.classification.base	32
4.1.3	net.fs.jclec.problem.classification.blocks	33
4.1.4	net.fs.jclec.problem.classification.blocks.fuzzy	34
4.1.5	net.fs.jclec.problem.classification.crisprule	34

4.1.6	net.fs.jclec.problem.classification.fuzzyrule	35
4.1.7	net.fs.jclec.problem.classification.exprttree	35
4.1.8	net.fs.jclec.problem.classification.multiexprtree	35
4.1.9	net.fs.jclec.problem.classification.multisyntaxtree	36
4.1.10	net.fs.jclec.problem.classification.syntaxtree	36
4.1.11	net.fs.jclec.problem.classification.listener	36
4.2	API reference	36
4.3	Implementing new algorithms	37
4.3.1	Algorithm	37
4.3.2	Evaluator	43
4.3.3	Species	45
4.4	Exporting new algorithms into WEKA	47
4.5	Running unit tests	49
5	Citation	51
	Bibliography	53

1. INTRODUCTION TO JCLEC

JCLEC [1] is an evolutionary computation (EC) framework developed in the Java programming language. The project started as a class library in 1999. In the years 2010–2011 the software has been completely improved in order to resolve some fundamental problems in the architecture and today it is in its fourth major version. It was released with the GNU general public licence (GPL) and it is hosted as a free software project in the SourceForge page (<http://jclec.sourceforge.net>).

JCLEC provides a high-level software environment to do any kind of Evolutionary Algorithm (EA), with support for genetic algorithms (binary, integer and real encoding), genetic programming (Koza style, strongly typed, and grammar based) and evolutionary programming.

JCLEC architecture follows strong principles of object oriented programming, where abstractions are represented by loosely coupled objects and where it is common and easy to reuse code. JCLEC provides an EC environment that its main features are:

- *Generic.* JCLEC is able to execute any kind of EC algorithm. So far, JCLEC supports most mainstream EC flavors such genetic programming, bit string, integer-valued vector and real-valued vector genetic algorithms, and evolution strategy. It also includes support for advanced EC techniques such as multiobjective optimization.
- *User friendly.* JCLEC possesses several mechanisms that offer a user friendly programming interface. The programming style promoted is high-level and allows rapid prototyping of applications.
- *Portable.* The JCLEC system has been coded in the Java programming language that ensures its portability between all platforms that implement a Java Virtual Machine.
- *Efficient.* A particular attention was given to optimization of critical code sections.
- *Robust.* Verification and validation statements are embedded into the code to ensure that operations are valid and to report problems to the user.
- *Elegant.* The interface of JCLEC was developed with care. Great energy was invested in designing a coherent software package that follows good object oriented and generic programming principles. Moreover, strict programming rules were enforced to make the code easy to read, understand and, eventually, modify. The use of XML file format is also a central aspect of JCLEC, which provides a common ground for tools development to analyze and generate files, and to integrate the framework with other systems.
- *Open Source.* The source code of JCLEC is free and available under the GNU General Public License (GPL). Thus, it can be distributed and modified without any fee.

The JCLEC software project provides a complete tutorial at the SourceForge page <http://jclec.sourceforge.net/data/JCLEC-tutorial.pdf>, including a series of algorithms to solve well-known problems such as the knapsack problem or the traveling salesman problem.

2. JCLEC CLASSIFICATION MODULE

The classification models are being applied in different areas such as bioinformatics, marketing, banks or web mining. There exists classification libraries that provide algorithms following different methodologies. However, it is difficult to find a library that contains genetic programming (GP) algorithms. GP is an evolutionary learning technique that offers a great potential for classification tasks [2]. This flexible heuristic technique allows us to use complex pattern representation such as trees, where any kind of operation or function can be used inside that representation and domain knowledge can be used in the learning process.

The JCLEC classification module is an intuitive, usable and extensible open source module for GP classification algorithms. This module is open source software for researchers and end-users to develop and use classification algorithms based on GP and grammar guided genetic programming (G3P) models [3], which makes the knowledge extracted more expressive and flexible by means of a context-free grammar. The JCLEC classification module houses implementations of rule-based methods for classification based on GP, supporting multiple model representations and providing to the users the tools to easily implement any classifier.

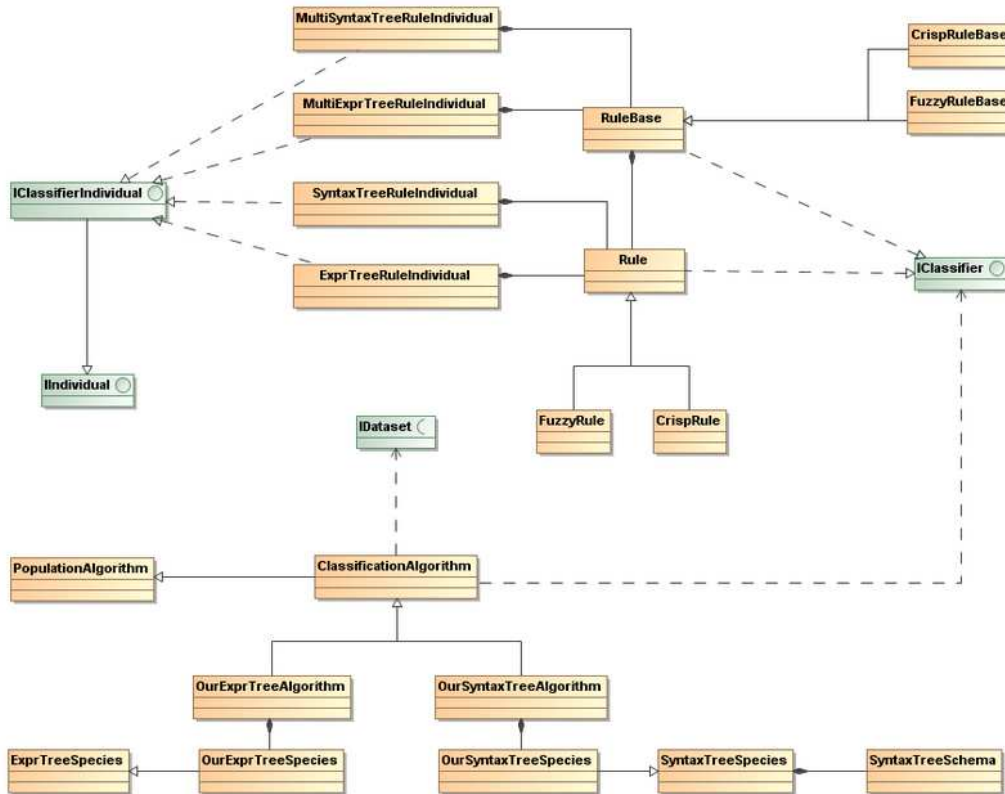


Figure 2.1: Class diagram for the JCLEC classification module

The classification module (Figure 2.1 depicts the class diagram) presents a hierarchical structure — this hierarchical structure is properly described in Section 4.1 — rooted by a base package, named *net.sf.jclec.problem.classification*, which provides the abstract classes with the properties and methods that any classification algorithm must contain. This module also provides a package to make reporters for the train and test classification partitions, indicating the classification performance of classifiers obtained by algorithms. This package contains a class with methods to make reports specifying the classifier features such as the rule base, the number of rules, the average number of conditions, the percentage of correct predictions (accuracy), the percentage of correct predictions per class, the geometric mean, the area under the curve, the Cohen’s kappa rate and the confusion matrix (error matrix).

For the sake of loading different formatted files, such as KEEL or ARFF, a series of utility classes are provided by the dataset package. Three different attribute types may be represented by this package, integer, continuous and categorical, and also a number of characteristics from the dataset are given, comprising type of attributes, number of classes, number of instances, etc.

Finally, the classification module includes some GP and G3P proposals described in the literature, and provides the necessary classes and methods to develop any kind of evolutionary algorithm for easily solving classification problems.

3. DOCUMENTATION FOR USERS

This chapter describes the process for end-users and developers to download, install and use the classification algorithms included in the JCLEC module.

3.1 Download and install

The classification module together with the JCLEC core can be obtained in one of the following ways, which are adapted to the user's preferences:

- Download runnable jar file or source files from SourceForge.net:

The runnable jar file provides to the user the fastest and easiest way of executing an experiment whereas the source files allow developers to implement personalized algorithms.

Download the files provided at <http://sourceforge.net/projects/jclec/files/>.

- Download source files from SVN.

Anonymous SVN access is also available via SourceForge. To access the source code repository you can use one of the following ways:

- Browse source code online (<https://sourceforge.net/p/jclec/svn/HEAD/tree/trunk/>) to view this project's directory structure and files.
- Check out source code with a SVN client using the following command:

```
svn checkout svn://svn.code.sf.net/p/jclec/svn/trunk/jclec4-base jclec4-base
svn checkout svn://svn.code.sf.net/p/jclec/svn/trunk/jclec4-classification jclec4-classification
```

- Download source files from GIT.

Anonymous GIT access is also available via SourceForge. To access the source code repository you can use one of the following ways:

- Browse source code online (<https://sourceforge.net/p/jclec/git/ci/master/tree/>) to view this project's directory structure and files.
- Check out source code with a GIT client using the following command:

```
git clone git://git.code.sf.net/p/jclec/git jclec
```

- Download source files from CVS.

Anonymous CVS access is also available via SourceForge. To access the source code repository you can use one of the following ways:

- Browse source code online (<http://jclec.cvs.sourceforge.net/>) to view this project's directory structure and files.
- Check out source code with a CVS client using the following command:

```
cvs -d : anonymous@jclec.cvs.sourceforge.net:/cvsroot/jclec checkout jclec4-base
cvs -d : anonymous@jclec.cvs.sourceforge.net:/cvsroot/jclec checkout jclec4-classification
```

- Download source files and import as project in Eclipse.

You can download JCLEC modules using CVS and import them as Eclipse projects.

Click on File → New → Project.

Select CVS → Project from CVS.

Fill the form using the following data:

Host: `jclec.cvs.sourceforge.net`

Repository path: `/cvsroot/jclec`

User: `anonymous`

Checkout from CVS

Enter Repository Location Information
Define the location and protocol required to connect with an existing CVS repository.

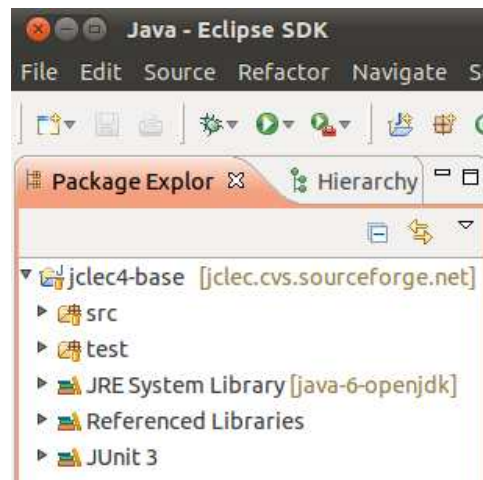
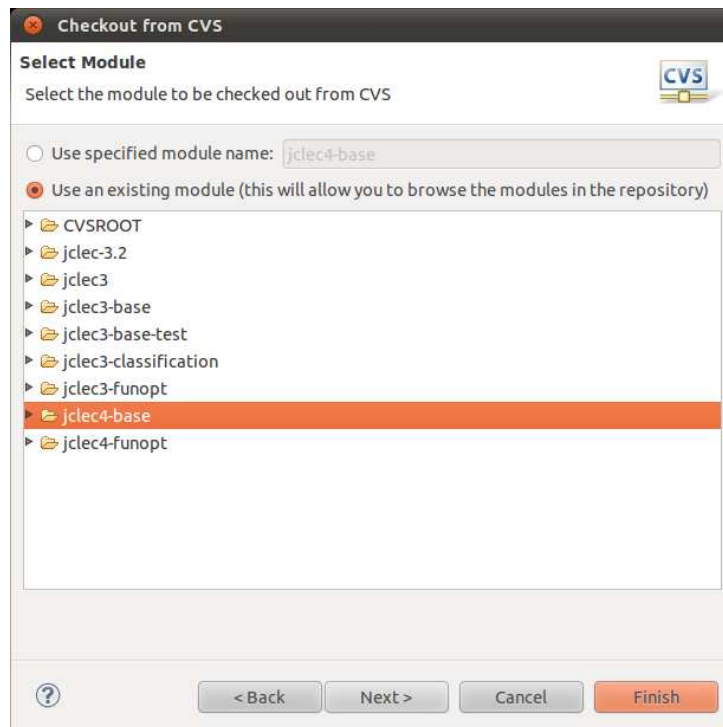
Location
Host: `jclec.cvs.sourceforge.net`
Repository path: `/cvsroot/jclec`

Authentication
User: `anonymous`
Password:

Connection
Connection type: `pserver`
☒ Use default port
☐ Use port:

☐ Save password (could trigger secure storage login)
To manage your password, please see '[Secure Storage](#)'
[Configure connection preferences...](#)

Click on use an existing module and select `jclec4-base`, `jclec4-classification` or any other wanted modules. Finally, `jclec4-base` and `jclec4-classification` projects source code and their libraries are imported into your workspace.



