**CSC3423 Biologically-Inspired Computing**

**Coursework 2: Biologically-inspired computing for machine learning**

[Introduction 1](#_Toc528337796)

[What you have to submit 2](#_Toc528337797)

[Description of the code 2](#_Toc528337798)

[Example dataset and how to run experiments. 6](#_Toc528337799)

# Introduction

*Submission will be via the NeSS system https://ness.ncl.ac.uk*

The learning objectives of CSC3423’s coursework 2 are:

1. To understand how biologically-inspired methods can be applied to the specific computational task of machine learning
2. To critically evaluate the suitability of different types of biologically-inspired computing for a given task
3. To practice the skill of scientific experimental reporting
4. To learn how to integrate one’s code within a given code base

The objective of the coursework is to use biologically-inspired computing for machine learning.

You will be provided with (1) the (Java) source code (described below) of four different algorithms applied to machine learning: Genetic Algorithms, Genetic Programming, Particle Swarm Optimisation and Neural Networks and (2) a specific dataset, split into a training and a test set.

You will be asked to (1) select two of the biologically-inspired computing algorithms and tune them in the provided data, similarly to what you have done for coursework 1 and

(2) critically assess and compare the suitability of the methods you have

Notes:

- By the time this specification is released the lecture on machine learning will have happen. Please read it carefully as it introduces the nomenclature used in this specification.

- You can modify the parts of the code that are specific to each of the four possible bio-inspired algorithms you can apply to this task, but you should not modify the rest of the code. In the description of the code below the parts that you can modify are stated.

- It is VERY important that you use references and citations correctly and do not copy material from other sources without correctly citing it. Please see the University information on plagiarism.

# What you have to submit

You have to submit (through NESS) the following two items:

* The zipped complete source code of your program (including the provided framework and your own code)
* A report (of max. **2000 words**) in which you explain, for each of the two selected nature-inspired algorithms, (a) why you selected it, (b) how you tuned it for the provided dataset and (d) its performance for the provided dataset in terms of (1) iterations of the nature-inspired algorithm that it takes to generate a classifier (if it applies) (2) predictive power of the classifier on training/test data and (3) run-time of the training proces. Also, critically evaluate and compare the suitability and performance of the selected nature-inspired methods

Marks will be awarded for:

* Description of the selected nature-inspired algorithms and why you selected: 20 marks
* Description of the tuning process of the selected algorithms on the provided dataset: 30 marks
* Report of performance: 30 marks
* Critical comparison between methods and overall reflection: 20 marks

This assignment will comprise 25% of the total marks for the module.

# Description of the code

The code you are provided has two parts. Firstly, there is a common set of classes to handle the general machine learning aspects (the machine learning framework). These classes are shared across all the bio-inspired strategies for machine learning. The second part are the classes that are specific to each bio-inspired algorithm.

**1) The machine learning framework**. The (minimalistic) framework is composed of 8 Java classes. **You should not modify any of these files**:

* Attribute.java. This class holds the characteristics of a specific attribute in the dataset. From its functions the parts that are important to you are:
  + The name of the attribute – getName()
  + The minimal and maximal value of the domain of the attribute: minAttribute() and maxAttribute()
* Attributes.java. This class contains information about the metadata of the dataset: the number of attributes, the characteristics of each attribute and the number of classes. Important parts of this class for you are:
  + The number of attributes in the dataset getNumAttributes()
  + Fetching the Attribute object corresponding to each attribute: getAttribute(intpos)
  + Getting the number of classes in the problem: public attribute numClasses
* Classifier.java. This class is a Java interface that serves as a wrapper for the machine learning models generated by the bio-inspired algorithms, so that the part of the code that has to evaluate their performance can work without having to know what type of model you are using. The important functions of this interface are:
  + - public abstract int classifyInstance(Instance ins). This function will receive an instance and attempt to predict its class. If the classifier can predict its class it will return a class index (between 0 and the number of classes in the dataset-1). If it cannot predict this instance it will return -1
    - public abstract void printClassifier(). Prints to screen a textual description of the classifier
    - public double getFitness(). This function will return the last fitness value that was computed for this classifier
    - public void computeFitness(InstanceSet is). This function will receive a training set and will compute the fitness of the classifier. That is, how good is at predicting all/part of the instances of the training set. This function cat be used to evaluate the solutions while they are generated by any of the population-based biologically-inspired algorithms (e.g. genetic algorithms, genetic programming, memetic algorithms, ant colony optimisation, particle swarm optimisation).
    - public void computeStats(InstanceSet is). This function will print to screen a few performance statistics of the classifier given a dataset
* ClassifierAggregated.java. This class contains a collection of classifiers that together are a complete solution to the classification problem. That is, (should) cover the whole space of solutions.
* Control.java. This is the main class of the framework that controls the training process.
  + The main function implements the learning algorithm generally known as ***iterative rule learning***, that iteratively builds a solution (*ClassifierAggregated* object) by sequentially learning classifiers (that implement *Classifier* interface).
  + To learn each classifier the generateSubsolution(InstanceSet trainingSet) function is called. **Here is where you select what specific bio-inspired algorithm you are using** to train a classifier using *trainingSet* and return a *Classifier* object.
* Instance.java. This class represents each of the instances (i.e. data points, database rows) of the dataset. Important functions to you are:
  + public double geAttribute(int attr). Return the value of this instance for attribute *attr*
  + public int getClassValue(). Returns the class label of this instance
* InstanceSet.java. This object holds a complete instance set (a collection of instances). Functions important to you are:
  + public int numInstances(). Returns the number of instances of the instance set
  + public Instance getInstance(int whichInstance). Return an *Instance* object for instance with index *whichInstance*
  + public Instance[] getInstances(). Returns all instances in the object as an array
* ParserARFF.java. This class reads the file format (called ARFF) used to specify the dataset. You don’t need to access this file.

