**CSC3423 – Biocomputing**

**Practical2: Genetic Programming**

The aim of this practical is to familiarise yourself with the working of the genetic programming algorithm that you have been provided for coursework 2.

Specifically the objectives of this practical are:

1. Learning how to run the GP code for machine learning
2. Understanding the different parts of the provided code and the epochx open source library
3. Starting to think about how to generate and extract the performance descriptors for the experiments with the GP
4. Starting to think about how to optimise the GP for the provided task

**Getting and running the code.**

The code for the coursework will be in Blackboard (Learning Materials > Assessment > Coursework 2 code).

Before we can run the code, Control.java has to be edited to select the GP version of the code:

public static Classifier generateSubsolution(InstanceSet trainingSet) {

//GAwrapper wrapper = new GAwrapper(trainingSet);

//return wrapper.generateClassifier();

**GPwrapper wrapper = new GPwrapper(trainingSet);**

**return wrapper.generateClassifier();**

//PSOwrapper wrapper = new PSOwrapper(trainingSet);

//return wrapper.generateClassifier();

//return new ClassifierMLP(trainingSet);

}

The following panel shows how to call (and the output it produces) the GP part of the code from the linux command line (the $ just indicates the prompt, don’t type it).

If you run this code from e.g. Eclipse you will need to tell the IDE to load the jar files associated to the EphochX (<https://www.epochx.org/>) GP open source library (epochx-1.4.1.jar and commons-lang-2.5.jar).

The program receives two command line argument which are the names of the training and test files of the provided machine learning dataset. Please note that the code (like any GP) depends on pseudo-random numbers, and hence the output it generates may be different when you run it.

$ java -cp epochx-1.4.1.jar:commons-lang-2.5.jar:. Control training.arff test.arff

Relation name TAO\_grid

Attribute name x

Attribute name y

Attribute name class

0 311.27 ADD(PDIV(MUL(SUB(MUL(MUL(var0 var0) MUL(var0 var0)) ADD(SUB(var1 var0) PDIV(var1 var1))) SUB(ADD(ADD(var0 var0) ADD(var0 var1)) PDIV(ADD(var1 var0) MUL(var0 var1)))) PDIV(MUL(PDIV(PDIV(var0 var1) ADD(var1 var1)) ADD(PDIV(var1 var0) MUL(var0 var0))) MUL(PDIV(MUL(var0 var0) SUB(var1 var1)) PDIV(MUL(var0 var0) MUL(var0 var0))))) ADD(SUB(ADD(MUL(PDIV(var1 var1) ADD(var0 var1)) SUB(ADD(var1 var1) PDIV(var1 var0))) PDIV(PDIV(SUB(var1 var0) PDIV(var0 var0)) ADD(MUL(var0 var1) PDIV(var1 var0)))) ADD(SUB(MUL(PDIV(var0 var0) MUL(var1 var0)) SUB(SUB(var0 var0) ADD(var0 var1))) PDIV(PDIV(ADD(var1 var0) ADD(var1 var1)) MUL(PDIV(var0 var0) ADD(var0 var0))))))

1 309.27 ADD(PDIV(MUL(SUB(MUL(MUL(var0 var0) MUL(var0 var0)) ADD(SUB(var1 var0) PDIV(var1 var1))) SUB(ADD(ADD(var0 var0) ADD(var0 var1)) PDIV(ADD(var1 var0) MUL(var0 var1)))) PDIV(MUL(PDIV(PDIV(var0 var1) ADD(var1 var1)) ADD(PDIV(var1 var0) MUL(var0 var0))) MUL(PDIV(MUL(var0 var0) SUB(var1 var1)) PDIV(MUL(var0 var0) MUL(var0 var0))))) ADD(SUB(ADD(MUL(PDIV(var1 var1) ADD(var0 var1)) SUB(ADD(var1 var1) PDIV(var1 var0))) PDIV(PDIV(SUB(var1 var0) PDIV(var0 var0)) ADD(MUL(var0 var1) PDIV(var1 var0)))) ADD(SUB(MUL(PDIV(var0 var0) MUL(var1 var0)) SUB(SUB(var0 var0) ADD(var0 var1))) PDIV(PDIV(PDIV(var0 var1) ADD(var1 var1)) MUL(PDIV(var0 var0) ADD(var0 var0))))))

2 270.65 SUB(var1 SUB(PDIV(ADD(PDIV(PDIV(var0 var1) SUB(var0 var0)) PDIV(ADD(var0 var0) MUL(var0 var0))) MUL(ADD(ADD(var0 var1) SUB(var1 var0)) MUL(MUL(var0 var1) PDIV(var1 var1)))) ADD(PDIV(ADD(MUL(var0 var0) MUL(var0 var0)) SUB(ADD(var0 var1) SUB(var0 var0))) MUL(MUL(SUB(var0 var0) MUL(var1 var0)) ADD(ADD(var1 var1) MUL(var1 var0))))))

.

.

.