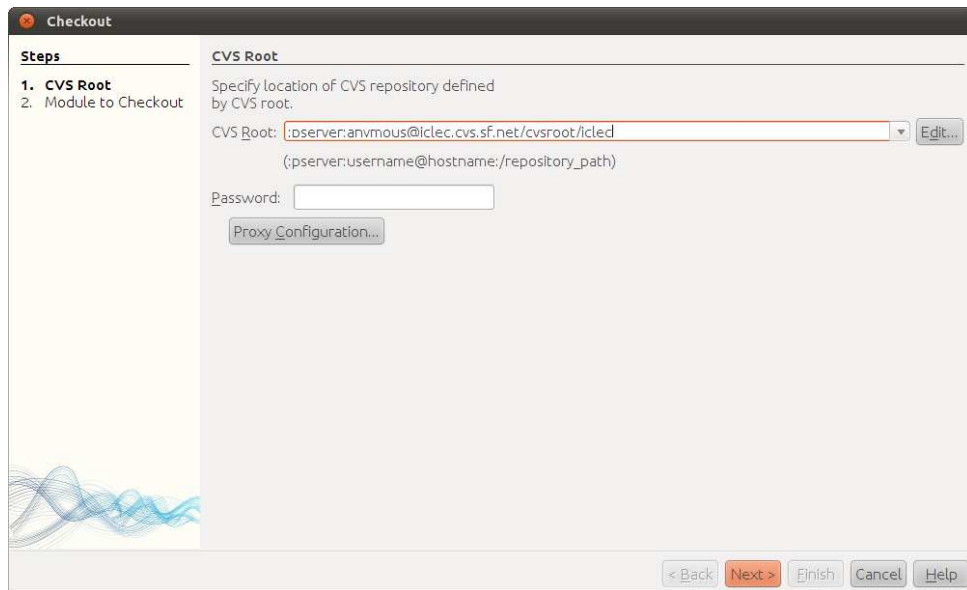
- Download source files and import as project in NetBeans.

You can also download JCLEC modules using CVS and import them as NetBeans projects.

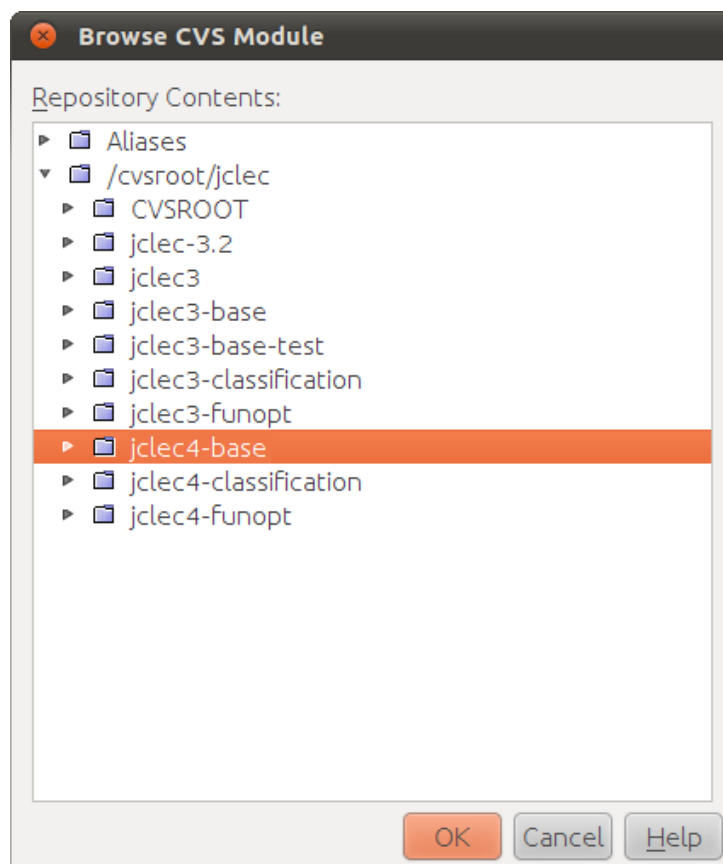
Click on Team → CVS → Checkout.

Fill the form using the following data:

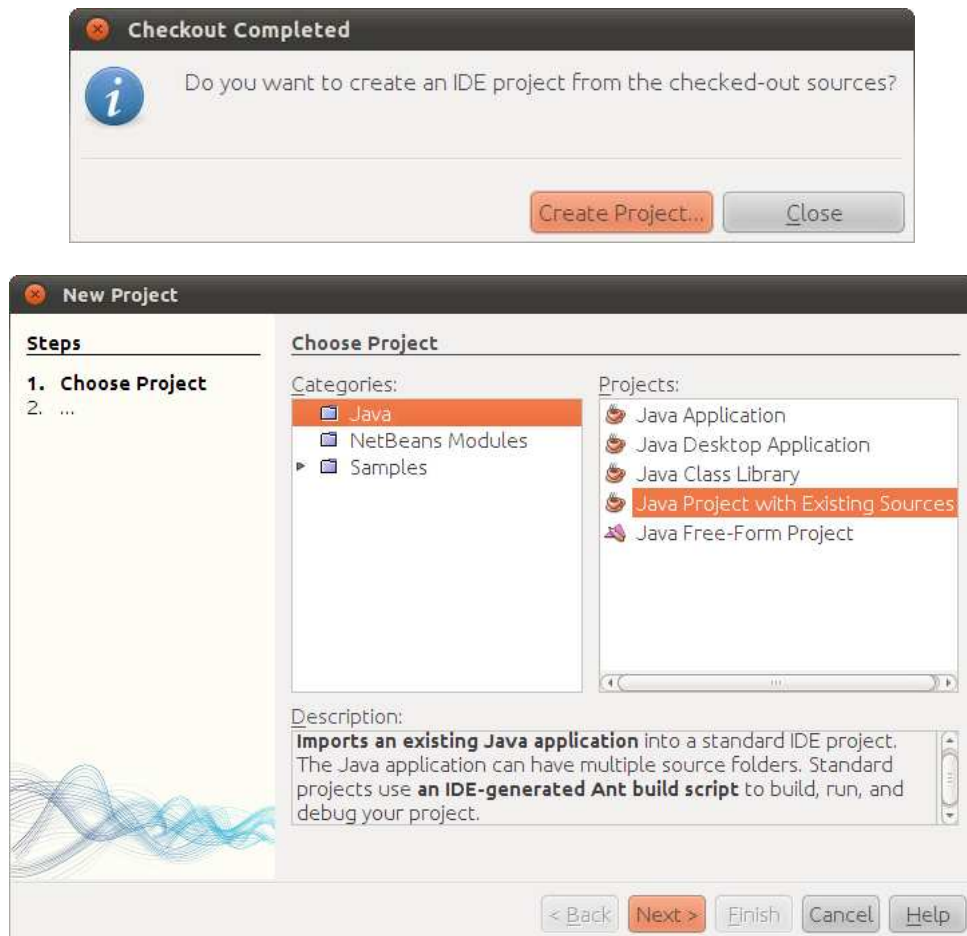
CVS Root : `anonymous@jclec.cvs.sf.net/cvsroot/jclec`



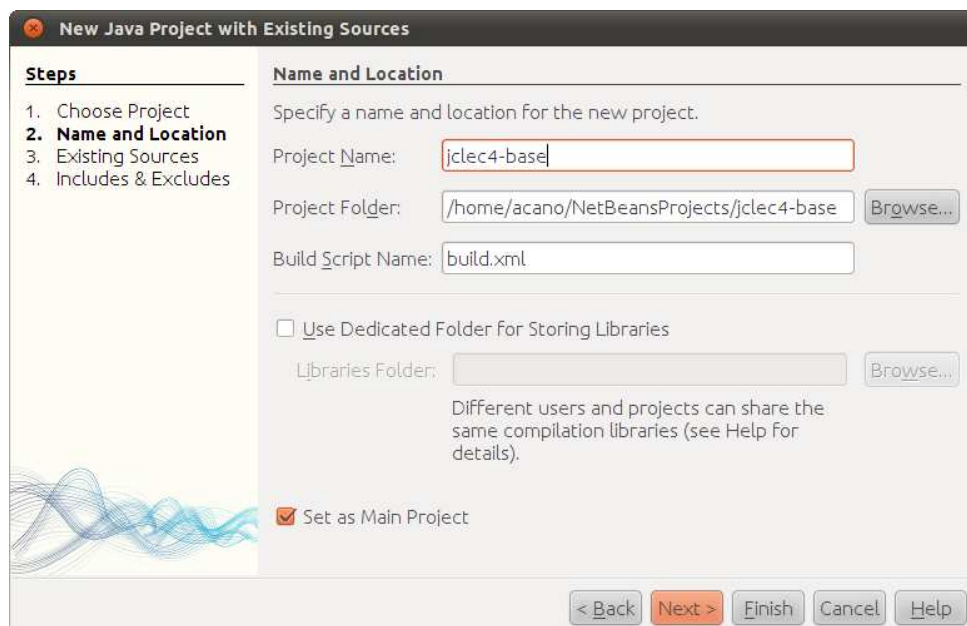
Browse an existing CVS module and select jclec4-base, jclec4-classification or any other wanted modules.



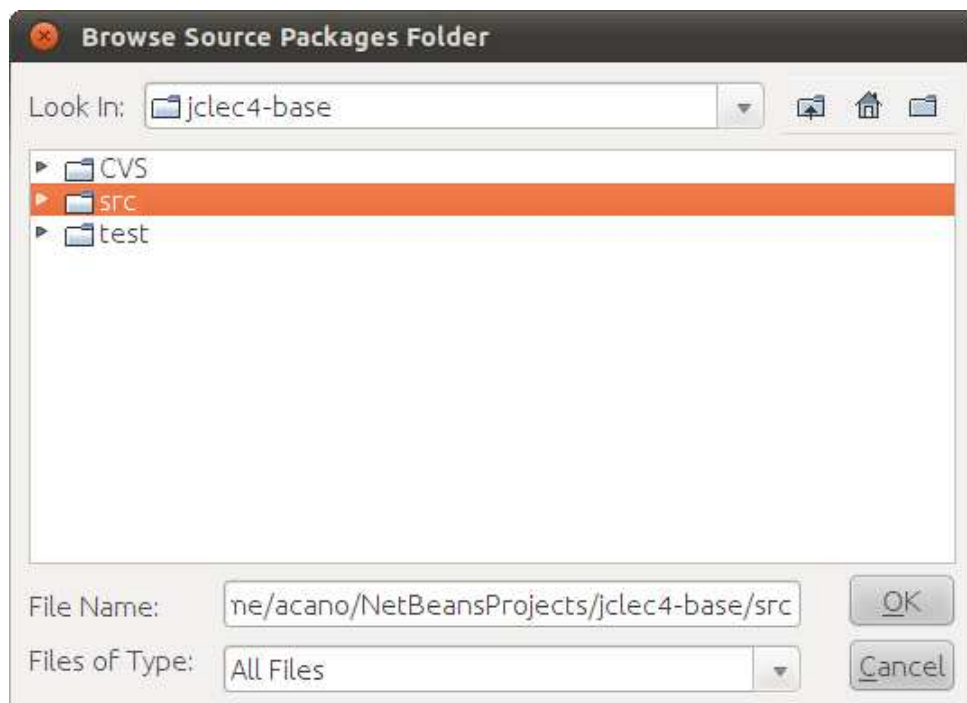
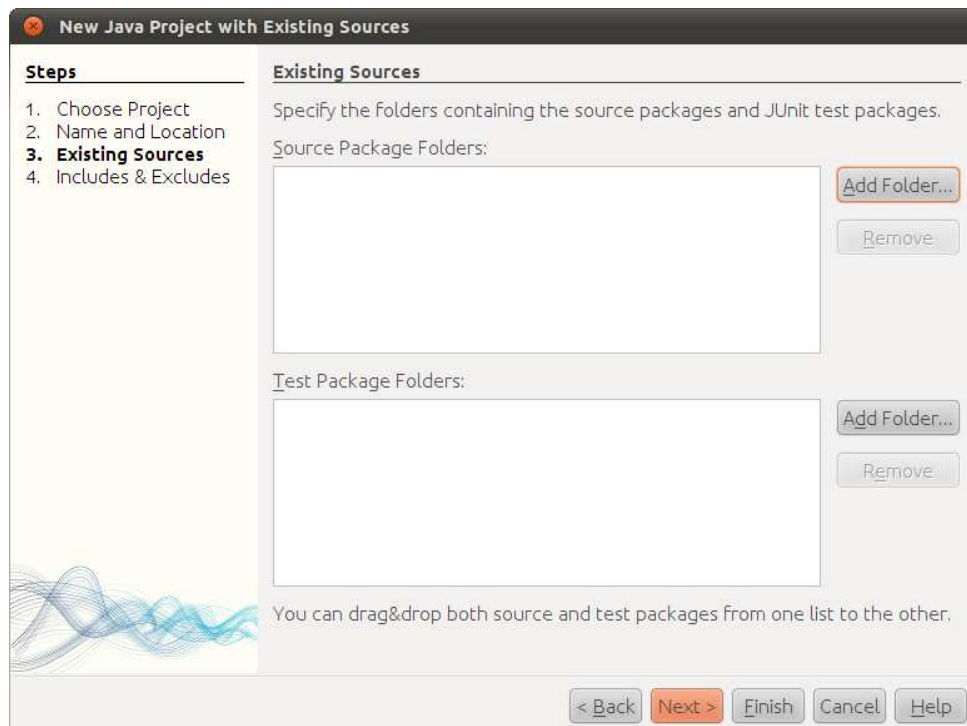
Click on create Project from checked-out source files.

