CSC3423 Bio-Computing Practical Report

0 Contents

N.B. I have altered 'Control' to provide visualisation and extra settings for conducting the experiment. I have also commented files where appropriate to aid clarity. To run either algorithm simply change the enumeration near the top. Static final member fields may also be edited to change test conditions (such as whether to show visualisations).

This report is 1995 words sans the appendix.

1 Genetic Algorithm

1.1 Revised Justification

I chose to pursue a genetic algorithm to optimise the classifier set as the same principles that were employed during each step of the process are transferrable to many different kinds of future problems. Any problem that has inputs that can be optimised, and furthermore can have a fitness function created for it to attest to the effectiveness of that specific construction, can be approached in this way. Since it suited the learning outcomes of the project, and since the overall solution is tiered to incorporate both optimisation and classification, it provided many opportunities to contrast different options taught in-class in order to produce the best result.

Additionally, I chose to explore 'hyperrectangles' as classifiers because humans are more adept at thinking geometrically, and therefore aspects of the problem could be interpreted through visualisation in a straightforward manner. The fitness function itself was already present, and therefore I wanted to see how the theory underpinning the aforementioned concepts affected the data. Regretfully, I did not have time to explore parallelism further than online research.

1.2 Implementation

I did not strongly deviate in my implementation from my proposal, however I discovered some unforeseen but necessary steps, while attempting to follow the same conventions set forth in the example code.
I created the class GeneticAlgorithmWrapper to facilitate the creation of classifiers using the 'Jenetics' framework.

Chromosomes are created with entries for the origin and size (bound) of each dimension of the problem, as well as the class of the classifier itself. These values are constrained to the range of values the InstanceSet provides, however they can be larger (my implementation works for any number of dimensions, and therefore all datasets). Since the range for each dimension may be different, the location of the origin point for each dimension is unlikely to be in tandem with the rest (so they are spatially disjoint). This could be partly enforced using linear interpolation, however it did not prove problematic during testing.

This blueprint is converted to a genotype, and then an engine built using immutable arguments that control how the algorithm will operate.

In the same class, I created a method evaluate to allow the genetic algorithm to evaluate the fitness of an individual, and another helper method buildClassifier to convert between genes and classifier objects. This kind of data transmission appears common when using a multitude of libraries; prone to errors, the codebase itself would benefit from unit testing given more time.

Finally, I created a new classifier ClassifierRectangle to extend Classifier. This performs simple distance calculations to classify a point if is within the bounds for every dimension relative to the origin.

1.3 Tuning & Performance

For each table row I have calculated the average of 10 of the same run using different random seeds in order to reduce the impact of anomalous data.

Some aspects of tuning were systematic and precursor to control variables. These were things like choosing the offspring selector and certain 'alterer's. I compared roulette-wheel to tournament selector, and single-point to multi-point crossover as these were the most appropriate options. Roulette-wheel, complicit with its theory, produced a more sporadic results range, and it along with multi-point crossover yielded less accurate results. Therefore, I moved forward using a tournament selector with single-point crossover – although it took longer the difference was negligible for the performance gain.

Selector Type	Accuracy (%)	Error (%)	Time (s)
Roulette Wheel (with Single Point Crossover)	84.53%	15.47%	5.03
Tournament (with Single Point Crossover)	93.68%	6.32%	5.73
Roulette Wheel (with Multiple Point Crossover)	87.89%	12.11%	4.64
Tournament (with Multiple Point Crossover)	89.47%	10.53%	4.62

I observed defaults in the library documentation as a baseline standard for experimentation. This enabled me to change one variable at a time and derive its effects.

Variable	Default	Optimal Value
Offspring Fraction (Survivors Size)	0.6	0.4
Maximum Generations	70	100
Population Size	50	250
Crossover Probability	0.2	0.4
Mutation Probability	0.15	0.5
Sample Size (Selector)	2	2

I tested each at their extremes to see which had an affect. Increasing the sample size and survivor fraction reduced the accuracy marginally and increased the number of classifiers produced by roughly 50%, as less fit candidates were being kept. I thought the number of classifiers produced would be in positive linear correlation with the time taken to compute an entire solution, however this assumption was incorrect and therefore they do not have to be minimised.

Mutation and crossover were more nuanced in their changes. Lower values for both resulted in more overlapping classifiers as they converged, and values that were too high saw a downward trend in accuracy while the number of classifiers increased drastically.