50 197.05 SUB(var1 SUB(SUB(PDIV(PDIV(var1 var1) MUL(PDIV(SUB(SUB(var0 var0) var0) var1) var1)) ADD(PDIV(ADD(MUL(var0 var0) var0) ADD(var0 var1)) SUB(ADD(SUB(SUB(ADD(SUB(ADD(var0 var1) SUB(var0 SUB(ADD(ADD(var0 var1) var1) SUB(var0 var0)))) var1) var0) var0) var1) SUB(PDIV(ADD(ADD(PDIV(ADD(MUL(SUB(ADD(var0 var1) SUB(var0 var0)) var0) MUL(var0 PDIV(var1 var1))) ADD(var0 var1)) SUB(var0 var0)) MUL(var0 PDIV(var1 var1))) SUB(ADD(var0 var1) SUB(var0 var0))) var0)))) ADD(PDIV(ADD(MUL(PDIV(var1 var1) var0) MUL(var0 var0)) ADD(var0 var1)) var1)))

Best program:

SUB(var1 SUB(SUB(PDIV(PDIV(var1 var1) MUL(PDIV(SUB(SUB(var0 var0) var0) var1) var1)) ADD(PDIV(ADD(MUL(var0 var0) var0) ADD(var0 var1)) SUB(ADD(SUB(SUB(ADD(SUB(ADD(var0 var1) SUB(var0 SUB(ADD(ADD(var0 var1) var1) SUB(var0 var0)))) var1) var0) var0) var1) SUB(PDIV(ADD(ADD(PDIV(ADD(MUL(SUB(ADD(var0 var1) SUB(var0 var0)) var0) MUL(var0 PDIV(var1 var1))) ADD(var0 var1)) SUB(var0 var0)) MUL(var0 PDIV(var1 var1))) SUB(ADD(var0 var1) SUB(var0 var0))) var0)))) ADD(PDIV(ADD(MUL(PDIV(var1 var1) var0) MUL(var0 var0)) ADD(var0 var1)) var1)))

Classifier of iteration 0. Accuracy 88.46%, coverage 100.00%

Iteration 0, removed 1698 instances, instances left 0

Overall stats at iteration 0. Accuracy 88.46%, error rate 11.54%, not classified 0.00%

Final classifier

cl0:SUB(var1 SUB(SUB(PDIV(PDIV(var1 var1) MUL(PDIV(SUB(SUB(var0 var0) var0) var1) var1)) ADD(PDIV(ADD(MUL(var0 var0) var0) ADD(var0 var1)) SUB(ADD(SUB(SUB(ADD(SUB(ADD(var0 var1) SUB(var0 SUB(ADD(ADD(var0 var1) var1) SUB(var0 var0)))) var1) var0) var0) var1) SUB(PDIV(ADD(ADD(PDIV(ADD(MUL(SUB(ADD(var0 var1) SUB(var0 var0)) var0) MUL(var0 PDIV(var1 var1))) ADD(var0 var1)) SUB(var0 var0)) MUL(var0 PDIV(var1 var1))) SUB(ADD(var0 var1) SUB(var0 var0))) var0)))) ADD(PDIV(ADD(MUL(PDIV(var1 var1) var0) MUL(var0 var0)) ADD(var0 var1)) var1)))

Stats on test data

Accuracy 88.42%, error rate 11.58%, not classified 0.00%

Total time: 14.045

**Explanation of the code**

The following lines are just a reproduction (for convenience) of the description fo the code in the coursework specification:

This case uses two files: GPwrapper.java and ClassifierGP.java. This strategy evolves the traditional genetic programming trees: mathematical formulas (hence, it is NOT a decision tree). These trees would have mathematical operators in their inner nodes and either the problem attributes or some constant numbers in their leaf nodes.

ClassifierGP.java is a simple wrapper that implements the Classifier interface and talks to the API of the open source GP library used in this case ([epochx](https://www.epochx.org/)). The only relevant function of the wrapper is classifyInstance. When a tree has to classify an instance it will run the mathematical expression encoded by the tree after initialising the variables with the values coming from the instance. This will generate a number. If this number if greater than 0, class 1 is predicted. Otherwise class 0 is predicted. A single tree predicts the whole dataset, and hence a single iteration of iterative rule learning is required.

The GPwrapper.java file has two main functions: getFitness() and generateClassifier(). The former function is called by the to evaluate individuals. It first creates a ClassifierGP object, and then iteratively classifies all instances in the training set. The number of classification mistakes is counted, and then (optionally) a penalty relative to the size of the tree is added to the (minimisation) fitness value.

The second main function is generateClassifier. The purpose of this function is to set up an run the genetic programming algorithm. Lines 57-79 set up the set of placeholder variables objects that are associated to the variable leaf nodes of the tree.

Lines 82-103 set up the selection of internal mathematical nodes that the GP algorithm has access to. You will see that the basic set of +, -, \* and / is uncommented, and many other possible operators are commented in the code. The explanation for all these operators is available at <https://www.epochx.org/javadoc/1.4/org/epochx/epox/math/package-summary.html>

Lines 106 to 119 show how to add constant leaf nodes to the set of the terminals that the GP algorithm has access to. Two types of constants are available: randomly initialised or with specific values, which sample at uniform intervals the range of values of the attributes in the dataset. These lines are commented but you can use them, or generate constants in different ways.

Lines 124-131 specify all the parameters of the GP algorithm. Line 140 runs the algorithm and line 141 extracts the best solution found in the run.

**How to measure performance & extract performance measures**

As explained in the coursework specification, there are a variety of performance measures for these experiments. The primary performance measure is the accuracy of the classifier on the test data, which is computed by the framework so you don’t have to do anything. However, from the running of the bio-inspired algorithm (GP in this case) we can also extract other performance measures: how many iterations of the GP algorithm were needed to generate a good classifier? What was the run-time of the algorithm?

**What to do with the code for the coursework?**

As in coursework 1, you should experiment with both the choice of GP operators and their parameters. You should also be systematic. Don’t randomly change many parameters/operators at once, because then you will not be able to determine what change was the one helping the most in improving the algorithm.

However, there are other aspects to explore. You will see in lines 57-119 of GPwrapper.java that the syntax for the evolved trees is set up. This means the set of internal nodes (mathematical operators) and the set of leaf nodes (dataset variables and constants). The choice and number of such nodes will make an impact in the performance of the algorithm. Moreover, you can also attempt to modify the fitness function (minimisation function) to e.g. increase the penalty associated to larger trees.