**2) Bio-inspired algorithm-specific files.**

You can modify the content of these files. At least you should play with the parameters of the methods, but if you feel adventurous, you can start to play with changing the knowledge representations of the methods (whenever feasible). **Hence, the tuning process is more complex than in coursework 1**.

1. **Genetic Algorithms Case.** The GA solution is composed of two files: GAwrapper.java and ClassifierSphere.java. The former class sets up the genetic algorithm, and you will see that it is very similar to the GA code in coursework 1 as it uses also the jenetics library. The latter class specifies the knowledge representation of the solutions evolved by the GA.

The GAwrapper.java is a relatively short file that specifies how to set up a genetic algorithm using the jenetics library and how to return objects following the Classifier interface to the coursework’s control class.

The class has three main functions:

1. evaluate. This is the function that the jenetics framework will call to evaluate the individuals of the GA population. It receives an individual of ‘Genotype<DoubleGene>’ type and returns a double. It makes use of the next function to build and evaluate a classifier from the genotype provided as argument to the function.
2. buildClassifier. This function receives a ‘Genotype<DoubleGene>’ object and builds a Classifier (class from the provided coursework framework) from the information encoded in such genotype. This function is an essential piece of the glue connecting the coursework framework to the jenetics library, as it makes the mapping from the basic individuals evolved by jenetics to the Classifiers that the provided coursework framework requires. If you look at the code you will see that the individuals evolved by jenetics multiplex the information of the sphere as a single vector containing, in this order, the coordinates of the centre of the sphere, its radius and its associated class. The associated class is defined as a continuous variable in the Genotype. To convert it to an integer we simply truncate the decimals. This is a quick fix for the fact that jenetics prefers to encode genotypes where all genes have the same type.
3. generateClassifier. This is the main function of this class. It is in charge of setting up and run the genetic algorithm using jenetics. The function has three main blocks: (1) defining the genotype. (2) setting up the components of the genetic algorithm and (3) running it. For part (1) the important aspect to remember is that the code cannot make assumptions about the number of attributes or the number of classes (the *machine learning* meaning of ‘class’) in the dataset. It needs to ask this information to the Attribute/Attributes classes (the *programming* meaning of ‘class’). For part (2), you will see in the code how the GA is parametrised: population size, number of elites, selection/crossover/mutation operators and parameters. For part (3), the code specifies the number of generations of the GA, sets the algorithm to run, collects the best individual from the population and returns a Classifier object made from that individidual.

About ClassifierSphere.java, the genetic algorithm is designed to optimise classifiers that are defined as a sphere. Each sphere is defined by (a) a center (b) a radius and (c) a class associated to the sphere. The classifier will be activated by any instance for which the euclidean distance from the sphere’s center to the instance is less than the radius, and will predict that it belongs to its associated class. The class ClassifierSphere.java implements this type of knowledge representation by extending the Classifier.java interface. A complete machine learning model will be composed by a set of spheres generated by the iterative application of the GA, following the *iterative rule learning* principle (as run by Control.java).

