Report

Genetic Programming (GP)

Initially I planned to implement Cartesian Genetic Programming (CGP), a form of GP with its major difference lying in the knowledge representation, using an array of floats to represent a graph instead of a tree. In the end I found this unfavourable, as the libraries I needed to implement it had little documentation and a process which I found peculiar, making use of parameter files in very tightly-coupled classes, rendering incorporation into the framework via anything other than a master-slave relationship an impossibility. I first explored alternative libraries, of which there were very few reputable sources written in Java but failing this decided to revert to Koza's form of GP using the library EpochX (EpochX, 2012), an intuitive library with vast documentation.

EpochX is an intuitive library with vast documentation, providing extensive options for tweaking the learning process, an important prerequisite to experimentation. It also provides real-time statistics on how the run is progressing (EpochX, 2012) which would be useful in analysis of the performance.

GP is a competitive algorithm for classification problems, with (Espejo, Ventura, & Herrera, 2010) even finding that in 54.72% of comparisons studied it was the best performing method. Findings such as this were encouraging, but the deciding advantage was its interpretability, as I realised high understanding would be crucial to produce the best quality solution in a limited timeframe.

To implement the GP I have created two classes. TreeModel.java extends from EpochX's GPModel class and provides the getFitness function required to evaluate the suitability of a generated program. For each set of inputs the program is evaluated, and the result compared to the expected output. If this result is correct, the score is incremented to keep track of the program's accuracy. The fitness is then defined as the number of outputs incorrectly predicted, a value which is subsequently compared to the current best fitness to update the best program of the model before finally returning the fitness value.

The second class, ClassifierGP.java, is an extension of the framework's classifier and acts as an interface between the framework and EpochX. Here, the functions, inputs and outputs available to programs are defined, the model's parameters defined and finally the model is run to conduct the process of training. A function for classifying a single instance is provided, which does so according to the best program currently stored in the tree model, returning the predicted class or -1 if it is unable to make a prediction.

To run the program, three arguments must be supplied: the training dataset, the test dataset, and finally a string, which can either be "gp" to select the GP algorithm or anything else to select the ant miner.

Experimentation involved tweaking aspects of the program and recording the performance resulting from this. To obtain the results each test was carried out five times and the median recorded. Because of the nature of the implementation each run through took the number of iterations required, and this column instead details at which iteration the fitness ceased to improve and convergence was reached. As the learning process is managed within ClassifierGP.java, this is where the changes specified were largely made, particularly using its constants to alter important variables such as the number of generations and probability of certain stages of the pipeline.

Constant	Chosen value (when not a control variable)
Population Size	50
Number of Generations	50
Number of Elites	10
Crossover Probability	0.7
Mutation Probability	0.1

Control variable(s)	Iterations required	Accuracy (Training/Test)	Run-time	Performance analysis
Generation number 200, 100, 50 and 20	157 requiring the algorithm to be run twice to cover the data set, 57, 30 and 13	(75/79.5), (82.5/82.45), (85.3/85.8), (55/53.7)	13.781 3.466 2.411 0.865	Too many generations result in a long run-time, but this does not equate to a better accuracy. Conversely, too few generations provide insufficient opportunity for the solutions to evolve and can drastically impact accuracy.
No Elites	24	80.5	2	A lack of elites impacts the run-time of the algorithm, although it does so in fewer iterations.
High number of elites (20)	49	71.6	1.024	Elitism appears to increase the number of iterations required to converge at a result, and indicates that the policy requires more iterations to obtain a higher accuracy.
Lower Mutation (0.1) vs Crossover (0.7)	41	80.3/80	1.373	The chance of mutation should be lower than the chance of crossover, as whilst mutations can be useful for expanding the search space for solutions if it is too common it can hinder evolution and accuracy.
Higher Mutation (0.7) vs Crossover (0.1)	19	68.5/70	2.241	

Maximum depth has a close relationship to the bloating of the individuals, as a greater depth results on a lighter restriction on how much a program is allowed to expand. A lower depth will also require fewer iterations to complete the learning process. A higher depth requires a greater number of generations to construct a good solution, but when supplied with enough the best solution can often result in a higher accuracy. Having said this, an increase in depth does run the risk of cluttering solutions with unnecessary terms. This is a small problem but an excessively high depth can become hinder accuracy, performance and the readability of the classifiers produced.

A final area for experimentation with the algorithm was the choice of selector used, comparing linear selector with tournament selector. Linear selector is the faster algorithm, with tests finding that runtime could be improved to as little as 0.6 of a second when using it. Despite this, I would argue that tournament selector's performance is better due to the higher accuracy, finding it to be as much as 12.5% better at classifying the test data.

Overall, the performance of the GP as outlined here is impressive, having the ability to accurately classify a dataset in a small runtime, but as I will go on to discuss its achievements are largely overshadowed by those of the ant miner.