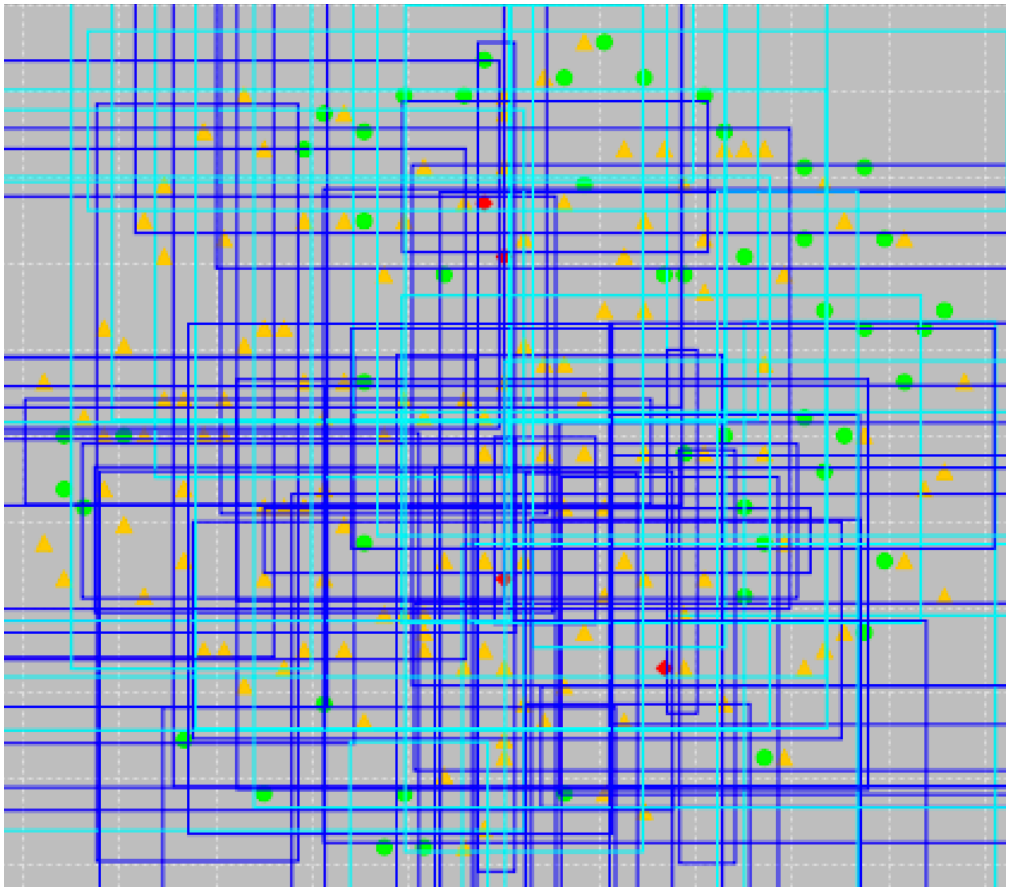

Fig. 1. A solution of the genetic algorithm with mutation and crossover probability of 0.01 generates 76 classifiers, many overlapping.