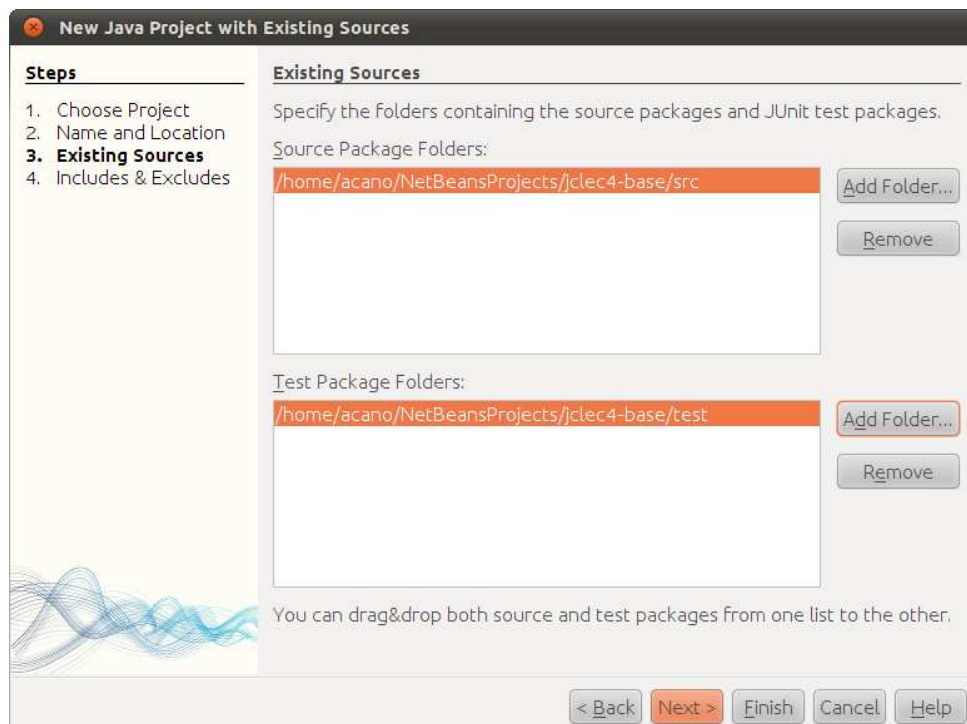
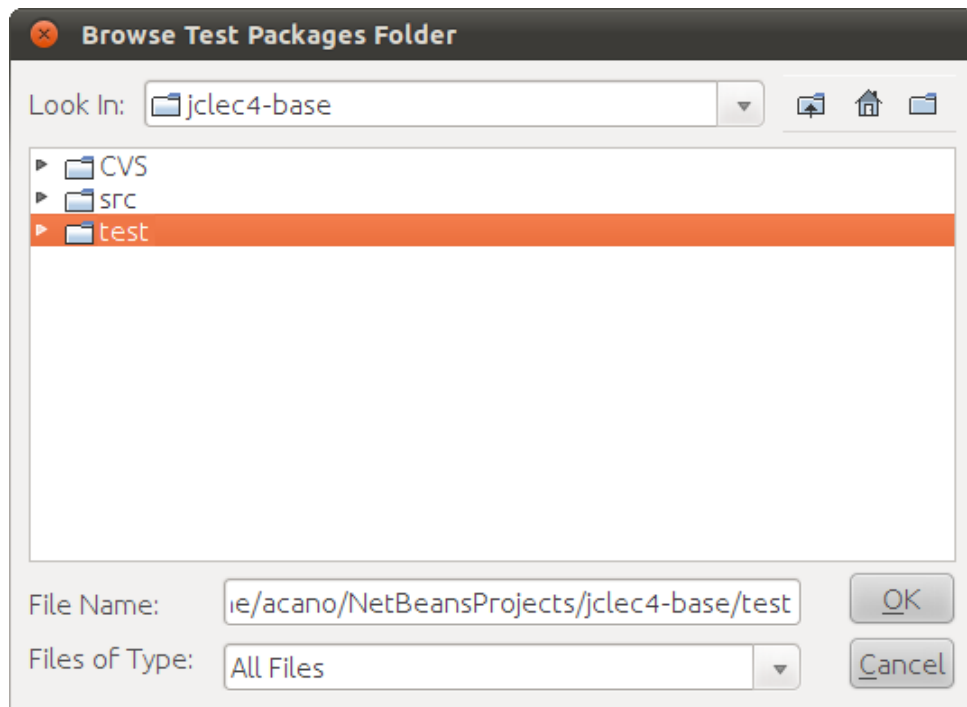


Specify jclec4-base and jclec4-classification as the project names.

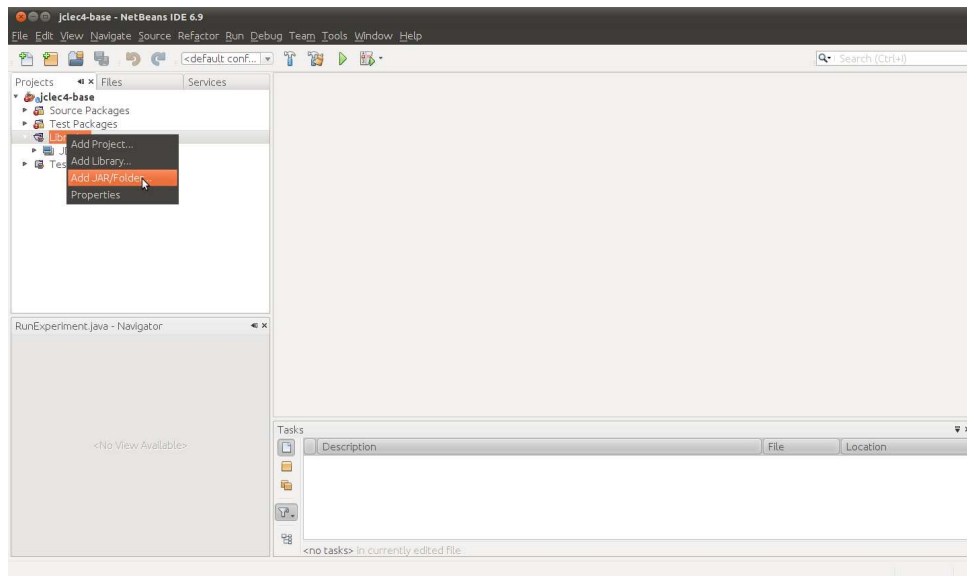


Specify the source and test folders.

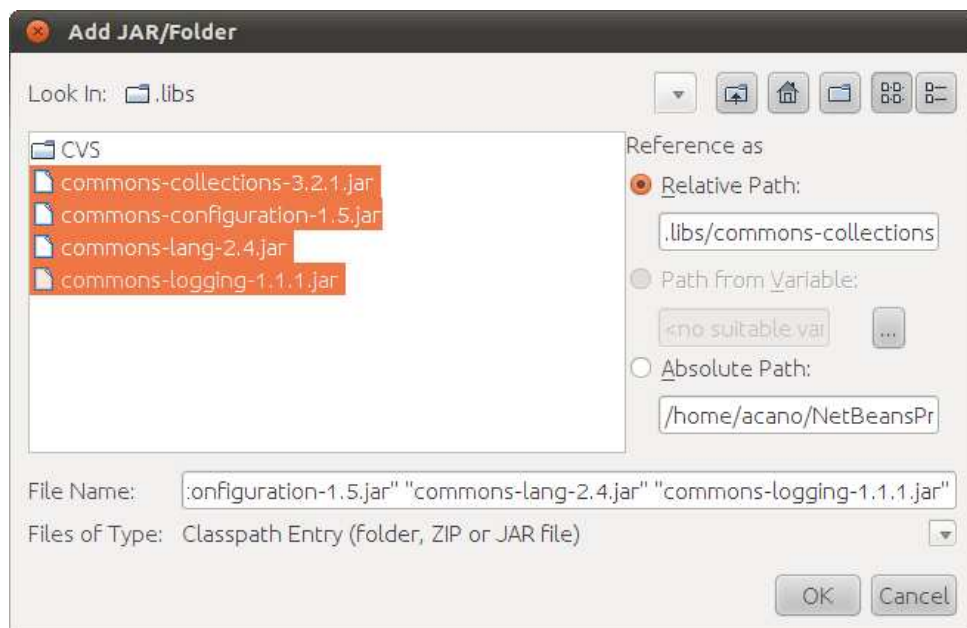




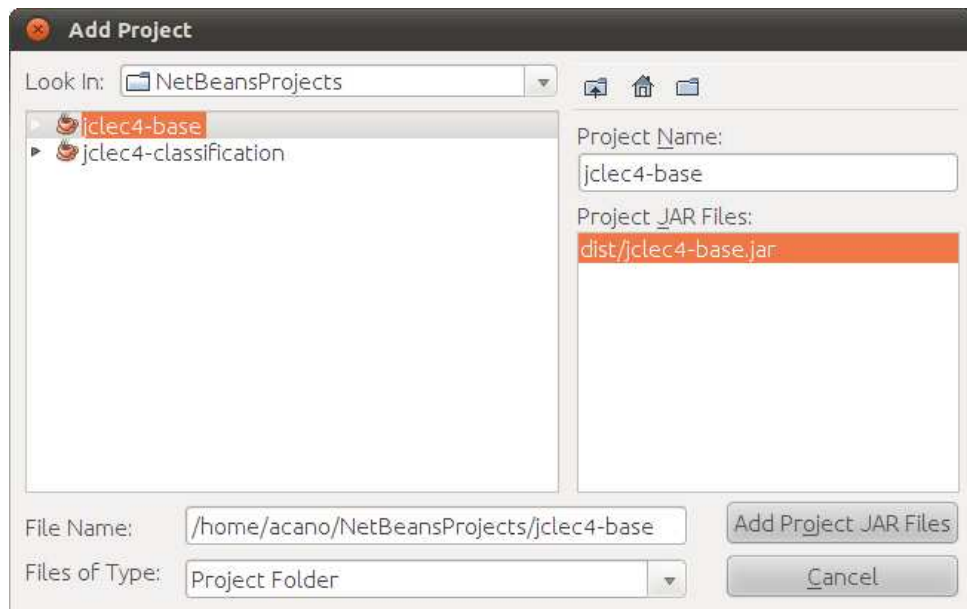
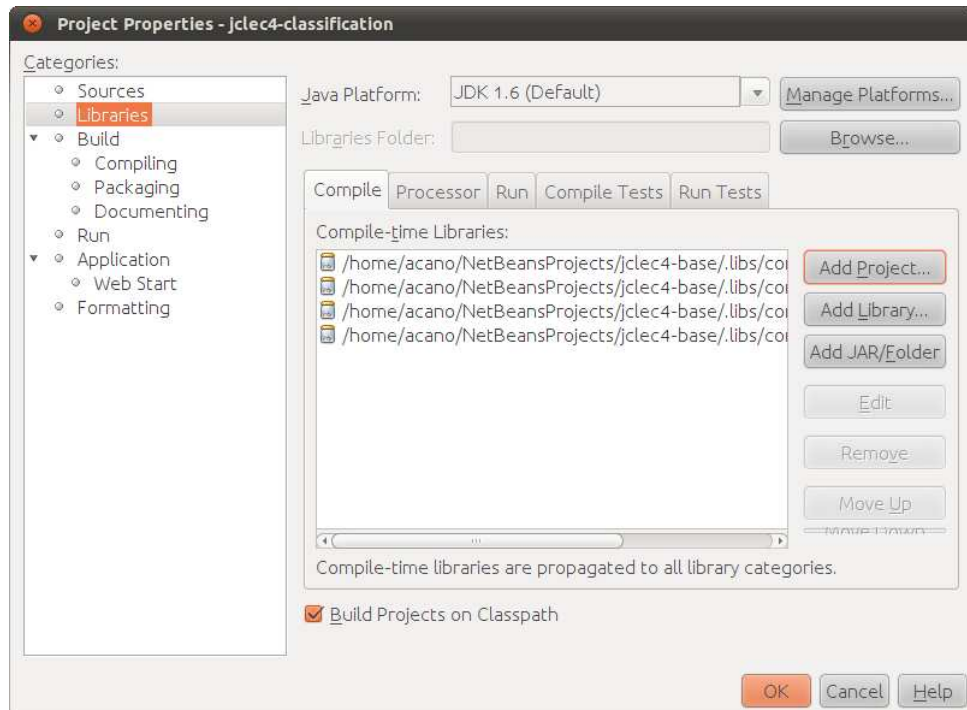
Add required libraries, right click Libraries folder and select Add JAR Folder.



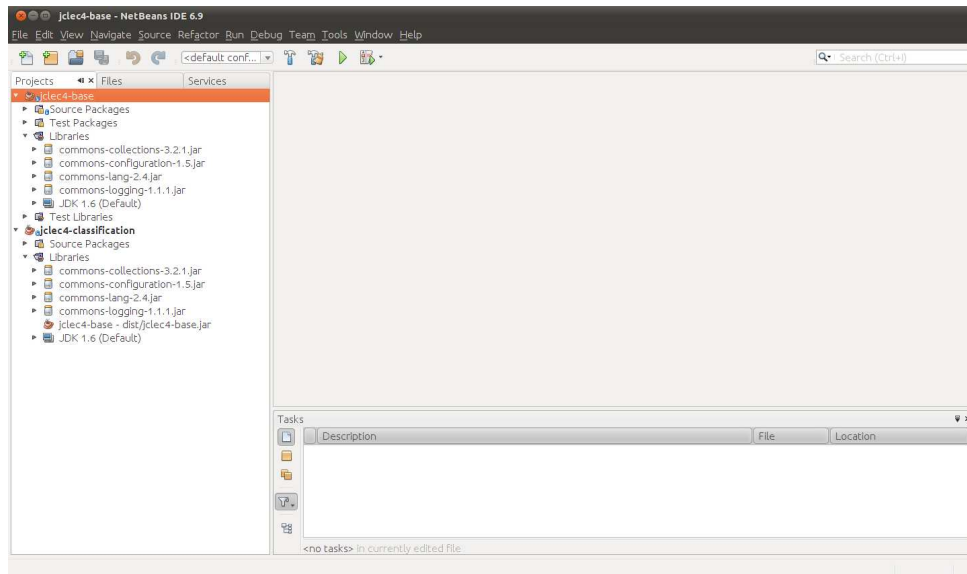
Add the jar files provided in the libs folder.