1. **The Particle Swarm Optimisation case**. The PSO solution uses four classes: PSOwrapper.java, MyFitnessFunction.java, MyParticle.java and (shared with the GA case) ClassifierSphere.java. The PSO cases uses the same strategy as the GA case: generate a classifier composed by a set of spheres. The three java files specific to PSO are again very similar to the PSO code in Coursework 1. All parts of the code have an equivalent to the GA case, as both solutions generate the same type of classifiers.
2. **The Genetic Programming case**. 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>

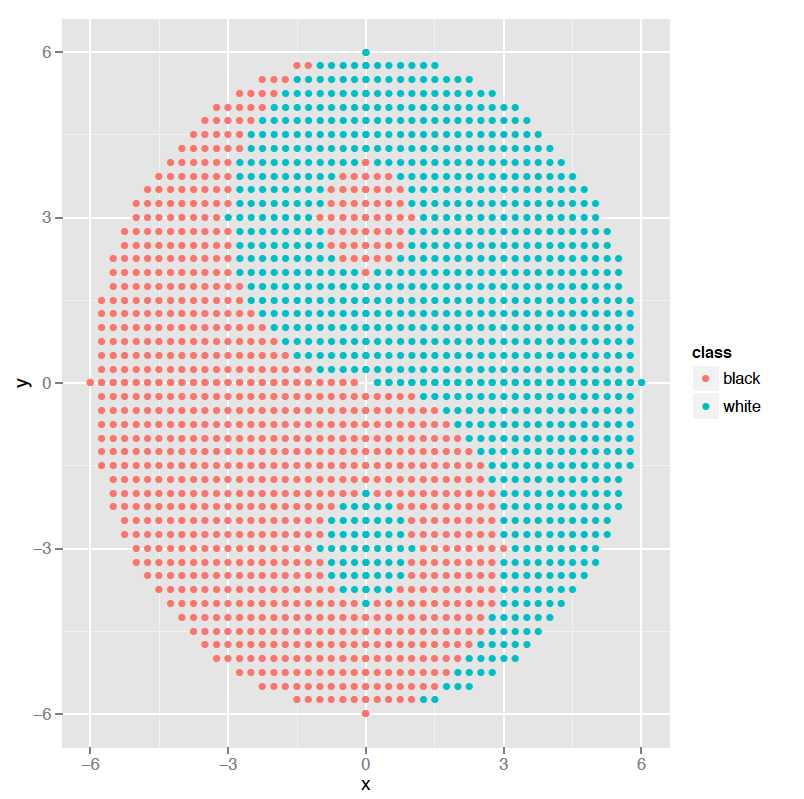
Lines 106 to 119 show how to add constant leaf nodes to the set of the terminals that the GP algorithm has access to. Two types of constants are available: randomly initialised or with specific values, which sample at uniform intervals the range of values of the attributes in the dataset. These lines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

1. **Neural network case**. The neural network used in this example is a simple multi-layer perceptron, which we will cover in the lectures in due course. A single java file covers this strategy: ClassifierMLP.java. this class extends the Classifier interface and it is in charge of both setting up the network, training it and then performing the classification of the data in the test set. In this case, a NN java open source library called Neuroph (<http://neuroph.sourceforge.net/>) has been used.

This class has four major functions: createInstance and createDataset are wrappers used to convert the data structures that hold the training data into the format that Neuroph requites. The ClassifierMLP (constructor) function sets up and trains the network. Line 41 sets up the topology of the network: the number of layers and number of neurons per layer. Lines 42 and 43 set up the parameters of the learning process, and line 44 performs the training. classifyInstance is the function that performs predictions of new instances. It queries the instance to specify the values for the network’s input layer, runs the whole network, gets the output and converts this output into a predicted class label. Like in the GP case, a single network predicts the whole dataset, so the training process would finish after one iteration of classifier generation.

# Example dataset and how to run experiments.

For this coursework you are provided with one dataset that represents the “Ying-Yang” figure to the right. The dataset is split into two files: training.arff and test.arff. The format of these files is called ARFF (<https://www.cs.waikato.ac.nz/ml/weka/arff.html>), but you don’t have to worry about it because my code handles the parsing of these files. In any case, these are text files, so you can open them with any text editor.

If you look at the Control.java file you will see that the program receives two arguments. The first one is the training file and the second one is the test file.

Here is an example output of the framework for the GA case, when calling the code from the UNIX command line:

$ java -cp jenetics-4.2.0.jar:. Control training.arff test.arff

Relation name TAO\_grid

Attribute name x

Attribute name y

Attribute name class

Classifier of iteration 0. Accuracy 100.00%, coverage 25.62%

Iteration 0, removed 435 instances, instances left 1263

Overall stats at iteration 0. Accuracy 25.62%, error rate 0.00%, not classified 74.38%

Classifier of iteration 1. Accuracy 100.00%, coverage 6.10%

Iteration 1, removed 77 instances, instances left 1186

Overall stats at iteration 1. Accuracy 30.15%, error rate 0.00%, not classified 69.85%

Classifier of iteration 2. Accuracy 100.00%, coverage 5.65%

Iteration 2, removed 67 instances, instances left 1119

Overall stats at iteration 2. Accuracy 34.10%, error rate 0.00%, not classified 65.90%

