Discovery of Hidden Data Structures Using Genetic Algorithms and Data Mining Techniques

Simon Llewellyn 04971824

1 INTRODUCTION

Research and pattern recognition can be a long and tedious task. If a person is given a list of 10 pieces of data and was asked to study the list to find the pattern that links each item together. It is likely that person will take a rather long time to come up with a single solution. If that solution was then tested against another 10 lines it is possible that the person may have found the under lying rule between each piece of data. Though what would happen if that same person was then given 50 new pieces of data that have a different structure. Will the person still be able to find the underlying rules to the data. It is unlikely. This aim of this assignment is to show that with the use of known machine learning algorithm inspired by nature it is can be possible to compute a set of rules that can accurately as possible classify the existing data which in turn can be used to also classify and predict possible future data with ease.

2 BACKGROUND RESEARCH

Martin Brown (2012) states "data mining is about processing data and identifying patterns and trends in that information so that you can decide or judge." In 2013 there was estimated 4.4 zettabytes (4.4 trillion gigabytes) of data in the world (EMC, 2014). With all this information available would it not be beneficial to allow a computer to process this information to find patterns and correlations that may help better our understanding of the world.

Data mining techniques have been used in many field of subject aid with the discovery of new trends and ideas. In the medical community it has been used to help diagnose many medical conditions. As explained by Austin *et al* (2013) "Modern data-mining and machine-learning methods offer advantages for predicting and classifying heart failure (HF) patients according to disease subtype". It has also been used to help identify patterns within energy consumption in China to help with load distribution and sustainability (Zhou, K. Yang, C. Shen, J. 2017).

There have been failures with data mining as well. In 20098 Google launched a service called Google Flu Trends (Ginsberg *et al*. 2009), where Google used data from search patterns to try and predict outbreaks of both the flu and dengue fever with the aim to produce high accuracy prediction greater than the Centers for Disease Control and Prevention (CDC). While the concept is sound Google had to retire the service due to inaccurate predictions where they were predicting much higher rates of infected people than the CDC. Google's data was proven to be inaccurate as stated by Lazer, *et al*. 2014. Lazer *et al* goes on to say that this was due to the fact Google was reliant on the users input; where many people are unsure is they had the flu or what it even is were searching for the term and thus giving erroneous data, which google compiled into its results.

These case show that the use of data mining can be used to do incredible things but the whole thing will fall about if the quality of the data is poor.

Many methods can be used for data mining but like the work by (Sharma, P. Saroj. 2015) this paper will focus on the use of a Genetic Algorithm (GA).
A GA could be described as the superstar of the machine learning toolkit. It is the most common, extremely powerful, and can be easily implemented.

By taking direct cues from nature, a GA allows the creation of a fittest solution from the use of the following methods of evolution:

- Offspring: creation of a high population from the combination of parents.
- Variant or mutation: occur across offspring to help survive in the environment.

- Selection: if the offspring are strong they will survive to become parents to pass.

These are the three stages of evolution that the GA is based upon.

3 EXPERIMENTATION

3.1 Setting up the Algorithm

Genes

The main feature of the GA is the gene. This is generally a sequence of data that represent the problem/solution. The goal of this assignment is to find a set of rules to classify the data given. This means the gene will need to represent the total amount of rules including their output. If the GA is trying to find 10 rules where each set of conditions is 5bits long and a result a 1bit the total gene size will be:

- (5 + 1) * 10 = 60bits

Initiate

The beauty of a GA is that it is never working a single solution, instead it is working on a plethora of potential solutions. These solution are then combined to hopefully create better solutions. To start, the algorithms initial population will need to be created using some method that randomly generate a gene using the same data type that the GA is trying to categorise a binary string the same size the the formula give above. This is repeat for n amount of times where n is the size of the population.

Tournament Selection

The aim of tournament selection is to gradually remove the weakest solution from the population. This is important as without selection the solution will never be able to become stronger. However, it is also important to keep a few of the lower scoring solution as not back yourself into a 'genetic corner' where the fitness may plateau early, and no fitter solutions are possible.

The selection operates by randomly choosing two members of the population and comparing the fitness. The fittest of the two is copied into the offspring population this need to be donw so the new population (offspring) is equal to that of the original population. The old population is now discarded. By randomly picking two elements it ensures that weaker solutions are selected and keeps the gene pool open while also keeping an element of randomness.