If jclec4-classification module is imported, add the jclec4-base project to Libraries in the project properties.



Finally, you are ready to run `jclec4-base` and `jclec4-classification` algorithms!



3.2 Compilation

The source code can be compiled in two ways:

1. Using the ant compiler and the build.xml file.
Open a terminal console located in the project path and type “`ant`”.
A jar file named “`jclec4-classification.jar`” will be created.
For cleaning the project just type “`ant clean`”.
2. Using the maven compiler and the pom.xml file.
Open a terminal console located in the project path and type “`mvn package`”.
A jar file named “`jclec4-classification.jar`” will be created.
For cleaning the project just type “`mvn clean`”.

3.3 Creating the configuration file

To execute any algorithm in the JCLEC classification module, an XML format configuration file is required. This configuration file comprises a series of parameters which define the settings of the algorithm and its operators. The most important parameters are described as follows:

- Algorithm type: this parameter establishes the algorithm used to solve the classification problem. It is required to choose one from the available in the package:
net.sf.jclec.problem.classification.algorithm

```
<process algorithm-type="net.sf.jclec.problem.classification.algorithm.tan.TanAlgorithm">
```

- Maximum number of generations: this parameter determines the maximum number of generations as a stopping criterion. Once this number is reached, the algorithm finishes returning its results. A sample of 100 generations is shown:

```
<max-of-generations>100</max-of-generations>
```

- Population size: specifies the number of individuals that form the population. A sample of 100 individuals is shown:

```
<population-size>100</population-size>
```

- Train and test files: this parameter allows determining the train and test dataset files. Firstly, the type of dataset to be used is determined by using the package *net.sf.jclec.problem.util.dataset*. Secondly, the train and test files are specified. Finally, the attribute that represents the data class in the dataset metadata is established. Following, a sample ArffDataset is determined:

```
<dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
  <train-data>data/iris/iris-10-1tra.arff</train-data>
  <test-data>data/iris/iris-10-1tst.arff</test-data>
  <attribute-class-name>Class</attribute-class-name>
</dataset>
```

- Genetic operators: both the crossover probability and the mutation probability are specified by using these parameters:

```
<recombination-prob>0.8</recombination-prob>
<mutation-prob>0.1</mutation-prob>
```

- Listener: regardless of the algorithm used, a listener to display and save the results should be determined from the package *net.sf.jclec.problem.classification.listener*. In the following example, a report is made every 10 generations. Furthermore, the name of the folder that comprises the reports is reportExample. Finally, a global report named summaryExample that collects summary results is made once the algorithm finishes.

```
<listener type="net.sf.jclec.problem.classification.listener.RuleBaseReporter">
  <report-dir-name>reports/reportExample</report-dir-name>
  <global-report-name>summaryExample</global-report-name>
  <report-frequency>10</report-frequency>
</listener>
```

3.4 Executing the algorithms

Once the user has created the configuration file which specifies the parameters and settings for the algorithm, there are three ways to execute the algorithm.

Using JAR file

You can execute any algorithm using the JAR file. It should be noted that your JAR file should also include the JCLEC core and the classification module. For instance, you could run the Falco algorithm, which is provided in the classification module, using the following command:

```
java -jar jclec4-classification.jar examples/Falco.cfg
```

Using Eclipse

You can execute JCLEC algorithms using Eclipse.

Click on Run → Run Configurations.

Create a new launch configuration as Java Application.



```
Project jclec4-classification
Main class: net.sf.jclec.RunExperiment
Program arguments: examples/Falco.cfg
```

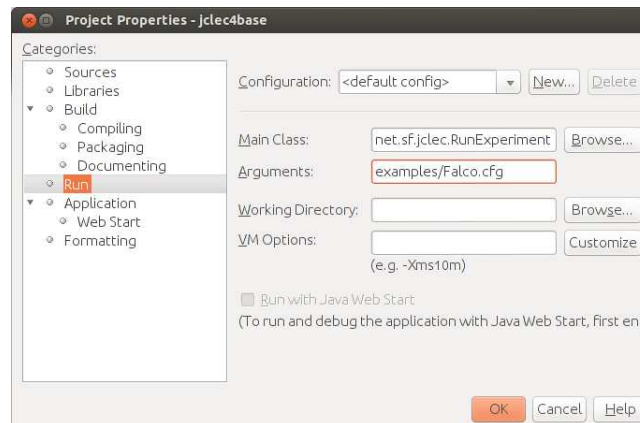
Finally, we execute our algorithm by clicking on the Run button. Notice that *Falco.cfg* is an example configuration file, so the user has to create, setup and select his own configuration file for his algorithm. For instance, the user could use any of the three available example configuration files, *Falco.cfg*, *Bojarczuk.cfg* and *Tan.cfg*.

In this chapter, the three example classification algorithms are executed, showing the configuration file and the output obtained when running them. These three classification algorithms are available in three sub-packages (*bojarczuk*, *falco*, *tan*) included within the package *net.sf.jclec.problem.classification.algorithm*.

Using NetBeans

You can execute JCLEC algorithms using NetBeans.

```
Click on Project properties → Run
Set the main class: net.sf.jclec.RunExperiment
Set the program arguments: examples/Falco.cfg
```



3.4.1 Falco classification algorithm

This algorithm implements Falco et al. [4] “*Discovering interesting classification rules with genetic programming*”. The configuration file comprises a series of parameters required to run the algorithm. Following, the Falco algorithm configuration file is shown.

```
<experiment>
<process algorithm-type="net.sf.jclec.problem.classification.algorithm.falco.FalcoAlgorithm">
  <rand-gen-factory seed="123456789" type="net.sf.jclec.util.random.RanecuFactory"/>
  <population-size>100</population-size>
  <max-of-generations>100</max-of-generations>
  <max-deriv-size>20</max-deriv-size>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-1tra.arff</train-data>
    <test-data>data/iris/iris-10-1tst.arff</test-data>
    <attribute-class-name>Class</attribute-class-name>
  </dataset>
  <alpha>0.9</alpha>
  <recombination-prob>0.8</recombination-prob>
  <mutation-prob>0.1</mutation-prob>
  <copy-prob>0.01</copy-prob>
  <listener type="net.sf.jclec.problem.classification.listener.RuleBaseReporter">
    <report-dir-name>reports/reportFalco</report-dir-name>
    <global-report-name>summaryFalco</global-report-name>
    <report-frequency>10</report-frequency>
  </listener>
</process>
</experiment>
```

After the algorithm is run, a folder containing the reports for the algorithm’s output is created. This folder contains the train and test classification reports, including the prediction for each data instance and a summary report which comprises information about the classifier inducted, its properties and several performance measures. There are many performance measures obtained from the confusion matrix which collects the real and predicted data class for the instances of the dataset. These are the accuracy, the geometric mean, the Cohen’s Kappa rate and the area under the curve; the last two are especially useful in imbalanced data problems. The report file also provides the runtime and shows the rules of the classifier.

Following, the output of the “TestClassificationReport.txt” file is shown:

```

File name: data/iris/iris-10-1tst.arff
Runtime (s.): 1.468
Number of different attributes: 4
Number of rules: 4
Number of conditions: 3
Average number of conditions per rule: 0,75
Accuracy: 0,9333
Geometric mean: 0,9283
Cohen's Kappa rate: 0,9000
AUC: 0,9667

#Percentage of correct predictions per class
Class Iris-setosa: 100,00%
Class Iris-versicolor: 100,00%
Class Iris-virginica: 80,00%
#End percentage of correct predictions per class

#Classifier
1 Rule: IF (<= PetalLength 2.158831 ) THEN (Class = Iris-setosa)
2 Rule: ELSE IF (> PetalWidth 1.722734 ) THEN (Class = Iris-virginica)
3 Rule: ELSE IF (IN PetalWidth 0.888823 1.72697936 ) THEN (Class = Iris-versicolor)
4 Rule: ELSE (Class = Iris-setosa)

#Test Classification Confusion Matrix
      Predicted
      C0  C1  C2  |
Actual C0  5   0   0  |  C0 = Iris-setosa
      C1  0   5   0  |  C1 = Iris-versicolor
      C2  0   1   4  |  C2 = Iris-virginica

```

3.4.2 Tan classification algorithm

This algorithm implements Tan et al. [5] “*Mining multiple comprehensible classification rules using genetic programming*”. Following, the Tan algorithm configuration file is shown.

```
<experiment>
<process algorithm-type="net.sf.jclec.problem.classification.algorithm.tan.TanAlgorithm">
  <rand-gen-factory seed="123456789" type="net.sf.jclec.util.random.RanecuFactory"/>
  <population-size>100</population-size>
  <max-of-generations>100</max-of-generations>
  <max-deriv-size>20</max-deriv-size>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-1tra.arff</train-data>
    <test-data>data/iris/iris-10-1tst.arff</test-data>
    <attribute-class-name>Class</attribute-class-name>
  </dataset>
  <w1>0.7</w1>
  <w2>0.8</w2>
  <recombination-prob>0.8</recombination-prob>
  <mutation-prob>0.1</mutation-prob>
  <copy-prob>0.01</copy-prob>
  <support>0.1</support>
  <elitist-prob>0.1</elitist-prob>
  <listener type="net.sf.jclec.problem.classification.listener.RuleBaseReporter">
    <report-dir-name>reports/reportTan</report-dir-name>
    <global-report-name>summaryTan</global-report-name>
    <report-frequency>10</report-frequency>
  </listener>
</process>
</experiment>
```

Following, the output obtained when running the algorithm is shown:

```

File name: data/iris/iris-10-1tst.arff
Runtime (s): 4.407
Number of different attributes: 4
Number of rules: 4
Number of conditions: 8
Average number of conditions per rule: 2,0
Accuracy (Percentage of correct predictions): 0,9333
Geometric mean: 0,9283
Cohen's Kappa rate: 0,9000
AUC: 0,9667

Percentage of correct predictions per class
Class Iris-setosa: 100,00%
Class Iris-versicolor: 100,00%
Class Iris-virginica: 80,00%
End percentage of correct predictions per class

Classifier
1 Rule: IF (AND NOT AND AND IN SepalLength 5.521539 7.4334537 < PetalWidth 1.579072
        > PetalWidth 1.275671 < PetalWidth 0.769985 ) THEN (Class = Iris-setosa)

2 Rule: ELSE IF (AND IN PetalWidth 0.582293 1.815772 IN PetalWidth 0.190182
        1.7987844 ) THEN (Class = Iris-versicolor)

3 Rule: ELSE IF (>= PetalLength 4.755571 ) THEN (Class = Iris-virginica)

4 Rule: ELSE (Class = Iris-setosa)

#Test Classification Confusion Matrix
      Predicted
      C0  C1  C2  |
Actual C0  5   0   0  |  C0 = Iris-setosa
      C1  0   5   0  |  C1 = Iris-versicolor
      C2  1   0   4  |  C2 = Iris-virginica

```


3.4.3 Bojarczuk classification algorithm

This algorithm implements Bojarczuk et al. [6] “*A constrained-syntax genetic programming system for discovering classification rules: application to medical data sets*”. Following, the Bojarczuk algorithm configuration file is shown.

```
<experiment>
<process
  algorithm-type="net.sf.jclec.problem.classification.algorithm.bojarczuk.BojarczukAlgorithm">
  <rand-gen-factory seed="123456789" type="net.sf.jclec.util.random.RanecuFactory"/>
  <population-size>100</population-size>
  <max-of-generations>100</max-of-generations>
  <max-deriv-size>20</max-deriv-size>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-1tra.arff</train-data>
    <test-data>data/iris/iris-10-1tst.arff</test-data>
    <attribute-class-name>Class</attribute-class-name>
  </dataset>
  <recombination-prob>0.8</recombination-prob>
  <copy-prob>0.01</copy-prob>
  <listener type="net.sf.jclec.problem.classification.listener.RuleBaseReporter">
    <report-dir-name>reports/reportBojarczuk</report-dir-name>
    <global-report-name>summaryBojarczuk</global-report-name>
    <report-frequency>10</report-frequency>
  </listener>
</process>
</experiment>
```

Following, the output obtained when running the algorithm is shown:

```

File name: data/iris/iris-10-1tst.arff
Runtime (s.): 1.65
Number of different attributes: 4
Number of rules: 4
Number of conditions: 4
Average number of conditions per rule: 1.0
Accuracy: 0,8000
Geometric mean: 0,7368
Cohen's Kappa rate: 0,7000
AUC: 0,9000

#Percentage of correct predictions per class
Class Iris-setosa: 100,00%
Class Iris-versicolor: 40,00%
Class Iris-virginica: 100,00%
#End percentage of correct predictions per class

#Classifier
1 Rule: IF (<= PetalLength 2.716659 ) THEN (Class = Iris-setosa)
2 Rule: ELSE IF (> PetalWidth 1.401309 ) THEN (Class = Iris-virginica)
3 Rule: ELSE IF (AND > SepalLength 5.086142 <= PetalWidth 1.572744 ) THEN (Class = Iris-versicolor)
4 Rule: ELSE (Class = Iris-setosa)

#Test Classification Confusion Matrix
      Predicted
      C0  C1  C2  |
Actual C0  5   0   0  |  C0 = Iris-setosa
      C1  0   2   3  |  C1 = Iris-versicolor
      C2  0   0   5  |  C2 = Iris-virginica

```

3.5 10-fold cross-validation experimentation

10-fold cross-validation is commonly used to validate the experimental results of an algorithm. In 10-fold cross-validation, the original dataset is partitioned into 10 subsamples. Of the 10 subsamples, a single subsample is retained as the validation data for testing the model, and the remaining 9 subsamples are used as training data. The cross-validation process is then repeated 10 times, with each of the 10 subsamples used exactly once as the validation data. The results from the folds are averaged to produce a single estimation.

