Evolving a classifier with an applied rule based approach

Jordan Draper 16018552

1 INTRODUCTION

The task was to solve a set of classification problems and pull the data structure from four different data sets as effectively as possible by applying any form of evolutionary algorithm. My approach was to apply a rule-based search with the genetic algorithm to evolve the classifier to solve the problem. Modifying parameters would manipulate the algorithms efficiency and allow me to extract the rules to interpret.

2 BACKGROUND RESEARCH

When dealing with classification problems conversation often turns to neural networks and rule based systems, another thing to consider is random forests. "Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest" (Breiman ,2001, pp. 5-32). What this means is that random forest is a classification algorithm with a large number of decision trees that operate as a group, each tree produces a class prediction and then whichever class gets the most votes becomes the model prediction. The trees are created randomly which means they are not closely correlated, the strength in this is that the trees protect each other from their individual errors, as a collective they should be able to move towards a solution. To ensure diversity between the trees something called bootstrap aggregation also known as bagging is used, with bagging we do not subset the training data in smaller sets and train each tree on a different set, instead we are feeding each tree the training set with replacement, so the training data can be the same. "There are two reasons for using bagging. The first is that the use of bagging seems to enhance accuracy when random features are used. The second is that bagging can be used to give ongoing estimates of the generalisation error (PE∗) of the combined ensemble of trees, as well as estimates for the strength and correlation" (Breiman, 2001, pp.5-32). Random features can be explained in that the random trees can only pick from a random subset of features when making a decision about the model which creates variation between the trees in the model.

Talking more about training data brings me onto big data and data mining. Big data can aid researchers to solve large and complex problems, however even when data-mining efforts produce models that provide insights we should still be cautious. "Big Data models are typically built from happenstance data. Even though they may point to associations between variables that can be manipulated to achieve a desired response, they cannot provide a causal link between the two. There may be other lurking variables that actually caused the response to change that either were not measured or were so correlated with other variables that their contribution was masked" (De Veaux et al, 2016, pp. 411-416). This research is relevant because when working with the datasets it is important to try understand what I am working with, this directly applies towards using a genetic algorithm as well, "by cycling between discovery and experimentation, the knowledge process can progress. Even though data mining cannot answer the question, it can pose new questions, and insights gained from empirical models can be used to accelerate the learning from the experimental process" (De Veaux et al, 2016, pp. 411-416).

There were surveys conducted that found 63% of the adult UK population are uncomfortable with AI systems replacing doctors and nurses with the tasks they normally perform. "These survey data resonate to the ethical and regulatory challenges that surround AI in healthcare, particularly privacy, data fairness, accountability, transparency, and liability. Successfully addressing these will foster the future of machine learning in medicine (MLm) and its positive

impact on healthcare." (Effy et al, 2018, pp. 1549-1676). Moving towards data mining and touching on liability there would be a question of who is responsible if a healthcare system built from previous data makes an error and diagnoses someone incorrectly; would it be the fault of the programmer, the organization, whoever provided the data or even who made the decision to use the system. The data set used also has to successfully represent the population. There is a saying, "garbage in, garbage out", so not representing the population correctly during training phases means the mined data is useless. Additionally poorly represented training data introduces biases. Keeping discussion of data mining ethics directed towards a medical application, the more interventions that arise down to data fed MLm the more autonomy is lost for patients, "the autonomy of patients in decisional processes about their health and the possibility of shared decision making may be undermined" (Effy et al, 2018, pp. 1549-1676). If the medical intervention ended up being incorrect it would cause lots of problems and put the healthcare industry under even more scrutinization than it currently is.

