# CSC3423 Bio-Computing

## Coursework

## "Nature-inspired computing for machine learning"

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

This document describes the specification for **both** pieces of coursework of CSC3423, identified in NESS as "coursework proposal" and "practical report".

The learning objectives of CSC3423's coursework are:

1. To understand how nature-inspired methods can be applied to the specific computational task of machine learning
2. To critically evaluate the suitability of different types of nature-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 nature-inspired computing for machine learning. You will be provided with the (Java) source code of a general framework for machine learning (described below) in which you will be asked to (1) select two different types of nature-inspired computing and integrate (and tune) them within this framework to train classifiers and (2) critically assess and compare the suitability of the methods you have selected and implemented. A dataset to perform your experiments is also provided.

Notes:
- The first lecture after the reading week will provide an in-depth description of the coursework
- You are free to implement the whole code of the nature-inspired algorithms or integrate any of the numerous existing open source implementations of these methods into the provided machine learning framework
- You should not modify the provided framework except in the places where you are explicitly told to do so. If you really feel the need to modify the code, please contact the lecturer **before** the submission to seek permission.
- Your code should not make any assumption about the characteristics of the data. Use the provided API to determine the structure of the problem. That is, make your code **generic.**

For both pieces of assessment 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.

**Specification of “coursework proposal”**

For this piece of assessment you have to write a report where you explain your proposal of **design** for the coursework. That is, for each of the two chosen nature-inspired algorithms, you have to explain (1) why you think it is suitable for the coursework’s machine learning task and (2) your strategy for integrating such algorithm within the provided software framework, including the choice of knowledge representation used to tackle the machine learning task.

This assignment will comprise 10% of the total marks of the module.

Marks will be awarded for:

- Justification for the selected nature-inspired algorithms: 30 marks
- Description of the knowledge representation: 20 marks
- Software design for the integration of the nature-inspired algorithms within the provided framework: 30 marks
- Figures: 10 marks
- Writing: 10 marks

The report has a word limit of **1000 words**.

**Specification of “practical report”**

The second piece of assessment will focus on the full implementation, testing and critical evaluation of your proposal of using nature-inspired algorithms for the provided machine learning task. You are allowed to diverge from the design previously submitted in the “project proposal” assessment.

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 integrated it into the framework, (c) how you tuned it to the problem at hand 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 (2) predictive power of the classifier 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:

- Revised justification and design for the selected nature-inspired algorithms and knowledge representations: 20 marks
- Description of your implementation: 30 marks
- Description of how your implementation was adjusted and tuned for the provided dataset: 10 marks
- Report of performance: 20 marks
- Critical comparison between methods and overall reflection: 20 marks
- **BONUS**: Generate visualisations of the solutions of your method and/or the functioning of your implementation of the nature-inspired methods: Up to 10 marks.

This assignment will comprise 40% of the total marks for the module.

**Description of the machine learning framework**

**- NOTE:** Before reading the text below, please be familiar with all the machine learning nomenclature that is explained in “coursework preparation lecture” of the module.
**-** You can download the machine learning framework from Blackboard at “Teaching Materials > Coursework”.

You are provided with a (minimalistic) machine learning framework in Java. The framework is composed of 11 Java classes:

- Attribute.java. This class holds the characteristics of each 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(int pos)
    - Getting the number of classes in the problem: public attribute numClasses
- Classifier.java. **This is one of the most important classes for you**. It contains the interface for the classifiers you will have to generate (using nature-inspired methods).
    - You will have to implement the following methods:
        - public abstract int classifyInstance(Instance ins). This function will receive an instance and attemps 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
    - Moreover, you are provided with three auxiliary functions.
        - 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. You can use this function to evaluate the solutions while they are generated by any of the population-based nature-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. **You don’t have to modify/access this class directly**
- ClassifierRandomSphere.java. This is an example of a toy classifying that implements the *Classifier* interface. More details in the next section
- 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 have to put the code of the nature-inspired algorithm** 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 getRealAttribute(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
- MTwister.java. This class implements the *Mersenne Twister* pseudo-random number generator (PRNG). You don't need to access this file.
- ParserARFF.java. This class reads the file format (called ARFF) used to specify the dataset. You don't need to access this file.
- Rand.java. This class is the interface to the pseudo-random number generator. Four funtions are important for you:
    - public static void initRand(). This function initialises the generator using the standard Java PRNG. This is the function called from Control.java. Hence, each time that you call the program the PRNG will generate different numbers.
    - public static void initRand(long seed). Initialises the PRNG with a specific seed. This means that the program will always provide the same output if it is specified with the same seed.
    - public static double getReal(). Returns a real number between [0,1]
    - public static int getInteger(int uLow, int uHigh) . Returns an integer number betwee *uLow* and *uHigh.*

**Example (toy) classifier**

The ClassifierRandomSphere.java class contains a toy classifier in the shape of a sphere. This 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 with the center is less than the radius, and will predict that it belongs to its associated class.