The configuration file can be edited to consider the 10-fold cross-validation process. The algorithm will execute 10 times, one for each fold. Replace the dataset parameter section to the following.

```
<dataset multi="true">
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-1tra.arff</train-data>
    <test-data>data/iris/iris-10-1tst.arff</test-data>
  </dataset>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-2tra.arff</train-data>
    <test-data>data/iris/iris-10-2tst.arff</test-data>
  </dataset>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-3tra.arff</train-data>
    <test-data>data/iris/iris-10-3tst.arff</test-data>
  </dataset>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-4tra.arff</train-data>
    <test-data>data/iris/iris-10-4tst.arff</test-data>
  </dataset>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-5tra.arff</train-data>
    <test-data>data/iris/iris-10-5tst.arff</test-data>
  </dataset>
  ...
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-8tra.arff</train-data>
    <test-data>data/iris/iris-10-8tst.arff</test-data>
  </dataset>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-9tra.arff</train-data>
    <test-data>data/iris/iris-10-9tst.arff</test-data>
  </dataset>
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-10tra.arff</train-data>
    <test-data>data/iris/iris-10-10tst.arff</test-data>
  </dataset>
</dataset>
```

3.6 Parameter optimization and batch processing

The configuration file allows to create batches of experiments with equal or different parameter configurations. This is very useful for two common tasks: obtaining the optimal parameter values, and evaluating an algorithm over multiple datasets. The former focuses on evaluating the algorithm with different parameter values to analyze which performs best. For instance, the user could wonder which mutation and crossover probabilities achieve the best results. In this case, the user would edit the configuration file to the following:

```
<recombination-prob multi="true">
  <recombination-prob>0.6</recombination-prob>
  <recombination-prob>0.7</recombination-prob>
  <recombination-prob>0.8</recombination-prob>
  <recombination-prob>0.9</recombination-prob>
</recombination-prob>

<mutation-prob multi="true">
  <mutation-prob>0.05</mutation-prob>
  <mutation-prob>0.1</mutation-prob>
  <mutation-prob>0.15</mutation-prob>
  <mutation-prob>0.2</mutation-prob>
  <mutation-prob>0.3</mutation-prob>
</mutation-prob>
```

The latter focuses on executing a set of experiments over multiple datasets, which is interesting to do when we want to evaluate the overall performance of an algorithm over a wide range of data. The configuration file would be edited as in Section 3.5, changing the path of the dataset folds filenames to consider the experimentation over the different datasets.

```
<dataset multi="true">
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/iris/iris-10-1tra.arff</train-data>
    <test-data>data/iris/iris-10-1tst.arff</test-data>
  </dataset>
  ...
  <dataset type="net.sf.jclec.problem.util.dataset.ArffDataSet">
    <train-data>data/appendicitis/appendicitis-10-1tra.arff</train-data>
    <test-data>data/appendicitis/appendicitis-10-1tst.arff</test-data>
  </dataset>
</dataset>
```

3.7 Using JCLEC classification algorithms in WEKA

WEKA [7] has become one of the most popular DM (Data Mining) workbenches and its success in researcher and education communities is due to its constant improvement, development, and portability. In fact, it can be easily extended with new algorithms. This tool, developed in the Java programming language, comprises a collection of algorithms for tackling several DM tasks as data pre-processing, classification, regression, clustering, association rules, also visualizing all the DM process. However, these algorithms are hardly used in situations where there are a huge amount of instances and attributes, situations where the execution becomes computationally hard. Instead, evolutionary algorithms are very useful in this kind of circumstances, because they are powerful for solving problems that require a considerable execution time. That is why it should be very interesting to harness the power of EAs also in the DM field, and WEKA appears to be a good platform to consider the inclusion of EAs.

In [8] an intermediate layer that connects both WEKA and JCLEC is presented, showing how to include any evolutionary learning algorithm coded in JCLEC into WEKA. This enables the possibility of running this kind of algorithms in this well-known software tool as well as it provides JCLEC additional features and a graphical user interface.

To use JCLEC classification algorithms in WEKA, download the JCLEC-WEKA plugin from <http://jclec.sourceforge.net/downloads/jclec4-weka.zip> and import it into WEKA using the package manager from the Tools menu. For further information, please consult the WEKA wiki at <http://weka.wikispaces.com/How+do+I+use+the+package+manager> or the information at [8]. Note: it requires WEKA $\geq 3.7.5$.

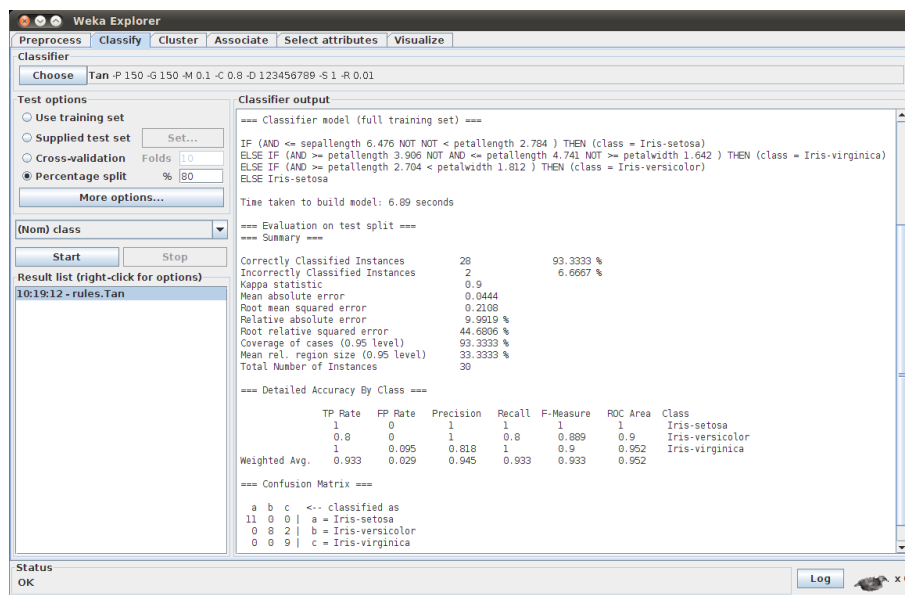


Figure 3.1: JCLEC classification algorithm running in WEKA

4. DOCUMENTATION FOR DEVELOPERS

This chapter is developer-oriented and describes the structure of the JCLEC classification module, the API reference, how to implement new algorithms and export them into WEKA, and the unit tests validation. We encourage developers to read the general JCLEC documentation, which is available at <http://jclec.sourceforge.net/data/JCLEC-tutorial.pdf>.

4.1 Package structure

Similarly to the JCLEC core, the structure of the JCLEC classification module is organized in packages (see Figure 4.1). In this section we describe the main packages in the JCLEC classification module whereas the next section presents the API reference.

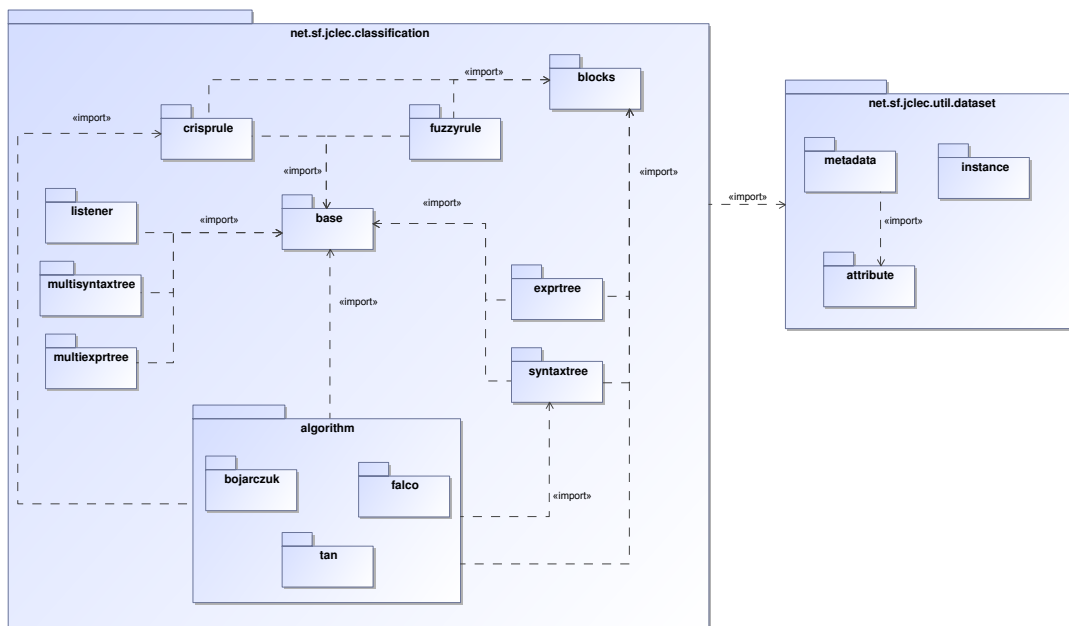


Figure 4.1: Package diagram for the JCLEC classification module

Figure 4.2 shows the class diagram for the JCLEC classification module.

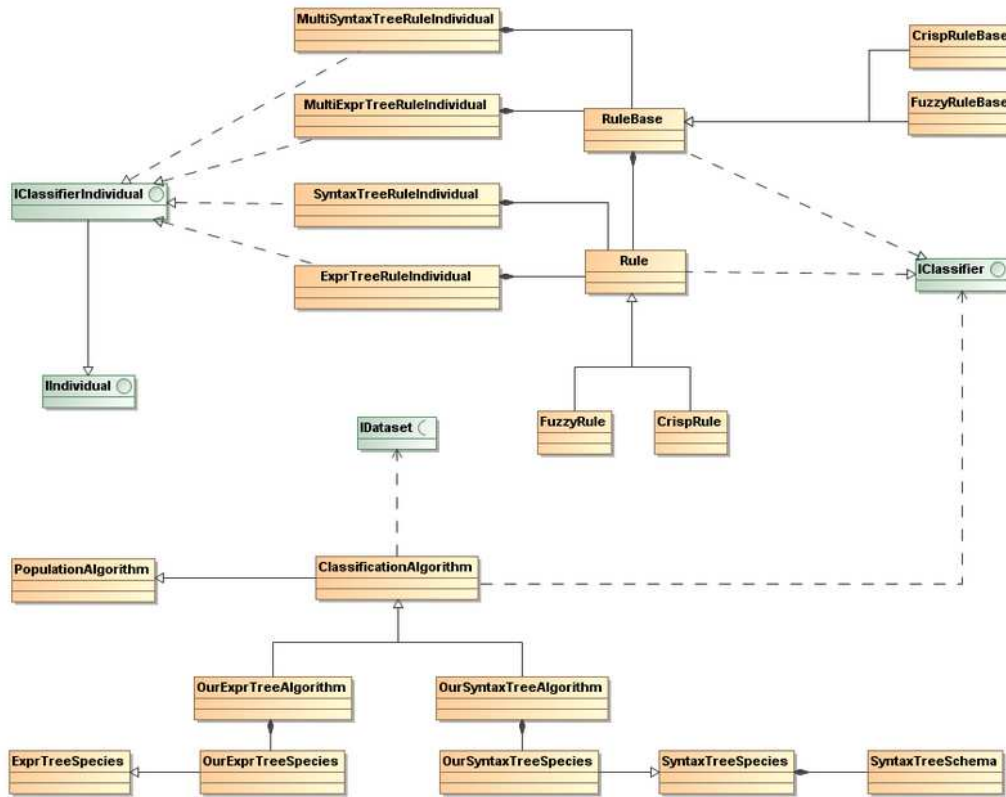


Figure 4.2: Class diagram for the JCLEC classification module

4.1.1 net.sf.jclec.problem.classification

This package contains the basic JCLEC classification module interfaces. These interfaces enable the implementation of a classifier and individuals.

Interfaces:

- **IClassifier**: generic interface for classifiers.
`double classify(IInstance instance);`
`double[] classify(IDataset dataset);`
`int[] [] getConfusionMatrix(IDataset dataset);`
- **IClassifierIndividual**: generic interface for classification individuals.
`IClassifier getPhenotype();`

4.1.2 net.fs.jclec.problem.classification.base

This base package provides the abstract classes with the properties and methods that any classification algorithm, reporter, rule or rule base must contain.

Classes:

- ClassificationAlgorithm: abstract class of a generic classification algorithm.

```
IClassifier getClassifier();
void setClassifier(IClassifier classifier);
IDataset getTrainSet();
void setTrainSet(IDataset dataset);
IDataset getTestSet();
void setTestSet(IDataset dataset);
```

- ClassificationReporter: abstract class of a classification listener.