3 EXPERIMENTATION

3.1 Data Set 1

My genetic algorithm has alterable parameters of gene length, population size, generation count and mutation rate. It also employs a rule base and a wild card. The individuals are created at random comprised of 0s, 1s and 2s, the 2 is the wildcard generalisation added to the binary set. Generalisation is used for identifying and removing redundant data and we can attempt to pull some structure out of the data with less rules by allowing the 2s to classify as either a 1 or a 0. Calculating fitness works by taking data set 1 and cycling over each entry in the set and comparing it against the rule base, when it finds the first rule with matching conditions and output then the fitness increments by 1 and moves to the next datapoint. Maximum fitness in dataset 1 is 60 due to having 60 data entries. The rule base is created by breaking down an individual, an individual with 70 genes could be broken down into 10 rules, as 6 bits build the condition and the 7th bit is the output, this matches up with the data sets.

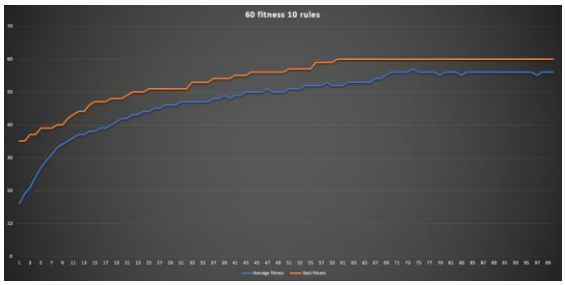

Figure 1: Reaching 60 fitness on dataset 1 with 10 rules.

Altering parameters of the GA means behavior of the classifier can be altered, tracked and therefore controlled. Mutation is an explorative function that creates small permutations in the search space, from multiple tests I deduced the sweet spot seemed to be around 1/chromosome length. Figure 2 shows that having a low mutation rate of 1 in 140 keeps the average population and the fittest individual very close together, it appears the individuals per generation are getting fitter together, the drawback here is that the average population reaches a peak of 49 fitness at generation 98. The second graph in figure 2 shows a high mutation rate of 1 in 15 per gene, the constant mutating is keeping the average population relatively low whilst the fittest individual excels in comparison to the overall population. The average fitness peaked at 32 whilst the fittest individual reached 48 which was discovered by generation 38.

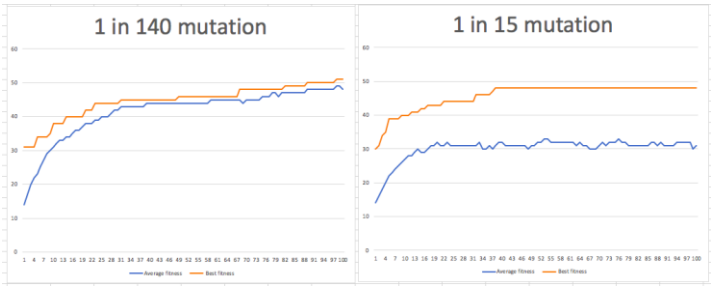

Figure 2: Comparison of mutation extremities on dataset 1.

The structure that I pulled out of data set 1 was with 5 rules, 35 gene length, a mutation rate of 1/70, 200 population and 100 generations, the highest fitness was 60 with the peak average being 58.
Structure:

012202 0, 002220 0, 102022 0, 110222 0, 222222 1
Visually you can see the first 2 bits are what is classifying the data and the first bits are 01, 00, 10, 11 and the generalisation of 22, if you take the number from the binary of the first to bits, e.g. 01 binary is 1 in decimal so you count in this amount of bits from the right and that is what the action bit results in.

3.2 Data Set 2

This dataset is the same as set 1 except more rules are required to try classify the data. Figure 3 shows dataset 1 parameters on dataset 2.

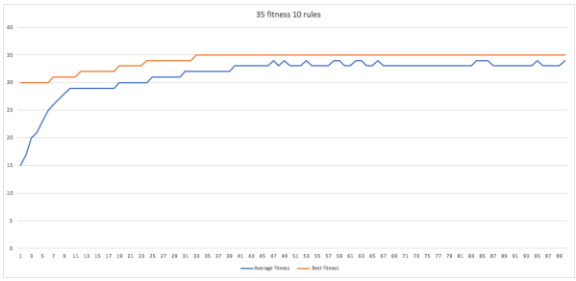

Figure 3: Applying dataset 1 parameters on dataset 2.