Classifier of iteration 3. Accuracy 100.00%, coverage 5.00%

Iteration 3, removed 56 instances, instances left 1063

Overall stats at iteration 3. Accuracy 37.40%, error rate 0.00%, not classified 62.60%

Classifier of iteration 4. Accuracy 100.00%, coverage 3.39%

Iteration 4, removed 36 instances, instances left 1027

Overall stats at iteration 4. Accuracy 39.52%, error rate 0.00%, not classified 60.48%

Classifier of iteration 5. Accuracy 100.00%, coverage 0.78%

Iteration 5, removed 8 instances, instances left 1019

Overall stats at iteration 5. Accuracy 39.99%, error rate 0.00%, not classified 60.01%

Classifier of iteration 6. Accuracy 100.00%, coverage 4.61%

Iteration 6, removed 47 instances, instances left 972

Overall stats at iteration 6. Accuracy 42.76%, error rate 0.00%, not classified 57.24%

Classifier of iteration 7. Accuracy 100.00%, coverage 5.76%

Iteration 7, removed 56 instances, instances left 916

Overall stats at iteration 7. Accuracy 46.05%, error rate 0.00%, not classified 53.95%

Classifier of iteration 8. Accuracy 100.00%, coverage 3.38%

Iteration 8, removed 31 instances, instances left 885

Overall stats at iteration 8. Accuracy 47.88%, error rate 0.00%, not classified 52.12%

Classifier of iteration 9. Accuracy 100.00%, coverage 0.90%

Iteration 9, removed 8 instances, instances left 877

Overall stats at iteration 9. Accuracy 48.35%, error rate 0.00%, not classified 51.65%

Classifier of iteration 10. Accuracy 100.00%, coverage 0.23%

Iteration 10, removed 2 instances, instances left 875

Overall stats at iteration 10. Accuracy 48.47%, error rate 0.00%, not classified 51.53%

Classifier of iteration 11. Accuracy 97.59%, coverage 99.43%

Iteration 11, removed 870 instances, instances left 5

Overall stats at iteration 11. Accuracy 98.47%, error rate 1.24%, not classified 0.29%

Classifier of iteration 12. Accuracy 100.00%, coverage 100.00%

Iteration 12, removed 5 instances, instances left 0

Overall stats at iteration 12. Accuracy 98.76%, error rate 1.24%, not classified 0.00%

Final classifier

cl0:Center -5.269630112376646,-2.1592710048762007, radius 4.5276756851482105, class 0

cl1:Center -0.529436639261716,-5.636268635897503, radius 1.7159399263228567, class 0

cl2:Center -5.98705744221019,4.500376002876093, radius 3.258870493408771, class 0

cl3:Center 0.8532685586094821,-1.2595894688101659, radius 1.0394697606833514, class 0

cl4:Center -0.7975599645542051,-1.0314858625539287, radius 1.2333762424179824, class 0

cl5:Center -4.361222695354619,2.130233042760395, radius 1.5238817153796735, class 0

cl6:Center -0.06640211152655873,2.9319544762663803, radius 0.9415824759447906, class 0

cl7:Center 1.6355326222492081,-3.954865014697779, radius 1.1246845171513922, class 0

cl8:Center 1.9542949744435543,-2.3782568557890826, radius 1.0122750276843804, class 0

cl9:Center -4.038742992494392,5.618094920624378, radius 1.9000287049849236, class 0

cl10:Center -3.133417492388924,-5.6404632128261545, radius 3.1882377875102335, class 0

cl11:Center 4.240011479597367,-3.990627095490598, radius 11.112969197391067, class 1

cl12:Center 2.358916959838959,4.799267827847675, radius 8.992170113691365, class 0

Stats on test data

Accuracy 96.32%, error rate 3.68%, not classified 0.00%

Total time: 8.417

From this output there are a few things to note:

* The solution consisted in 13 spheres iteratively covering the instances of the training set. Please note that for the last iteration the number of instances left was 0. This is the end condition of the learning loop.
* The overall stats for the last iteration provide you with the performance measures of the collection of classifiers for the training data
* Afterwards, the classifier (the set of spheres) is printed.
* Next you have the stats on test data. A good classifier is one that is able to obtain good performance on the test data, as it shows that it is able to generalise from the training data.
* Finally, the last line tells you the run-time of the training and test process in seconds.

To run the code for the other three classifiers, you need to edit Control.java and uncomment the appropriate lines of code from the generateSubsolution() function.

The command line instructions to run them would be:

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

PSO case: java -cp jswarm-pso\_2\_08-recompiled.jar:. Control training.arff test.arff

NN case: java -cp neuroph-core-2.93.jar:slf4j-api-1.7.5.jar:. Control training.arff test.arff