- Rule: abstract class of a classification rule.

```
ExprTree getAntecedent();
void setAntecedent(ExprTree antecedent);
double getConsequent();
void setConsequent(double consequent);
IFitness getFitness();
void setFitness(IFitness fitness);
Object covers(IInstance instance);
int[][] getConfusionMatrix(IDataset dataset);
double[] classify(IDataset dataset);
```

- RuleBase: abstract class of a base of classification rules.

```
void setClassificationRules(List<Rule> rules);
List<Rule> getClassificationRules();
double getDefaultClass();
void setDefaultClass(double defaultClass);
void addClassificationRule(Rule rule);
void addClassificationRule(int index, Rule rule);
void setClassificationRule(int index, Rule rule);
Rule getClassificationRule(int index);
int getConditions();
IFitness getFitness();
void setFitness(IFitness fitness);
int[][] getConfusionMatrix(IDataset dataset);
double[] classify(IDataset dataset);
```

4.1.3 net.fs.jclec.problem.classification.blocks

This package contains and represents several implementations of primitive functions that could be used in an expression tree node.

Classes:

- And
- Or
- Not
- Equal

- NotEqual
- Less
- LessOrEqual
- Greater
- GreaterOrEqual
- In
- Out
- AttributeValue
- ConstantValue
- RandomConstantOfContinuousValues
- RandomConstantOfDiscreteValues

4.1.4 net.fs.jclec.problem.classification.blocks.fuzzy

This package contains and represents several implementations of fuzzy primitive functions that could be used in an expression tree node.

Classes:

- Is
- Maximum
- MembershipFunction
- Minimum
- TriangularMembershipFunction

4.1.5 net.fs.jclec.problem.classification.crisprule

This package has implementations to represent the phenotype of a crisp rule-base individual.

Classes:

- CrispRule: class that represents a crisp classification rule.

```
double classify(IInstance instance);  
double[] classify(IDataset dataset);  
int getConditions();
```
- CrispRuleBase: class that represents a crisp classification rule base.

```
double classify(IInstance instance);  
double[] classify(IDataset dataset);
```

4.1.6 net.fs.jclec.problem.classification.fuzzyrule

This package has implementations to represent the phenotype of a fuzzy rule-base individual.

Classes:

- FuzzyRule: class that represents a fuzzy classification rule.


```
double classify(IInstance instance);
double[] classify(IDataset dataset);
int getConditions();
```
- FuzzyRuleBase: class that represents a fuzzy classification rule base.


```
double classify(IInstance instance);
double[] classify(IDataset dataset);
```

4.1.7 net.fs.jclec.problem.classification.exprtree

This package defines the necessary classes to implement genetic programming encoding individuals.

Classes:

- ExprTreeSpecies: species definition for ExprTree individuals.


```
void setTerminalSymbols(List<IAttribute> inputAttributes);
void setTerminalNodes(List<IPrimitive> terminals);
void setNonTerminalNodes(List<IPrimitive> nonTerminals);
```
- ExprTreeRuleIndividual: individual specification whose genotype is a expression tree and phenotype is a classification rule.


```
ExprTree getGenotype();
Rule getPhenotype();
```

4.1.8 net.fs.jclec.problem.classification.multiexprtree

This package defines the necessary classes to implement genetic programming encoding multiple individuals.

Classes:

- MultiExprTreeRuleIndividual: individual specification whose genotype is a multi expression tree and phenotype is a rule base.


```
ExprTree[] getGenotype();
RuleBase getPhenotype();
```

4.1.9 net.fs.jclec.problem.classification.multisyntaxtree

This package defines the necessary classes to implement grammar guided genetic programming encoding multiple individuals.

Classes:

- MultiSyntaxTreeRuleIndividual: individual specification whose genotype is a multi syntax tree and phenotype is a rule base.

```
SyntaxTree[] getGenotype();
RuleBase getPhenotype();
```

4.1.10 net.fs.jclec.problem.classification.syntaxtree

This package defines the necessary classes to implement grammar guided genetic programming encoding individuals.

Classes:

- SyntaxTreeRuleIndividual: individual specification whose genotype is a syntax tree and phenotype is a classification rule.

```
SyntaxTree getGenotype();
RuleBase getPhenotype();
```
- SyntaxTreeSchema
- SyntaxTreeSpecies: species definition for SyntaxTree individuals.

```
void setGrammar();
List<TerminalNode> setTerminalSymbols(List<IAttribute> inputAttributes);
void setTerminalNodes(List<TerminalNode> terminals);
void setNonTerminalNodes(List<NonTerminalNode> nonTerminals);
```

4.1.11 net.fs.jclec.problem.classification.listener

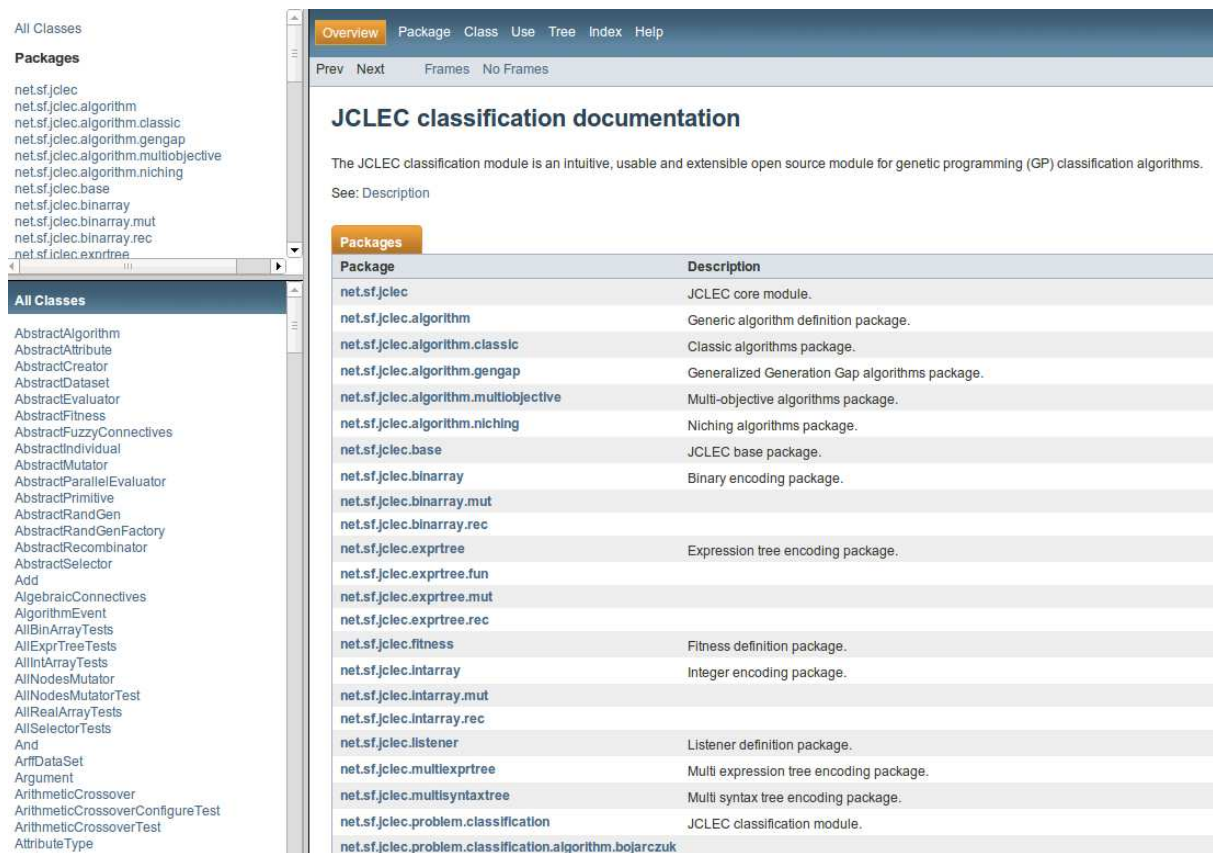
This package defines the listener to obtain reports in each generation.

Classes:

- RuleBaseReporter

4.2 API reference

API documentation includes information about the code, the parameters and the content of the functions. For further information, please visit the API reference which is available online at the website <http://jclec.sourceforge.net/data/jclec4-classification-doc/>.



The screenshot displays the JCLEC API reference website. On the left, a sidebar contains two sections: 'All Classes' and 'Packages'. The 'All Classes' section lists various abstract and concrete classes, including `AbstractAlgorithm`, `AbstractAttribute`, `AbstractCreator`, `AbstractDataset`, `AbstractEvaluator`, `AbstractFitness`, `AbstractFuzzyConnectives`, `AbstractIndividual`, `AbstractMutator`, `AbstractParallelEvaluator`, `AbstractPrimitive`, `AbstractRandGen`, `AbstractRandGenFactory`, `AbstractRecombinator`, `AbstractSelector`, `Add`, `AlgebraicConnectives`, `AlgorithmEvent`, `AllBinArrayTests`, `AllExprTreeTests`, `AllIntArrayTests`, `AllNodesMutator`, `AllNodesMutatorTest`, `AllRealArrayTests`, `AllSelectorTests`, `And`, `ArrfDataSet`, `Argument`, `ArithmeticCrossover`, `ArithmeticCrossoverConfigureTest`, `ArithmeticCrossoverTest`, and `AttributeType`. The 'Packages' section lists various packages, including `net.sf.jclec`, `net.sf.jclec.algorithm`, `net.sf.jclec.algorithm.classic`, `net.sf.jclec.algorithm.gengap`, `net.sf.jclec.algorithm.multiobjective`, `net.sf.jclec.algorithm.niching`, `net.sf.jclec.base`, `net.sf.jclec.binary`, `net.sf.jclec.binary.mut`, `net.sf.jclec.binary.rec`, and `net.sf.jclec.exptree`.

The main content area is titled 'JCLEC classification documentation'. It includes a description of the JCLEC classification module as an intuitive, usable, and extensible open source module for genetic programming (GP) classification algorithms. Below the description, there is a 'Packages' table with two columns: 'Package' and 'Description'.

Package	Description
<code>net.sf.jclec</code>	JCLEC core module.
<code>net.sf.jclec.algorithm</code>	Generic algorithm definition package.
<code>net.sf.jclec.algorithm.classic</code>	Classic algorithms package.
<code>net.sf.jclec.algorithm.gengap</code>	Generalized Generation Gap algorithms package.
<code>net.sf.jclec.algorithm.multiobjective</code>	Multi-objective algorithms package.
<code>net.sf.jclec.algorithm.niching</code>	Niching algorithms package.
<code>net.sf.jclec.base</code>	JCLEC base package.
<code>net.sf.jclec.binary</code>	Binary encoding package.
<code>net.sf.jclec.binary.mut</code>	
<code>net.sf.jclec.binary.rec</code>	
<code>net.sf.jclec.exptree</code>	Expression tree encoding package.
<code>net.sf.jclec.exptree.fun</code>	
<code>net.sf.jclec.exptree.mut</code>	
<code>net.sf.jclec.exptree.rec</code>	
<code>net.sf.jclec.fitness</code>	Fitness definition package.
<code>net.sf.jclec.intarray</code>	Integer encoding package.
<code>net.sf.jclec.intarray.mut</code>	
<code>net.sf.jclec.intarray.rec</code>	
<code>net.sf.jclec.listener</code>	Listener definition package.
<code>net.sf.jclec.multiexptree</code>	Multi expression tree encoding package.
<code>net.sf.jclec.multisyntaxtree</code>	Multi syntax tree encoding package.
<code>net.sf.jclec.problem.classification</code>	JCLEC classification module.
<code>net.sf.jclec.problem.classification.algorithm.bojarczuk</code>	

Figure 4.3: API reference website

4.3 Implementing new algorithms

Implementing new and personalized classification algorithms into the software is an easy task. A developer only has to create three Java classes using the same structure as other algorithms. These classes are the algorithm, the evaluator and the species.

4.3.1 Algorithm

A new algorithm included in the module should inherit from the abstract `ClassificationAlgorithm` class which contains the common properties and methods to all classification algorithms (individual's provider, evaluator, species, classifier, etc.). On the other hand, the algorithm class comprises all the particular properties and methods of the algorithm (individual's parent selector, recombinator, mutator, etc.). Each of these properties are respectively defined and configured by means of the configuration file, which specifies the classes and the attribute values. Following, the main characteristics of the Bojarczuk algorithm are described to illustrate an algorithm's code and functionality.

