Biocomputation Assignment

Luca Ricagni 16010178

1 INTRODUCTION

The task assigned was to evolve a system which, upon receiving a set of input variables, is able to classify them and give the correct predicted output. The fitness is therefore determined by how much of the data can be correctly classified by the system.
The approach taken in this assignment was a rule-based system. This means that each candidate is a series of rules which will be used to represent the data.
The programming language decided on to complete this task was MATLAB. MATLAB was chosen because of two main reasons. The first is its ability to manipulate matrices, which would be very useful when dealing with the offspring matrix. The second is the fact that MATLAB has inbuilt plot functions which, when used correctly, make it very easy to see how well the algorithm is working.

2 BACKGROUND RESEARCH

(Yoo, et al., 2016)
This paper discusses the use of data mining algorithms which can estimate groundwater pollution sensitivity patterns. In the described experiment, seven variables are used as input variables for the data mining algorithm: depth to water, net recharge, aquifer media, soil media, topography, vadose zone media and hydraulic conductivity. Four data mining algorithms were tested: artificial neural network, decision tree, case-based reasoning and multinomial logistic regression. The conclusion drawn by the paper was that the decision tree-based data mining and rule induction method was more accurate and consistent than the other methods.

(Weiss, et al., 2010)
This paper is about using rule-based data mining in order to improve yield in semiconductor manufacturing. Data is collected from the IBM 300mm fab, the machine on which the system is installed, which manufactures game chips and microprocessors. Tens of thousands of features are recorded and used in the system, this is far more than is required for this assignment, but it shows that rule-based data mining is a viable solution for large amounts of data.

(Farid, et al., 2016)
This report focusses on using an adaptive rule-based classifier for multi-class classification of biological data (Farid, et al., 2016). There are several problems with classifying biological data including: overfitting, noisy instances and class-imbalance data. The paper goes on to talk about the use of a random subspace to avoid overfitting, a boosting approach to classify noisy instances and making use of decision trees to deal with class-imbalance problems. The conclusion of the paper was that the performance of the system holds up well when compared with well known existing machine learning and data mining algorithms used on genomic data.

3 EXPERIMENTATION

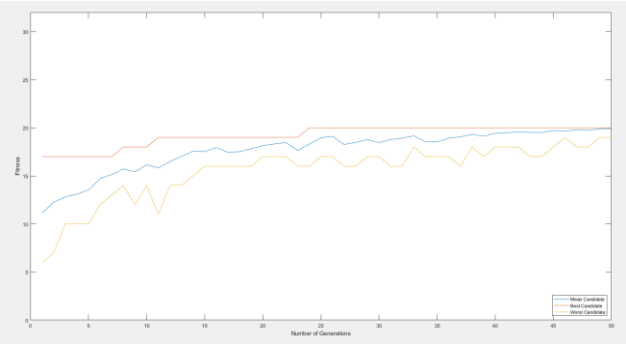
3.1 Data Set 1

The algorithm used in this task was a simple genetic algorithm which is designed to evolve over multiple generations to improve its fitness. At the start an initial population of rules is generated, each rule being five input bits and one output bit. As such each generation is represented as a matrix NxL, where N is the number of candidates and L is the length of said candidates. Each bit within this matrix is either a 0, a 1 or a 2, where 2 can mean either 1 or 2. This is the implementation of ‘fuzzy logic’ which helps to minimise the number of rules needed because otherwise you would need one rule for every data input, which slightly defeats the point of representing the data input. In addition, there is a partner array of size N which holds the corresponding fitness values for each candidate. In every iteration of the loop, i.e. every generation, these two arrays are modified using various functions such as:

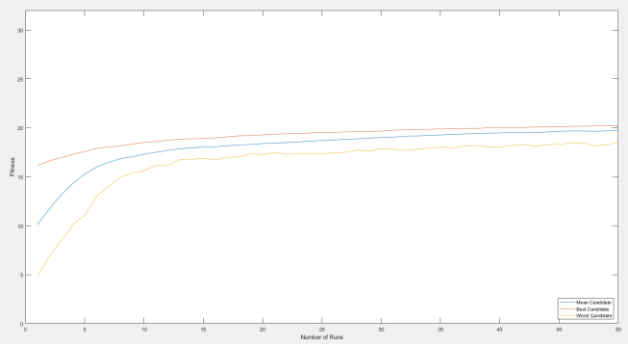
SinglePointCrossover, which flips the tails of pairs of candidates at random points, BitewiseMutation, which checks to see if each bit in each candidate mutates, there is then a further 50/50 chance to determine which of the two possible other states the bit flips to. The other function which modifies the population matrix is SelectOffspring which simply randomly selects two candidates, compares their fitness, and puts the one with higher fitness into the next generation.