Crossover

Standard crossover function is used by picking two elements in order and using a random number generator (RNG) to pick a point on the gene where the two genes can swap their tails. This helps mix up the population and hopefully gain a higher fitness. If the first half of one gene scores a high fitness while the end half of another gene also scores high by performing crossover you will be left with one very fit gene and another very weak gene, most likely the week gene will be filter out come the next selection process.

Mutation

Mutation applies subtle changes to the population, without it the solution will converge to the fittest solution there and replicated over and over again. This is due to the tournament selection weeding out the weakest solutions from the population the later generations will be repeatedly swapping their tails while never changing any of the values to find new solutions. To resolve this an element of change is needed to mix up the gene and help push towards the goal state. The frequency of the mutation depends on the scope of the problem. There will be more about this later.

Evaluate

Simply scores the population to find the fittest solution out of all the possible solution. This solution is then stored and compared against future generations. If a fitter solution appears the old solution is discarded and the new one tasks the place.

These six steps will be used to across all the whole assignment and used in each of the test however, there may be a few changes on how some are implemented these changes will be explain when they arise.

3.2 Data Set 1

Data representation

Data set 1 contains 32 rows of bits strings. Each row is six bits long. The first five of these bits are the conditions while the last bit represents the type. For example, if these each of the conditions represents a result of medical tests where 1 being positive and 0

being negative the combinations of conditions will give the output of the last bit (type); this again is represented either positive or negative using a 1 or 0. As the input data is binary we need only populate the initial population with genes complied from a random selection of 0 and 1s.

Fitness

The fitness of a solution is measured by comparing the conditions within each row of the data set against the rules generated in the possible solution (see **Appendix 2**).

The idea is that you pass the data set over the rules, once there is a match the quality of the rule will be scored and then breaks out of the loop. By using a break the algorithm ceases to compare that row of data against any other rule in the solution.

By using this strategy, it ensures that the fittest solution will contain an ordered list of rules that can be easily followed.

Generalisation

In its current state, when running the algorithm the fitness will always be equal to the number of rules it is trying to find. To combat this, the system needs to find a way to generalise the rules so that one rule may be equal to more than one line in the data set.

For instance, if data contains entries where the first and last condition contains a 1 and the type is always equal to 1. The system needs a way to remember this rule so in the future it can discard any need to check the conditions in between.

To achieve this a third character is required so when the system sees it it knows that it safely ignore the value and allow it to continue. In this case the # character was use.

To implement this a change is needed to the section where the conditions of both the rule and data are compared. In the previously the code would be similar to:

- IF rule[j].condition == data[i].condition

This code should now be changed to include a function that checks each induvial condition (see **Appendix 3**).

With the introduction of wildcards, the system now can classify multiple lines of data with the use of one rule.

The following two rules are considered to the be equal to the rule while still not being equal.

- Rule : 1###1 = 1
- Data 1: 10001 = 1
- Data 2: 11101 = 1

Results

When running the algorithm across data set 1 the results are somewhat disappointing, it shows the data contains no real structure or pattern. The highest fitness the system can achieve is equal to:

- (Number of rules – 1) + 17

Upon studying the data, it is apparent that the majority rows have a type of 0. When running the algorithm to find a single rule it returns:

- ##### = 0

The above rule has a fitness of 17 which mean it can successfully classify 17 lines of data out of a possible 32. Any other rules added to this to classify the rest of the data appear to only be direct copy of the data itself.

- Rule 1: 11000 = 1
- Rule 2: 11101 = 1
- Rule 3: 10111 = 1
- Rule 4: ##### = 0

Parameter Testing

Using the knowledge of this data set, it is possible to test out different parameters to find optimal search parameters.

The fitness for the following series of results are taken from the average fitness of each generation over 100 runs of the algorithm.

Each GA is looking for 10 rules to classify data set 1. The max fitness for 10 rules is 26.