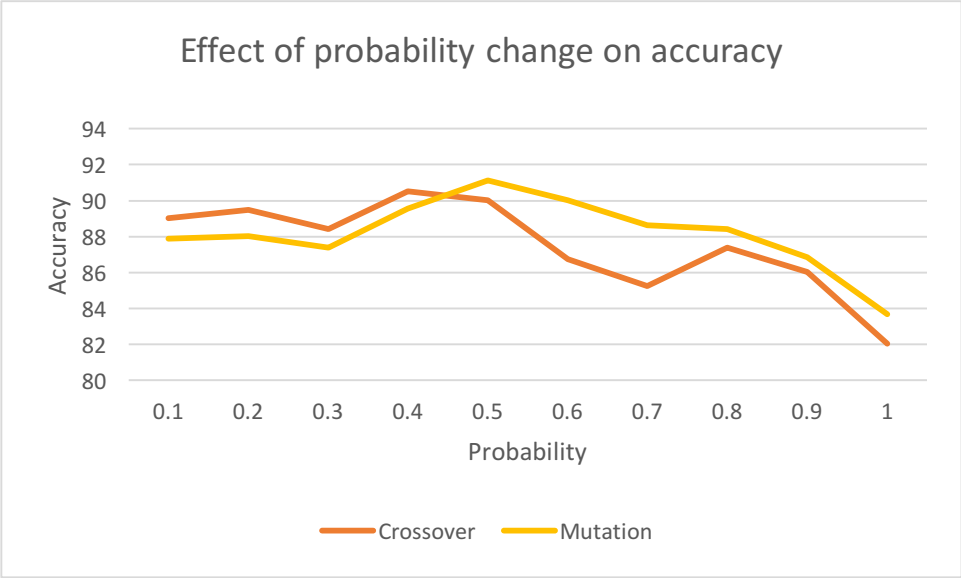

Fig. 2. Finding the optimal values for crossover and mutation probability relative to the final fitness of each.

Finally, the variables with most influence, population size and maximum generations were tested.

Population Size	Max Generations	Accuracy (%)	Error (%)	Time (s)
50	70	88.65	10.35	7.18
100	70	90.00	10.00	9.21
250	70	94.21	5.79	13.84
500	70	92.63	6.84	18.744
750	70	91.05	8.95	27.43
50	10	84.21	15.79	4.02
50	50	87.95	11.53	5.43
50	100	91.05	8.95	9.19
50	200	90.53	9.47	11.75
50	500	90.00	9.47	13.66

With more time I would have liked to have automated the optimisation of these input parameters to find the ‘true optimum’. This process would benefit from features outside of the scope of this project, such as multi-threading.

1.4 Performance

Below is the final performance for the datasets (3 and 4 for simplicity) provided, using the best interpolated values.

Dataset	Dimensions	Accuracy	Time	Classifier Count
3	6	94.62	13.95	46
4	2	94.21	6.23	28

As might be expected, it is slower on larger datasets with more dimensions.

2 Neural Network

2.1 Revised Justification

I chose to implement a neural network because they represent another topic that provides extensive further application now that I grasp the core concepts. I felt they would be a good contrast to genetic algorithms, since they operate in a completely different manner. Decisions regarding their structure can be ad-hoc, however they could be optimised by a genetic algorithm, which makes them a suitable counterpart for this project.

They are well suited to solving classification problems, and the libraries available to implement them are comprehensive and well-documented. They work with many dimensions, and their learning, if unintelligible is at least easily transferrable.

2.2 Implementation

I chose to use the 'Encog' library after reading about the performance issues with 'Neuroph'. The provided codebase was not entirely suitable for integration since it enforces the notion of a 'subsolution' and hoists iterative status printing to the highest-level method; there is only one network in my implementation, training takes place in the wrapper class and the classifier itself is forced to implement some redundant methods. It was, however, a useful lesson in working around the limitations of legacy architecture.

Due to the nature of a network taking an input and providing an output, it was impossible for it to not provide total coverage, even though this coverage is subject to error.

I created the class NeuralNetworkWrapper to handle training of the network. It feeds the network training data and iterates over it with the stopping condition of a reported error rate less than 0.1 (to finish early as this is sufficient) or a maximum number of epochs reached (to prevent infinite looping).

The NeuralNetworkClassifier itself stores a BasicNetwork and ReslientPropagation trainer (ergo the current state of the network in-memory), along with some methods to convert the InstanceSet and Instance objects into a format the library understands (BasicMLDataSet and BasicMLDataPair). It constructs the network using some immutable arguments that control how learning will operate, such as the structure of the network itself, and the types of neurons contained.

2.3 Tuning

As with the genetic algorithm, there were some higher-level decisions to be made that would affect what I could and couldn't change. Chiefly, the structure of the network, the type of activation function present in the neurons and the method of learning to be employed.

With back-propagation a learning rate and momentum could be set, however the accuracy using back-propagation, against my expectations, underperformed the recommended resilient-propagation during tuning by about 8%. Opting for the better outcome, I lost the ability to tweak certain aspects of that type of learning, such as the learning rate and momentum.

The maximum number of epochs was capped at 1000 since most runs started to fluctuate around a peak after a certain point.

I had intended to use a 'sigmoid' function to push values towards a binary output, however 'TANH' outperformed this (and 'ReLU'). Mixing the kinds of functions used in each neuron resulted in poor performance (roughly 50.53%).