As this is a fairly simple implementation of a genetic algorithm, the number of parameters needed is relatively few, the necessary ones are: the number of candidates in each generation, the length of each candidate, the mutation rate (given as a percentage) and the number of generations to run the algorithm over.

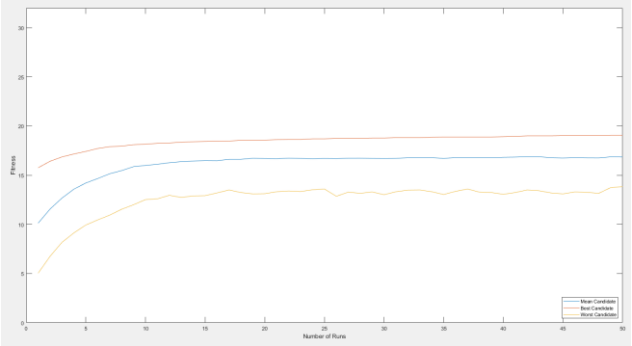
The fitness calculation is slightly more complex, it is a function which takes as input the length of each candidate, the current candidate and the input and output of the provided data. The function then iterates through each rule, namely each set of six bits (five input and one output) and checks to see if they match with the input data. If both the input and the output match, then fitness is awarded.



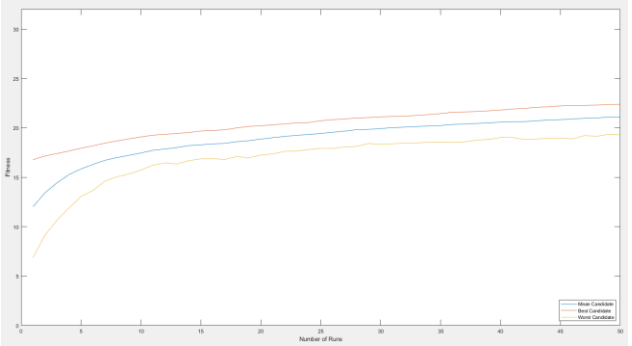
The above graph shows one run of the algorithm with the following parameters: population of 50, candidate length of 60, mutation rate of 1% and 50 generations.



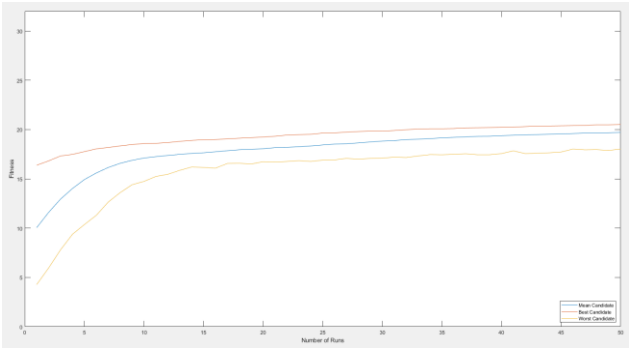
The above graph shows the algorithm with the same parameters over 50 runs, the graph shows smooth line which is an indicator that a consistent level of fitness is attained which demonstrations that the algorithm is reliable.



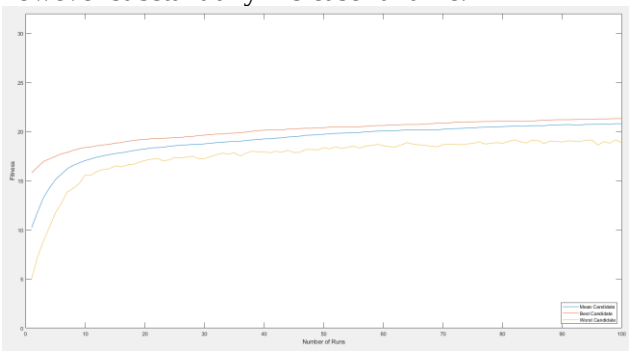
The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 60, mutation rate of 5% and 50 generations. The graph seems to suggest that, while the maximum fitness attained remains roughly the same, the average and lowest fitness drop quite a lot as well as taking more generations to reach the highest fitness.



The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 120, mutation rate of 1% and 50 generations. The maximum attained fitness for increasing the candidate length increases, showing a possible connection between the two, however this is because increasing candidate length increases the number of rules, which would obviously increase attainable fitness.