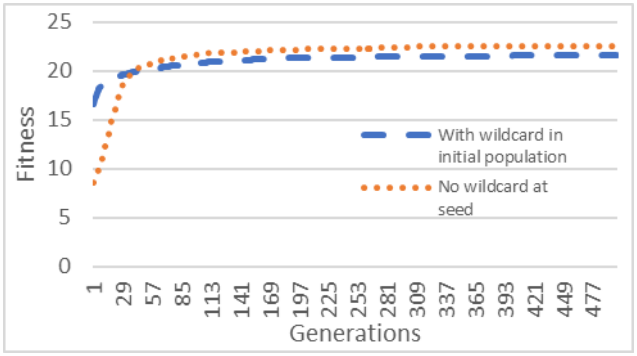

Figure 1: Performance when comparing wildcard in initial population data set 1.

Figure 1 shows that while using the wildcard as part of the initial seed population, the fitness of the

results starts high but them seem to plateau and stagnate before even reaching 100 generations. While looking at the population seeded without the wildcard it appears to match that the other results fitness in a relatively short time and continues to grow. Overall the average fitness score is much higher. This could be down the ordered list setting of the rules. Once a rule in a set has been established that contains nothing but wildcards this will then start to multiply out and be chosen as fitter candidates than other and will be harder to remove.

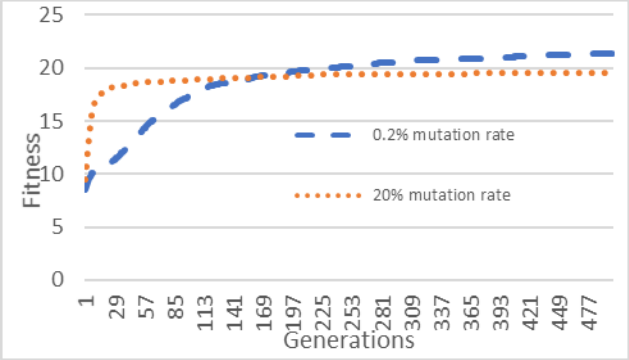

Figure 2: High mutation vs low mutation rate on data set 1.

Figure 2 shows that with a high mutation rate the fitness of the solutions climb rapidly but almost immediately stops a fair distance away for the best solution and stay around there with very little movement. The for this is while an optimum solution maybe within reach, it could be that its current gene only requires a stuble change, it is more likely to be mutated away than towards the optimum.
E.g. If the mutation was set to 33% chance. (1/3)

- Goal: 011011
- Current: 010011
- Outcome: 011010

The example shows that while the correct mutation was achieved and the gene was moved towards the goal the odds that another mutation would take place were too high and did take place thus carried the solution away from the goal.
On the other hand, the smaller mutation rate appears to creep up slowly to high fitness and carries on increasing surpassing the average fitness.
The main issue with this is the time it took to took surpass the high mutation rate almost 200 generation. On large problems this will be a high drain of resources.

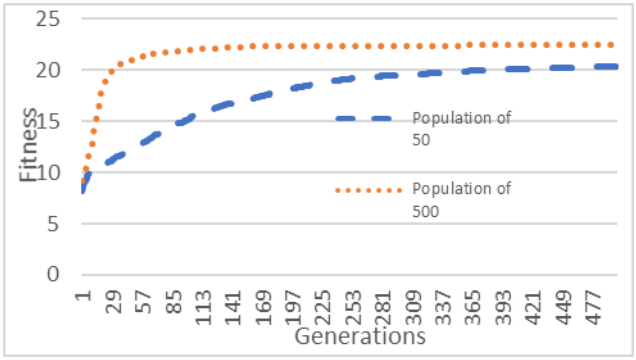

Figure 3: High vs low population on data set 1.

Figure 3 highlights the need of a decent size population. While the concept of a genetic algorithm is based upon the stochastic nature of genetics; where a random chance dictates a change which when pitted against others may prove to be more dominant. It makes sense that starting with a large population will increase the chance of finding a strong solution very early on whilst a small population may struggle to even achieve the same level of success throughout its life. Similar to mutation rate, population size accuracy requires a trade-off for computation time.

For the rest of this paper, the results of the other tests will be using what was discovered through trial and error to be their optimum values of mutation rate and population size.

3.2 Data Set 2

Differences

Data set 2 was similar in nature to that of data set 1 but with a slight change. Instead of having five conditions to one type this set of data had six conditions. This means the calculation for the gene has changed to:

- (6 + 1) * 10

Apart from gene size, there were no further changes required to the GA used for process data set 2.