The Bojarczuk algorithm inherits the common properties from the abstract `ClassificationAlgorithm` class: the individual's provider, the evaluator and the species. The user must specify the class' names for these properties in the configuration file. Moreover, other common properties used by classification algorithms are inherited, such as the classifier declaration, the train and test dataset specification, and

other parameters such as the population size, and the number of generations. On the other hand, it also adds three other properties of the particular algorithm: the parent selector, the recombinator and the copy probability responsible for the reproduction of the individuals.

```

////////////////////////////////////
// ----- Inherited Properties
////////////////////////////////////

/** Individual species */
protected ISpecies species;

/** Individuals evaluator */
protected IEvaluator evaluator;

/** Individuals provider */
protected IProvider provider;

/** Classifier */
protected IClassifier classifier;

/** Train Dataset */
protected IDataset trainSet;

/** Test Dataset */
protected IDataset testSet;

////////////////////////////////////
// ----- Algorithm Properties
////////////////////////////////////

/** Parents selector */
protected ISelector parentsSelector;

/** Individuals recombinator */
protected FilteredRecombinator recombinator;

/** Copy probability */
private double copyProb;

```

Next, each property must be accompanied by its respective get and set method.

```

////////////////////////////////////
// ----- Getting and setting properties
////////////////////////////////////

/**
 * Access the parent selection method
 *
 * @return parentsSelector
 */
public ISelector getParentsSelector()
{
    return parentsSelector;
}

```

```
/**
 * Establishes the parent selection method
 *
 * @param parentsSelector the parent selector
 */
public void setParentsSelector(ISelector parentsSelector)
{
    // Set parents selector
    this.parentsSelector = parentsSelector;
    // Contextualize selector
    parentsSelector.contextualize(this);
}

/**
 * Access to parents recombinator
 *
 * @return Actual parents recombinator
 */
public FilteredRecombinator getRecombinator()
{
    return recombinator;
}

/**
 * Sets the parents recombinator.
 *
 * @param recombinator the parents recombinator
 */
public void setRecombinator(IRecombinator recombinator)
{
    if (this.recombinator == null)
        this.recombinator = new FilteredRecombinator (this);

    this.recombinator.setDecorated(recombinator);
}

/**
 * Access to "copyProb" property.
 *
 * @return Current copy probability
 */
public double getCopyProb()
{
    return copyProb;
}

/**
 * Set the "copyProb" property.
 *
 * @param copyProb the copy probability
 */
public void setCopyProb(double copyProb)
{
    this.copyProb = copyProb;
}
```

The configure method is responsible for analyzing the configuration file, instantiating the operators specified and setting up their respective configurations. All configuration parameters should use a default value if the user does not specify. This configuration method particularly enforces the species, evaluator, individuals provider, parents selector and individuals recombinator since they are fixed by the authors of the Bojarczuk algorithm in their paper description. However, the flexibility of JCLEC allows of removing those fixed operators from the configure method, allowing to the user to specify these parameters in the configuration file dynamically. Therefore, it easily allows the use of the algorithm with different evaluators, parent selectors, recombinators or mutators.

```
public void configure(Configuration settings)
{
    // Fixed species, evaluator, provider, parents-selector and recombinator
    settings.addProperty("species[@type]",
        "net.sf.jclec.problem.classification.syntaxtree.bojarczuk.BojarczukSyntaxTreeSpecies");
    settings.addProperty("evaluator[@type]",
        "net.sf.jclec.problem.classification.syntaxtree.bojarczuk.BojarczukEvaluator");
    settings.addProperty("provider[@type]",
        "net.sf.jclec.syntaxtree.SyntaxTreeCreator");
    settings.addProperty("parents-selector[@type]",
        "net.sf.jclec.selector.TournamentSelector");
    settings.addProperty("recombinator[@type]",
        "net.sf.jclec.syntaxtree.SyntaxTreeRecombinator");
    settings.addProperty("recombinator[@rec-prob]",
        settings.getDouble("recombination-prob", 0.8));
    settings.addProperty("recombinator.base-op[@type]",
        "net.sf.jclec.syntaxtree.rec.SelectiveCrossover");

    // Call super.configure() method
    super.configure(settings);

    // Create CrispRuleBase classifier
    classifier = new CrispRuleBase();

    // Establishes the dataset metadata for the species
    ((BojarczukSyntaxTreeSpecies) species).setMetadata(getTrainSet().getMetadata());

    // Get maximum tree depth for individuals creation
    int maxDerivSize = settings.getInt("max-deriv-size", 20);
    ((BojarczukSyntaxTreeSpecies) species).setGrammar();
    ((BojarczukSyntaxTreeSpecies) species).setMaxDerivSize(maxDerivSize);

    // Establishes the training set for evaluating
    ((BojarczukEvaluator) evaluator).setDataset(getTrainSet());
    ((BojarczukEvaluator) evaluator).setMaxDerivSize(maxDerivSize);

    // Configure parents selector
    setParentsSelectorSetting(settings);

    // Configure recombinator
    setRecombinatorSetting(settings);

    // Set copy probability
    double copyProb = settings.getDouble("copy-prob", 0.1);
    setCopyProb(copyProb);
}
```


Finally, the iterative process of the Bojarczuk algorithm is defined in the following methods which are executed sequentially and repeatedly following the classic process of an evolutionary algorithm. Firstly, the initialization method is executed. It is responsible for the initial creation of the population using the individuals provider and the population size specified in the configuration file. The individuals of the current population are referenced as *bset*.

```
protected void doInit()
{
    // Create individuals
    bset = provider.provide(populationSize);
    // Evaluate individuals
    evaluator.evaluate(bset);
}
```

The selection methods performs the parent selection among the individuals of the current generation and references the parents as *pset*. The Bojarczuk algorithm also considers a pool of elitist individuals which are referenced as *eset* and they are considered for the parents selection.

```
protected void doSelection()
{
    pset = parentsSelector.select(bset, populationSize);
    pset.addAll(eset);
}
```

The generation method applies the genetic operators to create the offspring and evaluate the new individuals which are referenced as *cset*. It should be noted that those sterile individuals are included in the population *cset*. The Bojarczuk algorithm only applies the recombination operator.

```
protected void doGeneration()
{
    // Recombine parents
    cset = recombinator.recombine(pset);
    cset.addAll(recombinator.getSterile());

    // Evaluate all new individuals
    evaluator.evaluate(cset);
}
```

Any evolutionary algorithm in JCLEC includes the *doUpdate()* method. This method is responsible for update the populations of solutions. Thus, this method could vary from algorithm to algorithm.

Focusing on the update stage of the Bojarczuk algorithm, it copies individuals based on a copy probability, and those individuals that satisfy the probability are kept in the *cset* population, that is, the current population. In a subsequent step, the *doUpdate()* method select the best individuals from this current population and replace the general population *bset*. The best individuals for each data class are ensured to survive as elitists individuals, whose rules are used to build the classifier.

```

protected void doUpdate()
{
    // Do the reproduction operator
    for(IIndividual individual : bset)
    {
        if(randgen.coin(copyProb))
            cset.add(individual.copy());
    }

    List<Rule> classificationRule = new ArrayList<Rule>();
    List<Double> classes = new ArrayList<Double>();

    // We leave space for the best rules per class that existed in the previous population
    if(cset.size() + getTrainSet().getMetadata().numberOfClasses() > populationSize)
        bset = bettersSelector.select(cset, numberOfClasses);
    else
        bset = bettersSelector.select(cset);

    bset.addAll(eset);

    eset = new ArrayList<IIndividual>();

    //Select the best individual for each class
    for(int i=0; i < bset.size() && classes.size() != numberOfClasses; i++)
    {
        Rule rule = (Rule) ((SyntaxTreeRuleIndividual) bset.get(i)).getPhenotype();
        rule.setFitness(bset.get(i).getFitness());

        if(!classes.contains(rule.getConsequent()))
        {
            boolean added = false;

            for(int j = 0; j < classificationRule.size(); j++)
            {
                if(getEvaluator().getComparator().compare(classificationRule.get(j).getFitness(),
                                                            rule.getFitness()) <= 0)
                {
                    classificationRule.add(j,rule);
                    added = true;
                    break;
                }
            }

            if(!added)
                classificationRule.add(rule);

            classes.add(rule.getConsequent());

            eset.add(bset.get(i).copy());
        }
    }

    ((CrispRuleBase) classifier).setClassificationRules(classificationRule);

    cset = pset = rset = null;
}

```

Finally, evolution is finished if the number of generations or evaluations performed achieve the maximum allowed. Then, the classifier is submitted to the train and test data to evaluate its quality and obtain the performance measures.

```
protected void doControl()
{
    // If maximum number of generations or evaluations is exceeded, the algorithm is finished
    if (generation >= maxOfGenerations || evaluator.getNumberOfEvaluations() > maxOfEvaluations)
    {
        state = FINISHED;
        return;
    }
}
```

4.3.2 Evaluator

The evaluator class inherits from the `AbstractEvaluator` class, enabling the evaluation of each individual in the population from the evolutionary algorithm. The evaluation of the individuals from the population can be performed in parallel by extending from the `AbstractParallelEvaluator` class. It will automatically create as many threads as number of CPU cores to evaluate each individual independently, improving the runtime of the algorithm in multicore environments.

Next, the evaluator for the Bojarczuk algorithm is described to depict how the rules are evaluated over the instances of the dataset. Firstly, the phenotype of the individual is obtained, which represents the classification rule that comprises the antecedent and the consequent (predicted class). The Bojarczuk evaluator attempts to obtain the best possible predicted class for the antecedent of the rule. Therefore, it evaluates the sensitivity and specificity for each data class. To do so, it creates the confusion matrix for each class and checks whether the rule covers or not the instances of the dataset. Then, it assigns the consequent of the data class which obtained the maximum product of sensitivity and specificity. Finally, it computes the fitness of the individual based on the product of the sensitivity, specificity and simplicity of the rule. The simplicity is calculated regarding the length of the rule and the maximum number of derivations allowed by the grammar. The shorter and more sensitivity and specificity of the rule, the better fitness.

```
protected void evaluate(IIndividual individual)
{
    Rule rule = (Rule) ((SyntaxTreeRuleIndividual) individual).getPhenotype();

    int[] tp = new int [numClasses];    int[] tn = new int [numClasses];
    int[] fn = new int [numClasses];    int[] fp = new int [numClasses];

    //Calculate the confusion matrix for each class
    for(IInstance instance : dataset.getInstancees())
    {
        if((Boolean) rule.covers(instance)) {
            double value = instance.getValue(metadata.getClassIndex());
            tp[(int) value]++;
            for(int i=0; i<numClasses; i++)
                if(((int) value) != i)
                    fp[i]++;
        }
        else {
```

```

        double value = instance.getValue(metadata.getClassIndex());

        fn[(int) value]++;
        for(int i=0; i<numClasses; i++)
            if(((int) value) != i)
                tn[i]++;
    }
}

//Calculate the fitness for each class
double se = -1, sp = 1, sy, seAux, spAux;
int bestClass = -1;

for(int i=0; i<numClasses; i++)
{
    if(tp[i]+fn[i] == 0)        seAux = 1;
    else                        seAux = (double) tp[i]/(tp[i]+fn[i]);

    if(tn[i]+fp[i] == 0)        spAux = 1;
    else                        spAux = (double) tn[i]/(tn[i]+fp[i]);

    if(seAux*spAux == se*sp) bestClass = i;

    if(seAux*spAux > se*sp) {
        se = seAux;
        sp = spAux;
        bestClass = i;
    }
}

// Assign as consequent the class that reports the best fitness
rule.setConsequent(bestClass);
sy = (getMaxDerivSize() - 0.5*rule.getAntecedent().size() -0.5)/(getMaxDerivSize()-1);
individual.setFitness(new SimpleValueFitness(se*sp*sy));
}

```

Another commonly used evaluator for classification rules when the consequent is fixed, achieves a tradeoff between sensitivity and specificity.

```

protected void evaluate(IIndividual individual)
{
    Rule rule = (Rule) ((SyntaxTreeRuleIndividual) individual).getPhenotype();

    int tp = 0, fp = 0, tn = 0, fn = 0;

    //Calculate the confusion matrix
    for(IExample example : dataset.getExamples())
    {
        double Class = ((ClassicInstance) example).getClassValue();

        if((Boolean) rule.covers(example)) {
            if (Class == classifiedClass)
                tp++;
            else
                fp++;
        }
    }
}

```

```

        else {
            if (Class != classifiedClass)
                tn++;
            else
                fn++;
        }
    }

    // Sensitivity, specificity, fitness
    double se, sp, fitness;

    if(tp + fn == 0)
        se = 1;
    else
        se = (double) tp / (tp + w1*fn);

    if(tn + fp == 0)
        sp = 1;
    else
        sp = (double) tn / (tn + w2*fp);

    // Set the fitness to the individual
    fitness = se * sp;

    individual.setFitness(new SimpleValueFitness(fitness));
}

```

4.3.3 Species

This class represents the individual species to be used in the evolutionary algorithm. Here, the user could make a differentiation between expression-tree and syntax-tree respectively. In such a way, each GP classification individual is represented by means of the `ExprTreeRuleIndividual` class, which represents an individual, comprising all the features required: the genotype, the phenotype, and the fitness value. The nodes and functions in GP trees are defined by the `ExprTreeSpecies` class. Similarly to GP individuals, the `SyntaxTreeRuleIndividual` class specifies all the features required to represent a G3P individual, while the `SyntaxTreeSpecies` allows of defining the terminal and nonterminal symbols of the grammar used to generate the individuals.

