UFCFY3-15-3 BioComputation: Assignment – Getting Started

[Hand-in Deadline: November 28th 2019**]**

The hand-in for the assignment is a report on your attempts to solve a set of classification problems as effectively as possible using any form of evolutionary intelligence covered in the course, building upon your own genetic algorithm code developed in the first few lab sessions. The idea is that you will experience a classification data mining problem first-hand and produce the sort of output a commercial environment might expect.

In classification data mining a model is built using one or a variety of techniques from available data so that the "class" of previously unseen variable combinations can be estimated. A simple bank customer example was used in the genetic programming lecture (EA3) where the classification was whether or not a customer is likely to repay a loan, where a few variables about each customer were available. Building the model involves repeatedly passing the data into the model and correcting it based upon whether or not it gave the correct classification each time. I suggest you start by using a rule-based scheme to build the model, something we covered briefly in the second evolution lecture (EA2). Rules are in the general form:

IF <condition> THEN <output>

Building the model to match the data is the search problem and the assignment is to use a genetic algorithm (GA) to define what goes in the condition and output section of each rule. In essence, what you are doing is altering the fitness function of your binary GA code. At the moment you probably have a function that takes the candidate solution's genes as a parameter and returns the fitness of those genes on the counting ones task, ie, something like:

```
int fitness_function( individual solution) {

        int fitness=0;

        for( i=0 to gene_length-1) {
                if( solution.genes[i] == 1 ) fitness++
        }

        return fitness;
}
```

What this function now needs to do is turn an individual's genes – a string of ones and zeros – into a set of rules and to then see how well those rules model the data set. That is, each individual is now seen as a set of rules. The first data set (data1.txt) contains binary data each with six variables and one for the class. Hence each rule needs a condition containing five bits and an output of one bit, eg, IF <010001> THEN <1>. If each rule base consists of ten rules then each individual needs (6+1)x10 = 70 binary genes: the binary string is the rules concatenated together. Hopefully changing the number of genes in an individual is easy in your code – a static int perhaps?

I suggest you make a new data type for the rules, eg, in C:

```
#define ConL 6

typedef struct {
        int cond[ConL];
        int out;
} rule;
```

Then your fitness function can start by declaring an array of type rule and filling the conditions and output of each rule from the genes of the individual passed in. Note the use of an index variable (k) to move along the string of genes as the rule parts are filled:

```
#define NumR 10

int fitness_function( individual solution) {
        int fitness=0;
        int k=0;
        rule rulebase[NumR];

        for( i=0 to NumR-1) {
                for( j=0 to ConL-1) {
                        rulebase[i].cond[j] = solution.genes[k++];
                }
                rulebase[i].out = solution.genes[k++];
        }
……..

}
```

At this stage it might be an idea to print out both the individual's genes in the original string and the set of rules in their parts to check they match.

Note you will also need to open the data file and read it into your code in an array of another data type, eg:

```
#define Vars 6

typedef struct {
        int variables[Vars];
        int class;
} data;
```

To calculate the fitness of the rules, the data set needs to be passed over them one example at a time and a count made of each time the *first* rule whose condition matches the data item has an output that matches the class label:

```
for( i=0 to size_of_data-1 ) {
        for( j=0 to NumR-1) {
                if( matches_cond( data_example, rulebase[j].cond ) {
                        if( matches_output( data example, rulebase[j].output ) {
                        fitness++;
                        }
                        break; // note it is important to get the next data item after a match
                }
        }
}
```

As described in the EA2 lecture slides, matching here means having the same bit value in each location. Clearly this means the best thing the GA can find is 10 rules that match 10 different data items. Test your code until you are able to reliably evolve a best individual with fitness 8 or 9.

The next step is to introduce the # into the conditions of the rules which means making some alterations to the random seeding of your basic GA and to the mutation operator. Again, there is some info in the second evolution lecture to help regarding matching, etc. Once that's up and running, you should be able to evolve individuals with much higher fitnesses, on both data1 and data2. Don't worry about splitting for training and testing on the two binary data sets. On data1 ten rules is plenty to correctly classify all of the data. Print out the rules of the final best individual and see if you can ascertain the structure of the data.

We can talk about that step and the subsequent changes to deal with data 3 and 4 in the labs.