Structure

As the data set has an extra condition this double to the size of the input data from to 64 lines; one for each permutation between 000000 & 111111.
When the GA processed the data with the use of the wildcard it was able to classify all 64 rows within in 10 rules without any issues.
After running 100 GA's, 66 solutions correctly classified all 64 items in the data set and return a best

set of rules as:

- Rule 1: 110### = 0
- Rule 2: 01##0# = 0
- Rule 3: 10#0## = 0
- Rule 4: 11000# = 0
- Rule 5: 00###0 = 0
- Rule 6: ###### = 1
- Rule 7: 1#1011 = 1
- Rule 8: #01000 = 1
- Rule 9: 100011 = #
- Rule 10: 010#1# = 1

As mentioned previously, the scoring system work in a way that the rules are presented in the form of an ordered list and should be read from first to last.
With this is in mind the above set of rules contains 4 superfluous rules. Rules 7-10 are never seen by the data set due to rule 6. Once the data has passed through rules 1-5 and directly associated all data sets ending with a 0 anything else must be a 1. This means that the 64 pieces of can be classified buy a set of 6 rule. However, this is not the least amount of rule possible for this data set.

After 100 GA's configured to search for 5 rules, 21 correctly identified all 64 items in the data set. This is approximately a 1 in 5 chance of finding the below set of rules:

- Rule 1: 00###0 = 0
- Rule 2: 01##0# = 0
- Rule 3: 110### = 0
- Rule 4: 10#0## = 0
- Rule 5: ###### = 1

Though upon repeat test there is another set of 5 rules:

- Rule 1: 111### = 1
- Rule 2: 00###1 = 1
- Rule 3: 01##1# = 1
- Rule 4: 10#1## = 1
- Rule 5: ###### = 0

As you can see these set of rules are very similar in structure. The first 4 rules classify the conditions to return one specific type while rule 5 blanket cover the remaining rule as the other type.

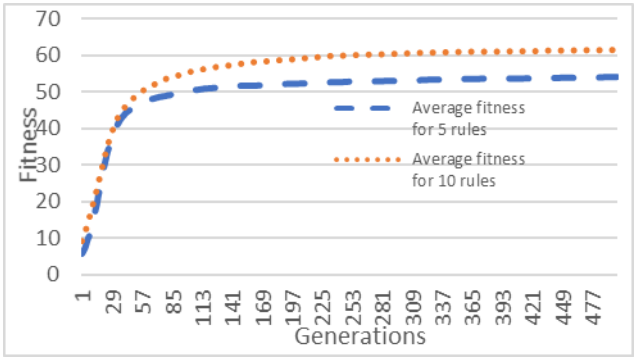

Figure 4: Average fitness of 10 and 5 rules data set 2.

Due to the low success rate of the '5 rule' GA the '10 rule' GA will consistently find an answer to the solution, but, will never be able to guarantee the most efficient one. **Figure 4** displays the trajectory of the '5 rule' line as steadily climbs. A few changes to the parameters; perhaps a lower mutation rate and high population, could make it more efficient. With the current data set the difference between calculating 10 rules as 5 is nothing but it is important to keep I mind that on larger data set this could mean the all the difference.

3.3 Data Set 3

Data Representation

Unlike data set 1 and 2 where the data was represented using binary (or trinary once you include the use of wildcards), this set uses floating-point number:

- 0.6875, 0.658, 0.3615 = 1

To compensate for this, a new way to represent this data within the gene is require.
Currently the formula that decides the length of gene operates on the assumption that each condition/bit of the gene can be represented by using 1 of 3 different characters. Now, each bit can be an infinite amount of values between 0.0 & 1.0 so a way of encapsulating these values is required.
The solution; use ranges.
Ranges allow for the floating-point numbers to be measured with ease. A number can be considered to be a machine if it falls between two values. To follow on from the previous example of floating-point conditions:

- (0.2 – 0.9), (0.6 – 0.7), (0.1 – 0.4) = 1

This will be a valid rule that matches the previous example. By doubling the size of the conditions in the gene it is possible to allow the gene to find suitable ranges which can be decided by the GA.