To attempt classification I increased generation count to 300 and the population to 600 to see if running it longer had impact.

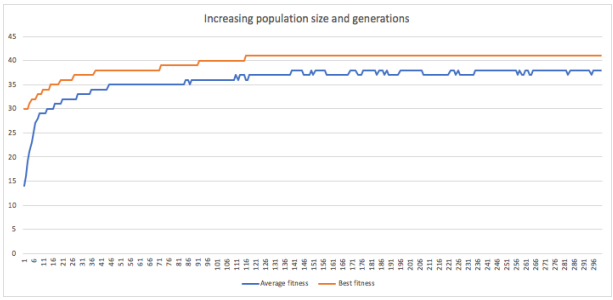

Figure 4: Increasing the generation count and population size in an attempt to classify data set 4.

Next I set my generations to 5000, gene length, population and mutation rate I set equally to 280 which also meant there would be 40 rules. This is how I achieved a peak individual fitness of 60 with an average fitness of 58, figure 5 shows the graph of these results.

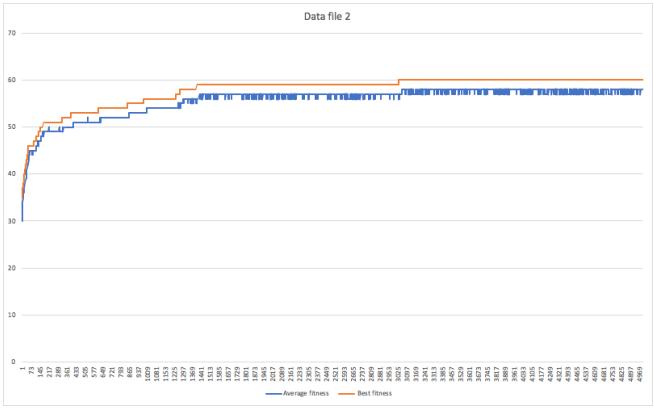

Figure 5: Reaching 60 fitness on dataset 2 after 3056 generations

The rules I pulled from dataset 2:
111100. 0, 111010. 0, 202001. 0, 120122. 0, 220200. 0, 022011. 0, 012222. 0, 100111. 0, 101101. 0, 100011. 0, 021110. 0, 112020. 0, 020122. 0, 121022. 0, 200100. 0, 201210. 0, 000000. 0, 010111. 0, 001111. 0, 002120 .0, 120012. 0, 211000. 0, 012112. 1, 111211. 1, 010121. 1, 101010. 1, 111220. 1, 021200. 1, 220211. 1, 002122. 1, 210200. 1, 110020. 1, 202222. 1, 120001. 1, 111122. 1, 111100. 1, 211022. 1, 000100. 1, 100021. 1, 112010. 1
The structure I have noticed within dataset 2 is that for one classification group if the 1s and 0s are even both even numbers e.g. 110000 is 1(x2) and 0(x4) the 2 and 4 being even so it classifies as a 0 here. To classify into the other classification the amount of 1s and 0s has to be odd numbers, e.g. 100000 1(x1) and 0(x5) the 1 and 5 are odd numbers so it classifies as a 1 on the action bit in this instance.
The data point “000011 1” is the only one that I can see that goes against the rule structure, this would explain why sometimes the algorithm only reaches 59 fitness instead of 60, when removing this datapoint I was able to get 59 fitness quickly.