Activation Type	Accuracy (%)	Error (%)	Time (s)
TANH	93.68	5.79	9.42s
Sigmoid	89.47	10.53	7.42s

The solution also performed better with bias nodes active, and better still when only active on the input and hidden layers, omitting output.

Left to experiment with the structure, I experimented with a different number of internal neurons per hidden layer however this did not yield anything particularly interesting or performant, so they have remained the same.

Internal Layers	Neurons per Internal Layer	Accuracy (%)	Error (%)	Time (s)
1	5	90.53	9.47	4.02
1	10	92.11	7.37	6.817
1	20	90.53	9.47	9.15
2	5	88.42	11.58	7.84
2	10	95.26	4.21	9.28
2	20	94.21	5.26	15.32
3	5	95.26	4.74	7.04
3	7	96.37	2.63	10.09
3	10	93.16	6.84	12.67
3	20	94.21	4.74	25.71

With more nodes, accuracy had an initial positive jump however the following increments increments were much smaller.

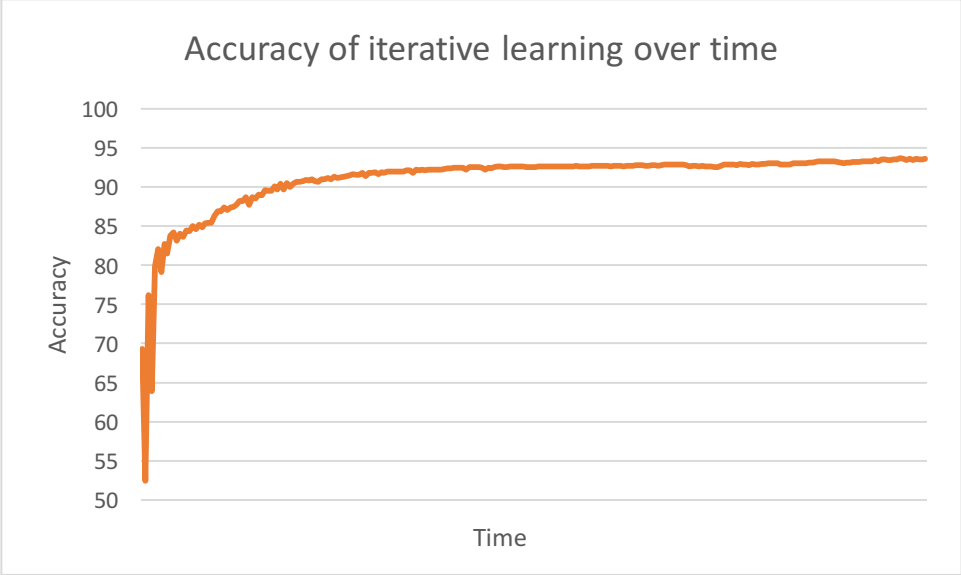

Fig. 3. The learning rate slows over time.

2.4 Performance

Below is the final performance for the datasets (3 and 4 for simplicity) provided, using the best settings.

Dataset	Dimensions	Accuracy (%)	Time (s)
3	6	96.31	10.05
4	2	97.89	9.31

During visualisation the node classification changes rapidly, with less flipping as it converges towards the ideal solution.

3 Conclusion

The neural network performed better than the genetic algorithm for the two key metrics of accuracy and execution time.

Algorithm	Accuracy (%)	Time (s)
Genetic Algorithm	94.64	14.41
Neural Network	96.31	10.02

Since the difference is quite small, it may be preferential to choose one data representation over another. In this case, the shape of the rectangular classifier means that it mightn't necessarily capture more resolute areas of the pattern, and it is therefore more prone to error. Visualisations were disabled during testing so as to not impact performance, however the neural network was harder to make sense of.

The genetic algorithm had more variables to tune, however most changes made an immediate impact. The neural network, on the other hand, took more guess-work.

In some cases, it may be preferable to eschew the full coverage of a neural network in favour of correct classification. Depending on the problem, it may be better to identify gaps in knowledge than to make an incorrect prediction. The genetic algorithm builds upon itself, and anything that isn't close enough to the ideal can be thrown away. Contrastingly, the neural network can essentially unlearn, which means that care must be taken during the training process.

There were many more things that could have been tested with more time, however I was happy to note key differences, similarities, correlations, and furthermore use these to make improvements during development.