As data set 3 is contains six floating point conditions and one binary bit for the type the gene length will follow the rule:

- ((6*2) + 1) * number of rules

Changes to the Algorithm

After changing the formula to set the size of the gene length the function that populates the gene will also need to be changed. Instead of populating it with binary characters it will need to be populated with random floating-point numbers between 0.0 & 1.0 (using a modular arithmetic it is possible to keep the type as being represented as binary character). Other major changes to the algorithm involve updating the mutation and fitness scoring functions

Mutation

As it is now possible to represent any number between 0.0 & 1.0 the use of a wildcard is now redundant, but mutation is still required to avoid stagnation of the solutions.

By keeping the method to measure the percentage of mutation per bit, once a mutation is required instead of changing the value to one of the other characters (as per the binary configuration) the values is +/- by a set amount. The action of performing either a + or - is decided by a RNG. By using this the job of the wildcard is taken over as the over time where a wildcard may be needed to the ranges can be altered to represent any number between 0.0 and 1.0.

It is also beneficial to add a condition to the code that sets an upper and lower limit so that once a number is either > than 1.0 it is rounded to 1.0, likewise if a number becomes < 0.0 it is rounded to 0.0.

Fitness Scoring

The other change is how the fitness needs to be scored. As the match_cond function in the pervious GA compared the value to either a 0, 1, or #, this method requires a subtle change to see if the condition in the data lands between two numbers (see **Appendix 4**).

The new match condition allows the two numbers in the gene to dictate the range to be in either (min, max) or (max, min). The rest of the matching algorithm stays the same to create an ordered list for the rule. It is worth noting that a counter is required compensate for the extra conditions in the gene.

Training and Testing

Data set 3 contains 2000 rows of data and to test the effectiveness of the GA this data has been split into two set of data; a training set and test set.

The split of these two set is decided by a RNG (see **Appendix 5**). The concept is by using a training set of known data to create a solution. It is then possible to test the validity of the solution by comparing it against remaining know data in the test set. If this solution garners a high percent when classifying the data, it stands to reason that it will perform equally well for any future data.

Results

When dealing with high amount of data the likelihood of being able to match 100% of the data is rather lower unless devoting a high level of resource and computational time to the problem to find the perfect parameters. Also, when using such a high level of data it is possible that there may be one or two erroneous pieces of data that may skew the results. However, it is possible to obtain a high level of certainty to the set of rules it may produce.

In the case of data set 3 it was possible to find a solution that returned 100% match rate after testing out different parameters. With a mutation size of 0.5 the algorithm was able to correctly classify 100% of both the training and test data (see **Appendix 6**).

After running the GA multiple times across the data set it was found that when the GA score >95% on the training set the rules would match that of the set that scored 100%.

As the rules created by the GA contained ranges for each condition of the data (see **Appendix 7 and 8)** its preferable to convert these into a more readable format. If it is taken that that "< 0.5 = 0" and ">= 0.5 = 1" it is possible to convert these ranges into bit strings like that of the previous two data sets (see **Appendix 9**).

- Rule 1: 00###0 = 0
- Rule 2: 1#0##0 = 0
- Rule 3: 0##0#1 = 0
- Rule 4: 1###01 = 0
- Rule 5: ###### = 1

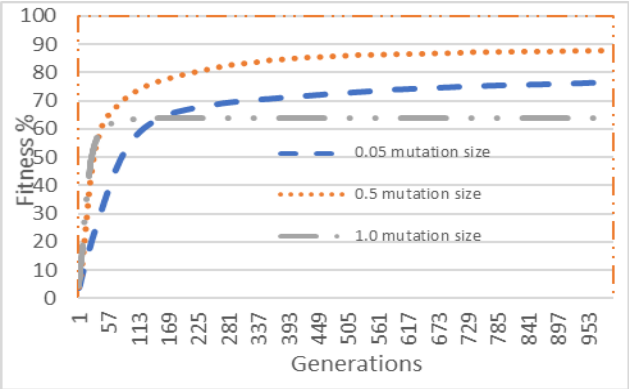

Figure 5: Average % fitness on different mutation size

A floating-point number GA introduces a new parameter; the size values are +/- by during the mutation section. **Figure 5** shows the difference between difference in mutation size.

Due to the nature of the how the training and test data divided, this graph displays the percent the fittest solution matches against the training set.