3.3 Data Set 3

Set 3 changed the data entries to floating point numbers within the range 0.000001 to 0.999999. When reading in the data file I changed the floating point numbers to integer by removing the first 2 characters. The number 0.999999 would become

999999, this meant I would not need to rewrite much code changing my data types. I will still refer to it as floating point throughout the report. I implemented lower and upper bounds for each floating data point. Given there are 6 data points there is a lower and upper bounds for each point meaning I would need 12 values and 1 more for the action bit. Therefore 13 genes can represent 1 rule, keeping to 10 rules means changing gene length to 130. Adding upper and lower bounds for each data point enables the algorithm to learn what bounds get the best fitness for each rule as we still have our action point as a feedback for what is satisfying the conditions.

Figure 6: Example of lower and upper bounds for each data point on data set 3 e.g. the 10 rules

```
Average population fitness: 770
The best rulebase is:
< 2574,518876 > < 6420,998767 > < 297,500078 > < 140,656728 > < 1307,990567 > < 7856,992894 > BIT: 0
< 474471,998538 > < 496414,998385 > < 786,998524 > < 126,997554 > < 1390,998710 > < 310,504723 > BIT: 0
< 987857,767900 > < 308297,947803 > < 646847,572593 > < 530684,846499 > < 900000,100000 > < 464137,412712 > BIT: 1
< 476647,348087 > < 503741,624506 > < 873652,341684 > < 306334,414033 > < 172382,691326 > < 88510,488094 > BIT: 1
< 137704,851342 > < 870607,219190 > < 356739,626128 > < 381868,199945 > < 612264,588555 > < 277235,860547 > BIT: 1
< 136537,485756 > < 900000,926202 > < 900000,229783 > < 616609,496430 > < 393867,274104 > < 735107,900000 > BIT: 1
< 245877,762551 > < 348242,922297 > < 170176,272447 > < 453099,677432 > < 39658,900000 > < 852411,461324 > BIT: 1
< 824184,996488 > < 778742,590149 > < 412201,393015 > < 764837,733348 > < 100000,263088 > < 876398,441505 > BIT: 1
< 43,999310 > < 1807,998796 > < 4798,999800 > < 4904,999065 > < 439,999275 > < 6158,999868 > BIT: 1
< 887043,119151 > < 887174,678315 > < 31785,920149 > < 553587,842475 > < 928293,100000 > < 469656,884427 > BIT: 1
 The fittest individuals fitness is 779
Generation: 1021
```

The mutation code had to be adjusted to deal with floating point numbers, I extended the existing mutation by adding something called “creeping” which involved creating a float jump value of + or - between 0.000001 and 0.200000 which makes the value then “creep” an amount within the range rather than completely switching value when mutated such as the implementation from the binary data set. If a value went out of bounds I just reset it into the bounds, for example going too low set the value back to 0.000001.

I split half the data (1000 entries) into training data to train my algorithm, the second half of the data was used as unseen data (remaining 1000 entries) once I had completed the generations just to see how this individual would perform on the unseen data set. This would be effective to show the uses of training data.

An initial run can be seen in figure 7 where gene length is 130, population is 200, creep maximum is 0.200000 with a mutation chance of 1 in 200.

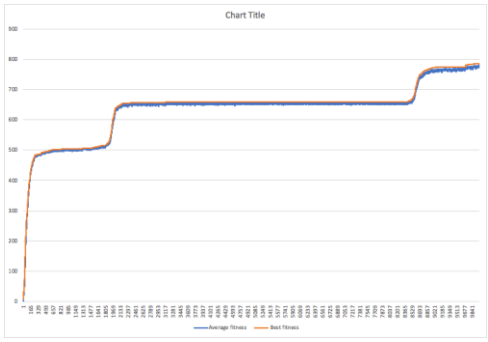

Figure 7: Initial run on dataset 3 achieving a fitness of 785 on training data

When I applied the rule base from figure 7 onto the unseen data it achieved a fitness of 753 out of a possible 1000. I plan to stick to a minimum of 10000 generations for all runs of dataset 3 because clearly from the graph there was a plateau for nearly 6000 generations before the algorithm was able to improve the solution.

Adjusting the population majorly affects the runtime of the algorithm per generation, I started at 200 population in figure 7. By changing it to 50 and also to 400 the impact can be seen on figure 8.