Ant Colony Optimisation (ACO)

The second algorithm selected in my proposal was followed up with an implementation similar to that which I had planned. A major reason for its selection is its drastic difference to GP in many areas. To elaborate, the implementation uses a very different to the function-representing trees of GP, opting instead for the construction of rule sets which outline a series of Boolean rules for asserting which class to predict for a given instance.

Bloating, a well-known problem with GPs, is not an issue with these rule sets due to the pruning mechanism utilised during an ant's construction of a rule, based on the use of a quality function to identify which terms within a rule can be removed without degrading coverage. This was another major factor which I took into consideration, along with the rapid discovery of good solutions that ACOs often boast as a result of their use of pheromones.

To implement an ACO method, I have extended the library Myra (Otero, 2016). I selected this library because of its high modularity, making it flexible to changes and extensions, and the support for continuous values through the class cAntMiner.java.

Integration into the framework did have its difficulties, as classifiers provided by the library manage the entire process of training and testing in a single class "run". As such, I began by editing the library, creating methods within the myra.Classifier class which deconstruct this functionality, creating methods for interpreting a data set, outputting the accuracy of the classifier and running the required set up for the training to work. I also tweaked the functionality to store key components of the process, such as the model, which acts as the representation, and make this available to external classes. Following this I implemented an evaluate function for evaluating a single Biocomputing.Instance object, to support AntClassifier's classifyInstance functionality. This also entailed complications, as the key model implementation myra.rule.RuleList, and the myra.rule.Rule objects it is made up of were not designed to classify individual instances. Because of this I have overridden classification in both classes to act on a single instance. Finally, as the library did not print out details about each iteration during the ant miner algorithm I have added this functionality myself within myra.rule.irl.SequentialCovering, printing the number of unclassified instances to track the progress of the algorithm.

Experimentation with the ant miner mainly took place in AntMiner.java, the superclass of the cAntMiner which I was using to run the experiment, and the class which controls most of the options such as colony size and stagnation.

I began by experimenting with colony size, a value set to 60 by default, achieving an accuracy of 91.58% on the test data in 1.343 seconds. Intriguingly, decreasing this drastically did not alter the result produced, but did improve the speed in which the algorithm was completed. This could be decreased down to as few as two ants without impacting the accuracy, but doing so resulted in a great degree of variation in the running time to find a solution, averaging at a worse performance overall than a colony with a few additional ants. I found that the optimal colony size was 6, resulting in an extremely fast running time of 0.376 seconds, achieving an accuracy of 91.58% in just 13 iterations, and have tweaked the library's code to incorporate this.

Another improvement to performance has been obtained by setting the stagnation value to 1, which means that the algorithm will terminate if no improvement in the global best is observed between iterations. Having said this, I have chosen to maintain the default value of 10 in this case, as I believe that the reason for the lack of an impact on the accuracy of the solution when decreasing it is due to a lack of stagnation when using the data set, and that another data set could result in a higher degree of stagnation that would be hindered by an aggressive policy towards it.

Interestingly, I have identified that whilst the GP implementation varies in the classifiers it produces after a successful evolution, the ant miner rarely does so. It almost exclusively results in the same solution which achieves an accuracy of 91.58% in 13 iterations. This is perhaps an indication that the ant miner is successful in finding the best solution that it can achieve within the parameters, whilst the GP algorithm regularly converges on a local minimum fitness. I believe that this is likely because the GP relies on chance via crossover and mutation to improve its fitness, whilst the ant miner is systematic in its rule discovery and pruning; a seemingly more suitable method as indicated by its regular achievement of 90% accuracy, in comparison to the 60-80% range achieved by the GP.

GP was relatively slow in comparison to the ant miner, taking typically between 1 and 3 seconds to complete, whilst the ant miner rarely took more than 1 second. The ant miner was unvarying in this respect, nearly always completing in a small range of timings around the 0.2 to 0.3 range, whilst the GP, even without altering parameters could run at very different speeds from test to test.

Another key benefit to the ant miner is the quality of the rules it produces, a result of the quality evaluation completed on every rule to prune it of unnecessary terms. In contrast, the GP does not include such a pruning mechanism, which can mean that the solutions found contain unnecessary terms and calculations during classification. As previously mentioned this is particularly a problem with higher maximum depths and can impact not only the interpretability of the solutions but also the performance of classification.

In conclusion, both algorithms are highly suited to the classification problem and demonstrate the exciting possibilities that could lay in store for the field of machine learning. In these experiments, with this dataset, the ant miner performed better than the GP method, and I believe that this is because it is a stronger algorithm, for the reasons outlined in this report.

References

EpochX. (2012). *EpochX Home Page*. Retrieved from EpochX Web site: http://www.epochx.org/

Espejo, P., Ventura, S., & Herrera, F. (2010). *A Survey on the Application of Genetic Programming to Classification.* IEEE. Retrieved from http://sci2s.ugr.es/sites/default/files/ficherosPublicaciones/1092_2010-Espejo-IEEETSMCC.pdf

Otero, F. (2016, February 13). A collection of ACO algorithms for the data mining classification task. Retrieved from https://github.com/febo/myra