4 CONCLUSIONS

While the use of a GA is extremely powerful at establishing patterns and rules hidden within data it feels that there may be stronger more reliable method of data mining algorithms. The results in this paper have shown that as a method for finding a solution to a problem is possible, though it is apparent that it may suffer when scaled.

The result for the experiment in on data set 1 show that unless a certain range for the population size and mutation rate it entered the chance of finding the optimum solution in a reasonable time is severely reduced, and while it is possible to find the most efficient set of rules e.g. data set 2 and 3 can be reduced to 5 rules, it is not a guaranteed that the GA will be able to find them on the first time.

It may be possible that expansion of the GA could extend to using another GA to decide on the specific values contains to archive the optimum population size, mutation rate and size.

REFERENCES

Austin, P. C., Tu, J. V., Ho, J. E., Levy, D. and Lee, D. S. (2013) Using methods from the data-mining and machine-learning literature for disease classification and prediction: a case study examining classification of heart failure subtypes. Journal of Clinical Epidemiology[online]. 66(4), pp. 398-407.

Brown, M. (2012) Data mining techniques. *IBM developerWorks.* 11 December. Available from: https://www.ibm.com/developerworks/library/ba-data-mining-techniques/index.html [Accessed 28 November 2017]

Ginsberg, J., Mohebbi, M.H., Patel, R.S. and Brammer, L. (2009) Detecting influenza epidemics using search engine query data. Nature (London) [online]. 457 (7232), pp.1012; 1012.

Lazer, D., Kennedy, R., King, G. and Vespignani, A. (2014) The Parable of Google Flu: Traps in Big Data Analysis. Science [online]. 343 (6176), pp.1203-1205.

Sharma, P. and Saroj, (2015) Discovery of Classification Rules Using Distributed Genetic Algorithm. Procedia Computer Science [online]. 46(Supplement C), pp. 276-284.

Zhou, K., Yang, C. and Shen, J. (2017) Discovering residential electricity consumption patterns through smart-meter data mining: A case study from China. Utilities Policy [online]. 44(Supplement C), pp. 73-84.

EMC. (2014) The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. *Executive Summary Data Growth, Business Opportunities, and the IT Imperatives* April 2014. Available from: https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm [Accessed on 28 November 2017]

APPENDIX

1) **Source code for project along with all result files used to create the graphs can be found:**
   https://github.com/UWE-SimonLlewellyn/BioAssignment

2) **Pseudocode for fitness scoring in data set 1 and 2**

```
LOOP I < rows in data set
    LOOP j < number of rules in solution
        IF rule[j].conditions == data[i].conditions
            IF rule[j].type == data[i].type
                Fitness++
            END IF
            Break loop j
        END IF
    END LOOP
END LOOP
```

3) **Pseudocode for wildcard match condition in data set 1 and 2**

```
Boolean match_con(rule.con, data.con) =
LOOP i  <  number of conditions in rule
    IF (rule.con[i] != data.con[i]) & (rule.con[i] != #)
        return false
    END IF
    return true
END LOOP
```

4) **Pseudocode for division of training and test sets**

```
LOOP i < data set 3
    r = random number between 0.0 and 1.0
    IF (r < 0.75)
        Add data[i] to training set
    ELSE
        Add data[i] to test set
    END IF
 END LOOP
```

5) **Pseudocode for checking is number from data lies between the ranges within the rule.**

```
a =  range1,  b = range2, c =  data
LOOP  i < length of rule;
    IF  a[i]  >  b[i]
        IF c[i]  >  a[i]  OR  b[i] > c[i]
            Return false
        END IF
    ELSE
        IF  c[i]  <  a[i]  OR  b[i]  <  c[i]
            Return false
```

```
            END IF
        END IF
        Return true
    END LOOP
```

6) Result for 5 rules matching 100% of data set 3

Trained using 1501 sets of data
Best's fitness on the training set 1501
Rule 1: (0.0 , 0.5) (0.5 , 0.0) (1.0 , 0.0) (0.0 , 1.0) (0.0 , 1.0) (0.0 , 0.5) = 0
Rule 2: (0.5 , 1.0) (1.0 , 0.0) (0.5 , 0.0) (1.0 , 0.0) (0.0 , 1.0) (0.0 , 0.5) = 0
Rule 3: (0.0 , 0.5) (1.0 , 0.0) (1.0 , 0.0) (0.5 , 0.0) (0.0 , 1.0) (0.5 , 1.0) = 0
Rule 4: (1.0 , 0.5) (1.0 , 0.0) (1.0 , 0.0) (0.0 , 1.0) (0.5 , 0.0) (0.5 , 1.0) = 0
Rule 5: (1.0 , 0.0) (0.0 , 1.0) (0.0 , 1.0) (1.0 , 0.0) (0.0 , 1.0) (0.0 , 1.0) = 1