The Bojarczuk algorithm specifies a `SyntaxTreeSpecies` class which defines the terminals and non-terminals symbols, and the productions rules of the context-free grammar. The terminal symbols are `=`, `≠`, `>`, `≤` operators, and `AND` and `OR` functions. Here, the user could easily add or remove the operators desired.

```

protected List<TerminalNode> setTerminalSymbols(List<IAttribute> inputAttributes)
{
    List<TerminalNode> terminals = super.setTerminalSymbols(inputAttributes);

    //Set fixed terminal symbol
    if(existCategoricalAttributes){
        terminals.add(new TerminalNode("=", new Equal()));
        terminals.add(new TerminalNode("!=" , new NotEqual()));
    }
    if(existNumericalAttributes){

```

```

        terminals.add(new TerminalNode(">", new Greater()));
        terminals.add(new TerminalNode("<=", new LessOrEqual()));
    }

    //Set fixed boolean operators
    terminals.add(new TerminalNode("AND", new And()));
    terminals.add(new TerminalNode("OR", new Or()));

    return terminals;
}

```

On the other hand, the non-terminal symbols method defines the production rules of the grammar. Beginning with *antecedent*, the rules are derivated until the terminal symbols are reached. Adding or removing production rules and changing the grammar can be easily done by modifying these lines.

```

protected List<NonTerminalNode> setNonTerminalSymbols(List<IAttribute> inputAttributes)
{
    List<NonTerminalNode> nonTerminals = new ArrayList<NonTerminalNode>();

    // Number of input attribute
    int numAttributes = inputAttributes.size();

    //Set fixed non terminal symbols
    nonTerminals.add(new NonTerminalNode("antecedent", new String []
        {"conjunction"}));
    nonTerminals.add(new NonTerminalNode("antecedent", new String []
        {"OR", "conjunction", "antecedent"}));
    nonTerminals.add(new NonTerminalNode("conjunction", new String []
        {"comparison"}));
    nonTerminals.add(new NonTerminalNode("conjunction", new String []
        {"AND", "comparison", "conjunction"}));

    if(existCategoricalAttributes) {
        nonTerminals.add(new NonTerminalNode("comparison", new String []
            {"categorical_comparator", "categorical_attribute_comparison"}));
        nonTerminals.add(new NonTerminalNode("categorical_comparator", new String []
            {"="}));
        nonTerminals.add(new NonTerminalNode("categorical_comparator", new String []
            {"!="}));
    }

    if(existNumericalAttributes) {
        nonTerminals.add(new NonTerminalNode("comparison", new String []
            {"numerical_comparator", "numerical_attribute_comparison"}));
        nonTerminals.add(new NonTerminalNode("numerical_comparator", new String []
            {">"}));
        nonTerminals.add(new NonTerminalNode("numerical_comparator", new String []
            {"<="}));
    }

    // Set non terminal symbols
    for(int i =0 ; i<numAttributes; i++){
        IAttribute attribute = inputAttributes.get(i);
        String [] attr = new String [] {attribute.getName(), "values"+attribute.getName()};

        switch(attribute.getType()) {
            case Numerical:

```

```

        nonTerminals.add(new NonTerminalNode("numerical_attribute_comparison", attr));
        break;
    case Integer:
        nonTerminals.add(new NonTerminalNode("numerical_attribute_comparison", attr));
        break;
    case Categorical:
        nonTerminals.add(new NonTerminalNode("categorical_attribute_comparison", attr));
        break;
    }
}
return nonTerminals;
}

```

4.4 Exporting new algorithms into WEKA

WEKA's design allows to include new algorithms easily. Any new class is picked up by the graphical user interface without additional coding needed to deploy it in WEKA. To do so, new algorithms should inherit some properties from certain classes, which also indicate the methods that should be implemented. New classification algorithms should extend the **AbstractClassifier** class. An abstract class called **ClassificationAlgorithm** that collects the common properties of classification algorithms has been developed for JCLEC, which extends from **AbstractAlgorithm**. Thus, any classification algorithm will inherit from **ClassificationAlgorithm** the common properties and methods, and it will just specify its particular properties.

The architectural design, developed to include JCLEC in WEKA, follows the schema represented in Figure 4.4, where it is depicted that WEKA provides an abstract class from which any classification algorithm should inherit **AbstractClassifier**. The abstract class **ClassificationAlgorithm** extends **AbstractClassifier**, and it is in charge of defining the properties and methods that any evolutionary classification algorithm shares, e.g., population size, number of generations, crossover and mutation operators, etc. Finally, any particular algorithm extends from this class and calls the corresponding execution method of JCLEC. This way, the JCLEC classification algorithm is executed in WEKA.

Next, the methods of the intermediate layer that have to be taken into account when including any classification algorithm are described. The following methods should be implemented:

- **void buildClassifier(Instances data)**. This method generates a classifier from a training set.
- **double classifyInstance(Instances instance)**. This method classifies a new instance using the classifier learned previously.
- **Capabilities getCapabilities()**. This method determines if the algorithm is compatible with the type of each attribute in the dataset (e.g., numeric, nominal, etc.).
- **String globalInfo()**. Returns information about the algorithm, which will appear when selecting the *About* option in the graphical user interface.
- **TechnicalInformation getTechnicalInformation()**. Shows information about the author, year of the publication, etc.
- **void setOptions(String[] options)**. This method establishes the parameters of the algorithm, e.g., -P 50 -G 100—the former indicates the population size and the latter the number of generations.
- **String [] getOptions()**. Returns the set of parameters previously established.

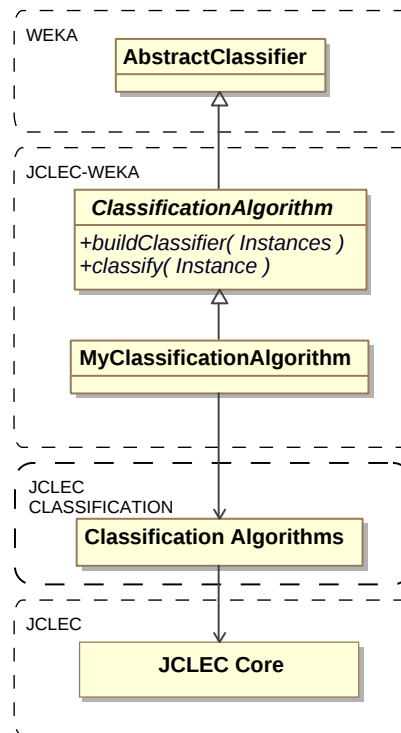


Figure 4.4: Architectonical design JCLEC-WEKA

- **Setters** and **getters** methods that set and get the parameter values.
- **String toString()**. This method shows the results obtained in the graphical user interface.

An additional and very useful tool contained in WEKA is the package manager. This tool allows the inclusion of any external library or code necessary to run new algorithms or features in WEKA, incorporating the files into the properly structure so that it avoids the developer to modify WEKA's source code. The directory structure of any new package has to fulfill a fixed anatomy. Hence, the JCLEC-WEKA intermediate layer is necessary in order to be able to instantiate the execution code of the algorithm, as shown in Figure 4.4. This way, a connection between WEKA and JCLEC is established. To use JCLEC classification algorithms in WEKA, download the JCLEC-WEKA plugin from <http://jclec.sourceforge.net/downloads/jclec4-weka.zip> and import it into Eclipse and WEKA using the package manager from the Tools menu. Further information about how to organize the structure of the packages can be consulted in the WEKA wiki at <http://weka.wikispaces.com/How+are+packages+structured+for+the+package+management+system%3F>.

In this section you will find the results of running unit tests over the algorithms available in the JCLeC classification package. A unit test is a piece of code written by a developer that tests a specific functionality in the code which is tested. Unit tests can ensure that functionality is working and can be used to validate that this functionality still works after code changes. For the sake of running unit tests, JUnit framework was used.

-
- The screenshot shows the IntelliJ IDEA IDE interface. The main editor displays the `FalcoAlgorithmTest.java` file, which contains a JUnit test for the `FalcoAlgorithm` class. The test method `testFalcoAlgorithm()` creates a `FalcoAlgorithm` instance, configures it with an XML configuration file, and then calls `execute()`. The IDE shows that the test passed successfully.
- On the left, the "Package Explorer" shows the project structure, including the `net.sf.jcle.problem.classification.syntaxtree.falco` package. Below it, the "Failure Trace" panel is empty, indicating no failures.
- At the bottom, the "Console" panel shows the output of the test execution, confirming that the test passed and the IDE version is 18.0.0 (2011.17.45.23).
- ```

1 package net.sf.jcle.problem.classification.syntaxtree.falco;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
6
7 @RunWith(SpringJUnit4ClassRunner.class)
8 public class FalcoAlgorithmTest {
9
10 @Test
11 public void testFalcoAlgorithm() {
12 FalcoAlgorithm falco = new FalcoAlgorithm();
13
14 // Job configuration
15 XMLConfiguration jobConf;
16 try {
17 jobConf = new XMLConfiguration("./examples/falco.cfg");
18 String header = "process(0)";
19 falco = new FalcoAlgorithm();
20 ((IConfigure) falco).configure(jobConf.subset(header));
21
22 } catch (ConfigurationException e) {
23 e.printStackTrace();
24 }
25
26 falco.execute();
27 }
28 }

```
- Console Output:
- ```

<terminated> FalcoAlgorithmTest [JUnit] C:\Program Files\Java\jdk6\bin\javaw.exe (18/10/2011 17:45:23)

```

Figure 4.5: Falco unit test

- Tan unit test

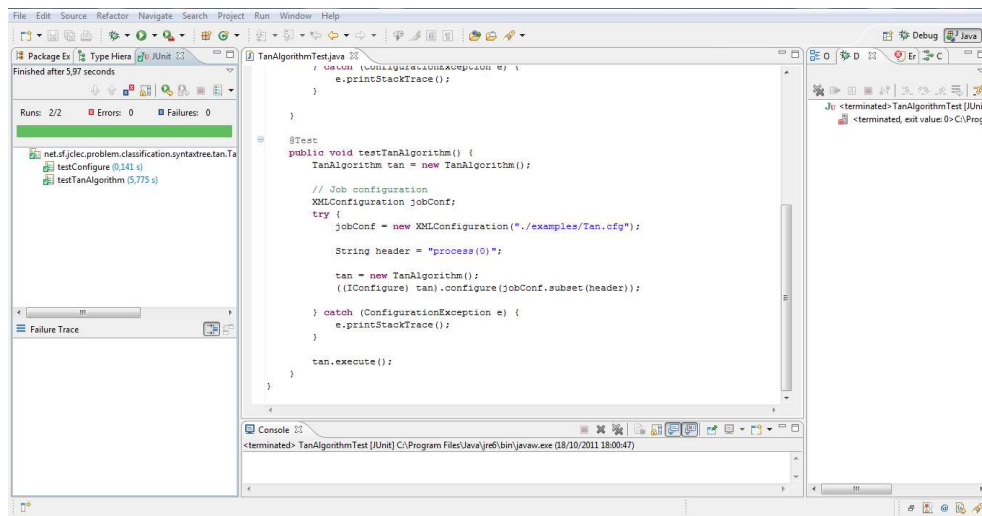


Figure 4.6: Tan unit test

- Bojarczuk unit test

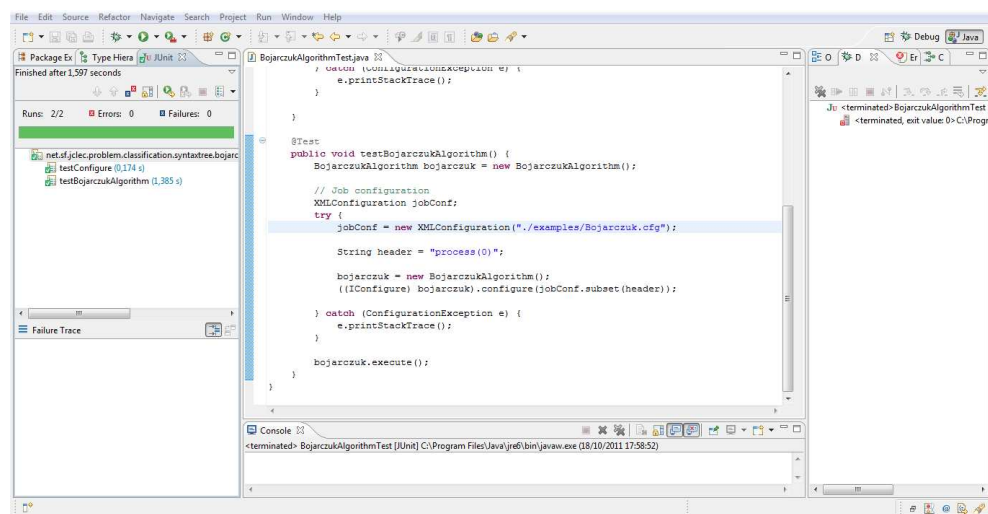


Figure 4.7: Bojarczuk unit test

5. CITATION

When referring to the software, please include the following citation:

```
@ARTICLE{Cano:JMLR:2014,  
  author={Cano, A. and Luna, J. M. and Zafra, A. and Ventura, S.},  
  title={A Classification Module for Genetic Programming Algorithms in JCLEC},  
  journal={Journal of Machine Learning Research},  
  volume={--},  
  pages={--},  
  year={2014},  
  note={\url{http://jclec.sourceforge.net/classification}}  
}
```


BIBLIOGRAPHY

- [1] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, “JCLEC: A Java Framework for Evolutionary Computation,” *Soft Computing*, vol. 12, pp. 381–392, 2007.
- [2] P. G. Espejo, S. Ventura, and F. Herrera, “A Survey on the Application of Genetic Programming to Classification,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 40, no. 2, pp. 121–144, 2010.
- [3] R. McKay, N. Hoai, P. Whigham, Y. Shan, and M. O’Neill, “Grammar-based Genetic Programming: a Survey,” *Genetic Programming and Evolvable Machines*, vol. 11, pp. 365–396, 2010.
- [4] I. De Falco, A. Della Cioppa, and E. Tarantino, “Discovering interesting classification rules with genetic programming,” *Applied Soft Computing*, vol. 1, no. 4, pp. 257–269, 2001.
- [5] K. C. Tan, A. Tay, T. H. Lee, and C. M. Heng, “Mining multiple comprehensible classification rules using genetic programming,” in *Proceedings of the Evolutionary Computation on 2002. CEC ’02*, vol. 2, pp. 1302–1307, 2002.
- [6] C. C. Bojarczuk, H. S. Lopes, A. A. Freitas, and E. L. Michalkiewicz, “A constrained-syntax genetic programming system for discovering classification rules: application to medical data sets,” *Artificial Intelligence in Medicine*, vol. 30, no. 1, pp. 27–48, 2004.
- [7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemannr, and I. H. Witten, “The WEKA Data Mining Software: An Update,” *SIGKDD Explorations*, vol. 11, pp. 10–18, 2009.
- [8] A. Cano, J. M. Luna, J. L. Olmo, and S. Ventura, “JCLEC Meets WEKA!,” *Lecture Notes on Computer Science*, vol. 6678, pp. 388–395, 2011.