4 Appendix

The contents of this appendix are also provided in the submission folder.

4.1 Sample Output

Genetic Algorithm

```
Random seed is 1

Relation name TAO_grid

Biocomputing.Attribute name x
Biocomputing.Attribute name y
Biocomputing.Attribute name class

Classifier of iteration 0. Accuracy 100.00%, coverage 21.85%
Iteration 0, removed 371 instances, instances left 1327
Overall stats at iteration 0. Accuracy 21.85%, error rate 0.00%,
not classified 78.15%

Classifier of iteration 1. Accuracy 99.53%, coverage 12.60%
Iteration 1, removed 151 instances, instances left 1176
Overall stats at iteration 1. Accuracy 30.68%, error rate 0.06%,
not classified 69.26%

Classifier of iteration 2. Accuracy 100.00%, coverage 4.24%
Iteration 2, removed 60 instances, instances left 1116
Overall stats at iteration 2. Accuracy 34.22%, error rate 0.06%,
not classified 65.72%

Classifier of iteration 3. Accuracy 100.00%, coverage 5.65%
Iteration 3, removed 54 instances, instances left 1062
Overall stats at iteration 3. Accuracy 37.40%, error rate 0.06%,
not classified 62.54%
...
Final classifier
cl0:Class: 0, Dimensions:
{ origin: -3.713392508516529, bound: 5.528606636072313 }
{ origin: -5.251380236512823, bound: 11.371798908345687 }

cl1:Class: 0, Dimensions:
{ origin: -4.642997400816201, bound: 3.64812870795199 }
{ origin: 2.771105751241347, bound: 7.176336900227046 }

cl2:Class: 0, Dimensions:
{ origin: 0.0343352002486208, bound: 2.957006025118044 }
{ origin: -1.1089269345716088, bound: 1.5655938347945932 }
...
Stats on test data
Accuracy 95.26%, error rate 4.74%, not classified 0.00%

Total time: 4.895
```

Neural Network

```
Random seed is 1
Relation name TAO_grid
Biocomputing.Attribute name x
Biocomputing.Attribute name y
Biocomputing.Attribute name class
Accuracy 79.16%, coverage 61.90%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 50.00%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 75.91%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 83.80%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 87.63%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 85.28%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 88.04%, coverage 100.00%
 [BasicNetwork: { 2 7 7 7 1 }
Accuracy 88.52%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 89.34%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 89.28%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 89.34%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
Accuracy 89.87%, coverage 100.00%
[BasicNetwork: { 2 7 7 7 1 }
...
Classifier of iteration 0. Accuracy 89.87%, coverage 100.00%
Iteration 0, removed 1698 instances, instances left 0
Overall stats at iteration 0. Accuracy 89.87%, error rate
10.13%, not classified 0.00%

Final classifier
cl0:[BasicNetwork: { 2 7 7 7 1 }

Stats on test data
Accuracy 91.58%, error rate 8.42%, not classified 0.00%

Total time: 0.341
```

4.2 Visualisations

I could have visualised many different metrics; most are able to be graphed retrospectively, and therefore I focused on the evolving classification of points in the solution as a unique and useful tool to assess the performance of both algorithms. This is best seen in the running application as it is interactive. Final outputs for best-case are included below.