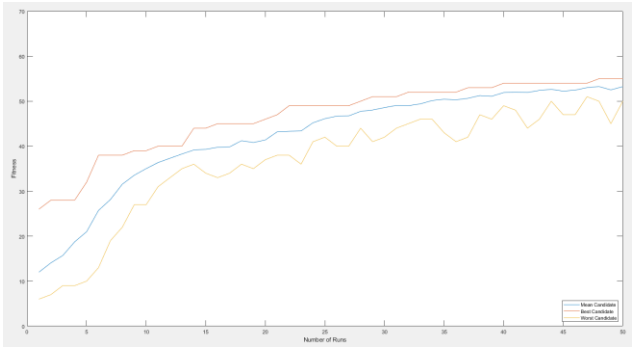
The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 100, candidate length of 60, mutation rate of 1% and 50 generations. This did not increase the maximum attained fitness or lower the number of generations needed to reach maximum fitness, it did however substantially increase runtime.



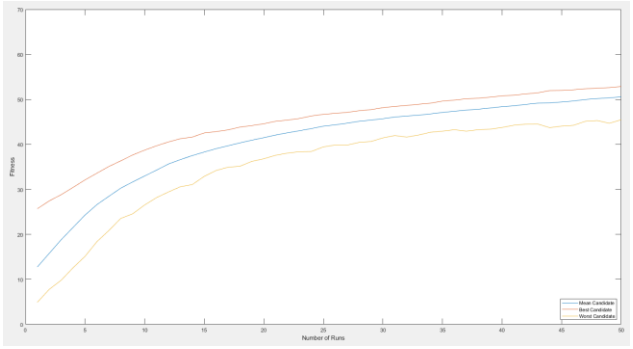
The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 60, mutation rate of 1% and 100 generations. This change had minimal effect on the maximum attained fitness or the number of generations taken to reach it.

3.2 Data Set 2

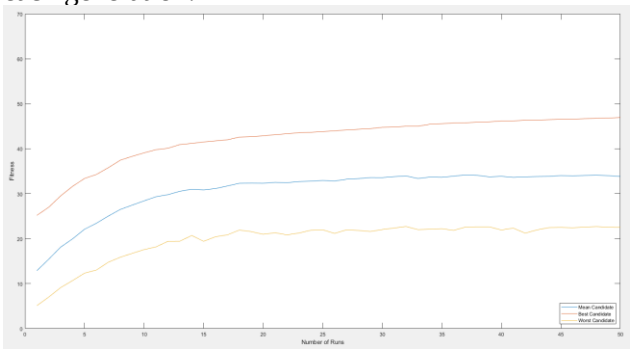
This data set is very similar to data set 1 in that it is just a set of 0s and 1s which make up the input and a single 0 or 1 which makes up the output. The only difference between the two data sets is the length of the data (7 instead of 5) and the amount of data (64 instead of 32). As a consequnce of this getting the algorithm to work with data set two was fairly straightforward as the functions all are soft coded to find things like the length of the data and the amount of it. The only thing that is not soft coded is the variable ‘dataName’ which dictates which data file is loaded into the program.



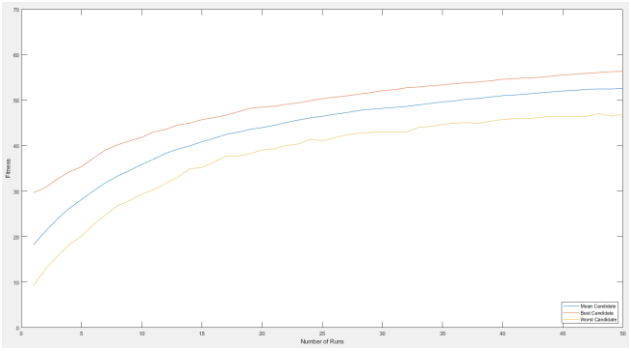
The above graph shows 1 run of the algorithm with the following parameters: population of 50, candidate length of 80, mutation rate of 1% and 50 generations. The algorithm reaches a fairly high fitness.



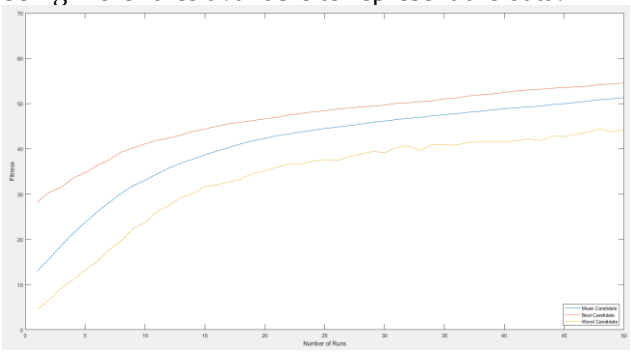
The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 80, mutation rate of 1% and 50 generations. Again, the smooth curves are evidence of consistent fitness attained on each generation.