Rule 1: 00###0 = 0
Rule 2: 1#0##0 = 0
Rule 3: 0##0#1 = 0
Rule 4: 1###01 = 0
Rule 5: ###### = 1
Tested using 499 pieces of data
Best's fitness over test set 499
Equals 100.00% accuracy on training set
Equals 100.00% accuracy on test set

7) Result for 5 rules in data set 3 as floating-point ranges showing one version the minimum rule set.

Trained using 1533 sets of data
Best's fitness on the training set 1514
Rule 1: (0.4999999 , 0.0) (0.0 , 1.0) (1.0 , 0.0) (0.0 , 0.49684206) (0.0 , 1.0) (0.5 , 1.0) = 0
Rule 2: (1.0 , 0.47470355) (0.0 , 1.0) (0.0 , 0.5038541) (1.0 , 0.0) (1.0 , 0.0) (0.0 , 0.4999999) = 0
Rule 3: (0.4999999 , 0.0) (0.0 , 0.4847692) (0.0 , 1.0) (0.0 , 1.0) (0.0 , 1.0) (0.4999999 , 0.0) = 0
Rule 4: (0.45344314 , 1.0) (1.0 , 0.0) (1.0 , 0.0) (1.0 , 0.0) (0.51030993 , 0.0) (1.0 , 0.4756125) = 0
Rule 5: (1.0 , 0.0) (1.0 , 0.0) (1.0 , 0.0) (0.0 , 1.0) (1.0 , 0.0) (1.0 , 0.0) = 1

Rule 1: 0##0#1 = 0
Rule 2: 1#0##0 = 0
Rule 3: 00###0 = 0
Rule 4: 1###01 = 0
Rule 5: ###### = 1
Tested using 467 pieces of data
Best's fitness over test set 460
Equals 98.76% accuracy on training set
Equals 98.50% accuracy on test set

8) Result for 5 rules in data set 3 as floating-point ranges showing the alternate version of the rule set.

Rule 1: (0.0 , 0.5190855) (1.0 , 0.48845738) (0.0 , 1.0) (0.0 , 1.0) (0.0 , 1.0) (0.0 , 0.53799367) = 1
Rule 2: (0.46587685 , 1.0) (0.0 , 1.0) (1.0 , 0.50109446) (0.0 , 1.0) (0.0 , 1.0) (0.4999999 , 0.0) = 1
Rule 3: (0.4999999 , 0.0) (0.0 , 1.0) (1.0 , 0.0) (0.5 , 1.0) (0.0 , 1.0) (1.0 , 0.5) = 1

Rule 4: (0.49479422 , 1.0) (0.0 , 1.0) (0.0 , 1.0) (0.0 , 1.0) (0.4999999 , 1.0) (1.0 , 0.45964053) = 1
Rule 5: (0.0 , 1.0) (1.0 , 0.0) (1.0 , 0.0) (1.0 , 0.0) (0.0 , 1.0) (0.0 , 1.0) = 0

Rule 1: 01###0 = 1
Rule 2: 1#1##0 = 1
Rule 3: 0##1#1 = 1
Rule 4: 1###11 = 1
Rule 5: ###### = 0
Trained using 1529 sets of data
Best's fitness on the training set 1504
Tested using 471 pieces of data
Best's fitness over test set 460
Equals 98.36% accuracy on training set
Equals 97.66% accuracy on test set

9) Pseudocode to covert data set 3 rulebase into bit string.

```
IF( a < b)
   IF(a >= 0.4){
      Return 1
   ELSE IF(b <= 0.6)
      Return 0
   ELSE
      Return #
   END IF
ELSE
    IF (b >= 0.4)
      Return 1
    ELSE IF (a <= 0.6)
       Return 0
     ELSE
       Return #
     END IF
END IF
```