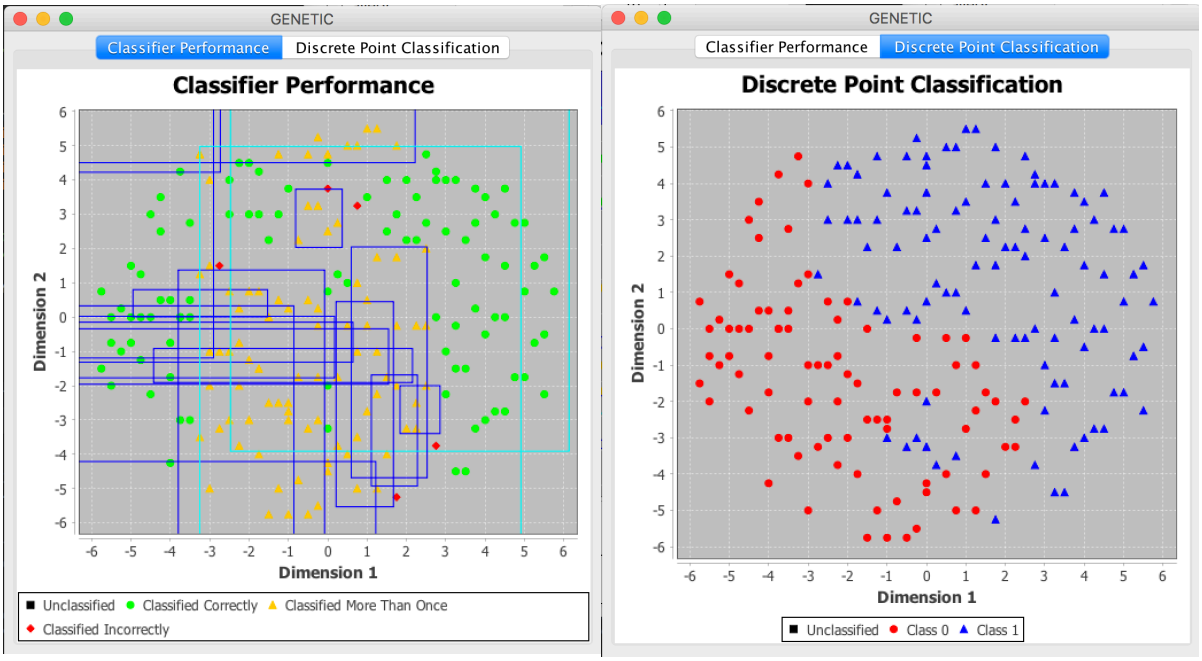

Fig. 4. The performance of the genetic algorithm visualised.

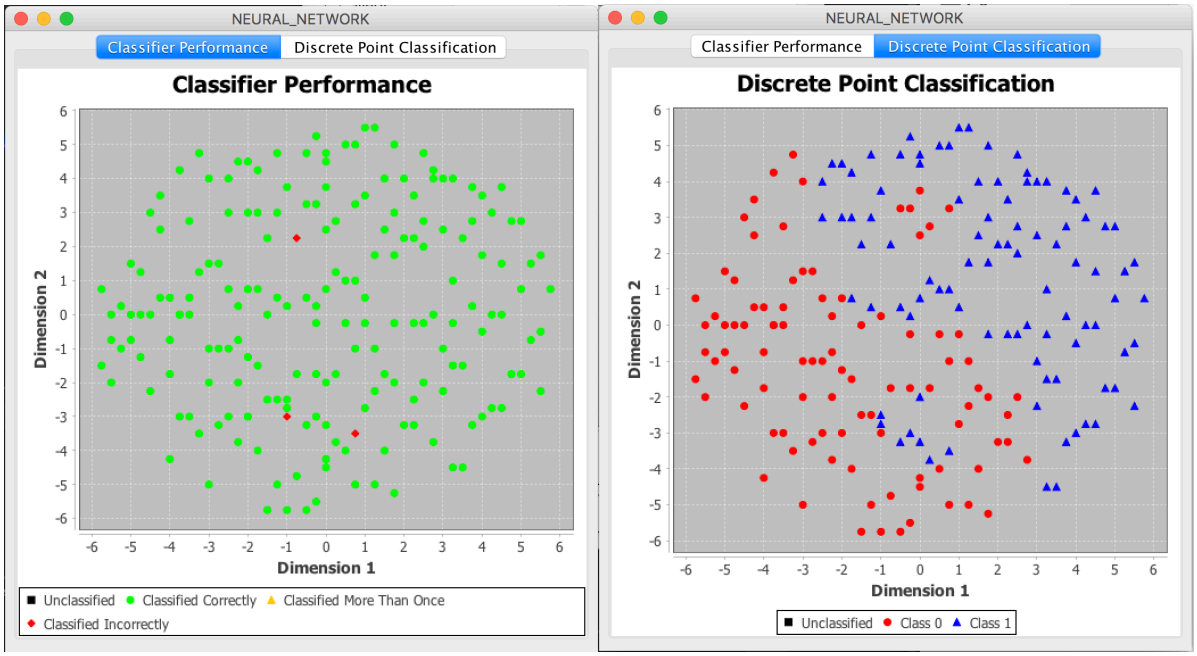

Fig. 5. The performance of the neural network visualised.