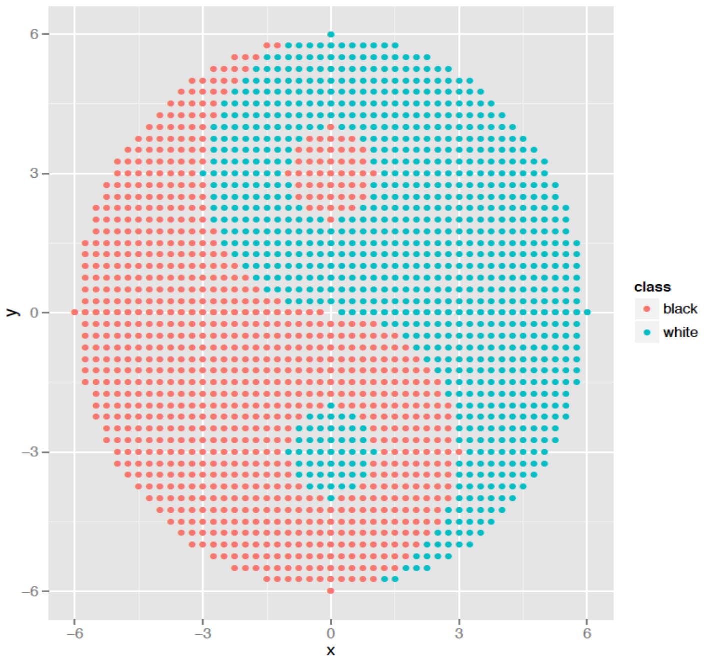
The "training" process of the classifier, contained in the constructor of the class and in function computeMajorityClass, consists in the following steps:

1. Select at random instance from the training set. Set up the centre of the sphere to be that instance
2. Set up a radius at random to be between 0 and 4 (which is good enough for the example dataset. In the real world, this number would need to be decided depending on the characteristics of the dataset).
3. Identify all instances from the training set that are within the sphere. Set up the class of the sphere to be the majority class of these instances.

Through these functions you will see examples of how to access the *InstanceSet*, *Instance*, *Attributes* and *Rand* objects. Moreover, you will see how the *doMatch*, *getClassValue* and *printClassifier* functions (the three functions from the *Classifier* interface that you have to implement) work.

**Example dataset and how to run experiments.**

For this coursework you are provided with a dataset that represents the "Ying-Yang" figure. Each instance has two attributes, x and y and a class with values "black" and "white". If you plot the instances as if they were coordinates you would see this:



The dataset is divided into two files: TrainFold0 and TestFold0. The former is the training set. That is, the set of instances that are used to generate the classifier. The latter is what is called the test set. It is used to validate the predictive power of the classifier by testing in on *unseen* data. 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 toy random sphere classifier (The output depends on the random seed) when calling the code from the UNIX command line:

```
$ java Biocomputing.Control TrainFold0 TestFold0
Random seed is 953555860076483139
Relation name TAO_grid
Attribute name x
Attribute name y
Attribute name class

Classifier of iteration 0. Accuracy 100.00%, coverage 7.30%
Iteration 0, removed 124 instances, instances left 1574
Overall stats at iteration 0. Accuracy 7.30%, error rate 0.00%, not classified 92.70%

Classifier of iteration 1. Accuracy 73.26%, coverage 22.03%
Iteration 1, removed 374 instances, instances left 1200
Overall stats at iteration 1. Accuracy 23.44%, error rate 5.89%, not classified 70.67%

Classifier of iteration 2. Accuracy 55.75%, coverage 13.31%
Iteration 2, removed 226 instances, instances left 974
Overall stats at iteration 2. Accuracy 30.86%, error rate 11.78%, not classified 57.36%

Classifier of iteration 3. Accuracy 64.80%, coverage 11.54%
Iteration 3, removed 94 instances, instances left 880
Overall stats at iteration 3. Accuracy 35.04%, error rate 13.13%, not classified 51.83%

Classifier of iteration 4. Accuracy 97.04%, coverage 17.90%
Iteration 4, removed 240 instances, instances left 640
```

```
Overall stats at iteration 4. Accuracy 48.65%, error rate 13.66%, not classified 37.69%

Classifier of iteration 5. Accuracy 100.00%, coverage 3.89%
Iteration 5, removed 33 instances, instances left 607
Overall stats at iteration 5. Accuracy 50.59%, error rate 13.66%, not classified 35.75%

Classifier of iteration 6. Accuracy 100.00%, coverage 0.71%
Iteration 6, removed 12 instances, instances left 595
Overall stats at iteration 6. Accuracy 51.30%, error rate 13.66%, not classified 35.04%

Classifier of iteration 7. Accuracy 39.13%, coverage 2.71%
Iteration 7, removed 19 instances, instances left 576
Overall stats at iteration 7. Accuracy 51.94%, error rate 14.13%, not classified 33.92%

Classifier of iteration 8. Accuracy 98.23%, coverage 23.32%
Iteration 8, removed 279 instances, instances left 297
Overall stats at iteration 8. Accuracy 67.96%, error rate 14.55%, not classified 17.49%

Classifier of iteration 9. Accuracy 39.47%, coverage 28.80%
Iteration 9, removed 204 instances, instances left 93
Overall stats at iteration 9. Accuracy 75.32%, error rate 19.20%, not classified 5.48%

Classifier of iteration 10. Accuracy 100.00%, coverage 3.77%
Iteration 10, removed 38 instances, instances left 55
Overall stats at iteration 10. Accuracy 77.56%, error rate 19.20%, not classified 3.24%

Classifier of iteration 11. Accuracy 72.34%, coverage 2.77%
Iteration 11, removed 13 instances, instances left 42
Overall stats at iteration 11. Accuracy 78.33%, error rate 19.20%, not classified 2.47%

Classifier of iteration 12. Accuracy 92.50%, coverage 7.07%
Iteration 12, removed 13 instances, instances left 29
Overall stats at iteration 12. Accuracy 79.09%, error rate 19.20%, not classified 1.71%

Classifier of iteration 13. Accuracy 98.45%, coverage 7.60%
Iteration 13, removed 18 instances, instances left 11
Overall stats at iteration 13. Accuracy 80.15%, error rate 19.20%, not classified 0.65%

Classifier of iteration 14. Accuracy 100.00%, coverage 3.83%
Iteration 14, removed 7 instances, instances left 4
Overall stats at iteration 14. Accuracy 80.57%, error rate 19.20%, not classified 0.24%

Classifier of iteration 15. Accuracy 100.00%, coverage 15.31%
Iteration 15, removed 4 instances, instances left 0
Overall stats at iteration 15. Accuracy 80.80%, error rate 19.20%, not classified 0.00%

Final classifier
cl0:Center -4.5,-3.75, radius 2.3089691601481683, class 0
cl1:Center 4.25,-2.25, radius 3.491313951902863, class 1
cl2:Center -2.75,4.75, radius 3.00178042496596, class 1
cl3:Center 2.0,-5.0, radius 2.5976416167331955, class 0
cl4:Center 3.0,2.25, radius 2.6127128458145803, class 1
cl5:Center 0.25,5.5, radius 1.4136938996179247, class 1
cl6:Center 0.25,-1.25, radius 0.39390543764315206, class 0
cl7:Center 0.75,3.75, radius 0.9924677473009722, class 0
cl8:Center -4.5,-1.25, radius 3.5752415749186746, class 0
cl9:Center -0.75,-0.75, radius 3.1527679039055405, class 1
```

```
cl10:Center -2.0,-4.5, radius 1.1726955168816948, class 0
cl11:Center 0.0,2.5, radius 0.9757796388528728, class 0
cl12:Center -0.75,-5.25, radius 1.8635908695551546, class 0
cl13:Center 2.0,5.0, radius 2.0737853623167113, class 1
cl14:Center -5.0,2.75, radius 1.529044061323871, class 0
cl15:Center 5.75,1.0, radius 3.439464335199321, class 1

Stats on test data
Accuracy 82.11%, error rate 17.89%, not classified 0.00%

Total time: 0.806
```

From this output there are a few things to note:

- The solution consisted in 16 random 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. **When you tune the nature-inspired method to generate a better classifier, you have to optimise the test accuracy.**
- Finally, the last line tells you the run-time of the training and test process in seconds.