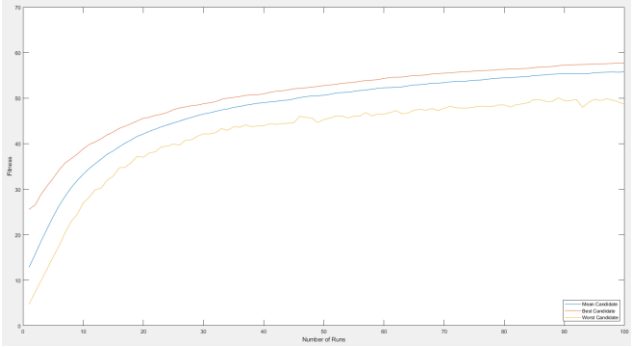
The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 80, mutation rate of 5% and 50 generations. Similar to its effect on data set 1, increasing the mutation rate seems not to affect the maximum fitness or the number of generations taken to reach it much, it does however quite drastically lower the mean and the lowest fitness.



The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 160, mutation rate of 1% and 50 generations. This increases the overall fitness attained but again this is due to there being more rules available to represent the data.



The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 100, candidate length of 80, mutation rate of 1% and 50 generations. Much like with data set 1, increasing the population size seems to do nothing other than increase the program runtime.



The above graph shows the average over 50 runs of the algorithm with the following parameters: population of 50, candidate length of 80, mutation rate of 1% and 100 generations. Changing this variable beyond a certain point has very little desirable effect.

3.3 Data Set 3 (and UCI, etc data)

Much like data sets 1 and 2, modifying the program to work with data set 3 should in theory be relatively easy as the biggest difference is the sheer amount of data (2000 instead of 32 or 64). However, the biggest challenge for data set 3 is that instead of being 0s and 1s, the input data values are all six-bit floating point numbers, which meant that importing the data into MATLAB and then converting it into a useable format proved very challenging and was ultimately unable to be solved in the allocated time. In addition, altering the functions used earlier, particularly the fitness function, introduces a whole new set of issues such as having to compare floats instead of integers. Also generating the initial population would be very arduous because, out of every 8-bit rule, the first seven would have to be randomly generated six-bit floats (the input) and the eighth bit (the output) would be a 0 or a 1.

Owing to the size of data set 3, the idea is to use sections of the data (quarters for example) to train the system and then test the system on the remaining data to see how well it performs. In order to improve the accuracy and validity of the tests make sure to run the system using different sections of the data to train the system.

4 CONCLUSIONS

If this task was to be repeated there are a few things that would be done differently. The first is that more time would be set aside to complete data set 3 because that is where the main meat of the assignment is. Since data set 3 was unable to be imported correctly it meant that, as mentioned before, using some of the data to train the system and then testing the system on the rest of the data was unable to be attempted. The second would be to write the function to calculate the fitness values in such a way that it calculated the fitness of one candidate at a time, rather than all of them at once which was the original method of implementation. A great deal of time was lost due to not simply rewriting the fitness function earlier and instead attempting to make the rest of the code work around it, often to extremely inefficient results.

Overall this task was very enjoyable and interesting, and a lot was learned, both terms of how to implement a genetic algorithm but also just how

powerful they can be and the wide variety of problems they can solve just by changing the way in which fitness is calculated.

REFERENCES

Farid, D. M., Al-Mamum, M. A., Manderick, B. & Nowe, A., 2016. An adaptive rule-based classifier for mining big biological data. *Expert Systems with Applications,* Volume 64, pp. 305-316.

Weiss, S. M. et al., 2010. Rule-based data mining for yield improvement in semiconductor manufacturing. *Applied Intelligence,* 33(3), pp. 318-329.

Yoo, K. et al., 2016. Decision tree-based data mining and rule induction for identifying hydrogeological parameters that influence groundwater pollution sensitivity. *Journal of CLeaner Production,* Volume 122, pp. 277-286.

Links to all the code:
AllCodeCombined:
https://pastebin.com/TafKzbAy

Links to individual functions:

Main: https://pastebin.com/YW4snMct
CalculateFitness: https://pastebin.com/tm0x1wB4
SinglePointCrossover:
https://pastebin.com/ZaRqqnwV
BitwiseMutation:
https://pastebin.com/RW1UHQNb
EvaluateSolutions:
https://pastebin.com/HFnr7qG5
ImportData: https://pastebin.com/bfZCV1Zn
SelectOffspring: https://pastebin.com/CaTXskPv
DataSet1: https://pastebin.com/5BMymPG6
DataSet2: https://pastebin.com/HmhQqxwd