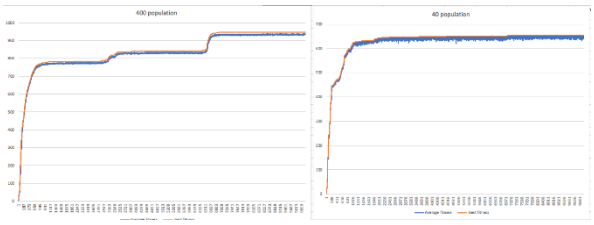

Figure 8: population changes, 50 on left 400 on right

From running the algorithm with population 400 I was able to achieve a peak fitness of 948 which satisfies 94.8% of the training data, when applying this to the unseen data it achieved 90.5% (905/1000) which is an excellent result for trying to spot the data structures.

The next thing I tried was to lower the population to 150 to increase the speed of the GA. Then I decided to try altering mutation chance, from a high chance of 1 in 30 to a lower chance of 1 in 200. Figure 8 shows the results. Having a high mutation chance of 1 in 30 led to the GA only reaching 788 peak fitness whereas having a 1 in 200 reached a peak of 876. Again like the binary dataset having the mutation rate similar to

the gene length or population size leads to the better results.

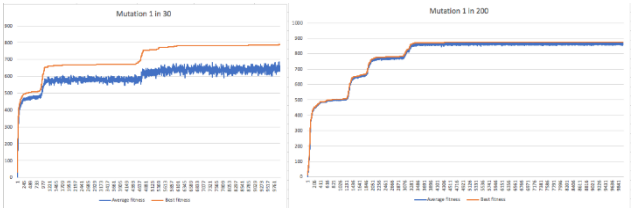

Figure 8: Affects of varying mutation rates

The creep aspect of the mutation function would allow the float to creep any range from 0.000001 to 0.200000, I decided to try setting the range to 0.100000 and 0.300000 to see how that affected the mutation. I noticed that there seemed to be more jumps up in the data when I had a higher creep rate, having it quite low looks like the changes take a lot longer to happen. However from what I learnt on the binary data set if I set mutation too high then it is also negative when trying to reach the goal in optimal time.

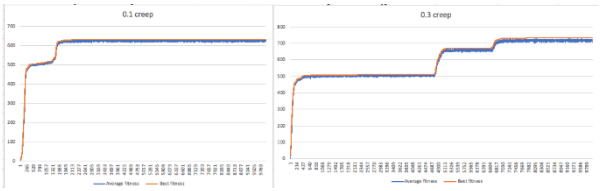

Figure 9: Comparison on setting 0.1 and 0.3 creep limits

After experimenting with the parameters the target was to achieve optimal fitness and test how the top individual performs against the remaining unseen data. I also wanted to work with 5 rules because with 10 rules it is hard to determine the rules, much like data set 1. With parameters set to 65 gene length, 5 rules, 400 population, 1 in 200 mutation with a 0.2 creep distance. After multiple runs the best fitness could not beat what was reached with 10 rules. 948 on training data with 905 on the unseen data (10 rules). The 5 rules are below, 771 fitness was the maximum reached with 5 rules, even though fitness was lower they are still useful for estimations:
Rule 1:
< 999999,587449 > < 629959,144823 > < 305528,309721 > < 368628,955282 > < 999999,219450 > < 368341,360964 > BIT: 0
Rule 2:
< 1,504796 > < 1,502005 > < 501977,999999 > < 10297,999999 > < 1,999999 > < 1,999999 > BIT: 1
Rule 3:
< 491812,999999 > < 1,999999 > < 1,999999 > < 1,999999 > < 501071,999999 > < 499932,999999 > BIT: 1
Rule 4:
< 1,999999 > < 1,999999 > < 1,999999 > < 1,999999 > < 1,999999 > < 1,999999 > BIT: 0
Rule 5:
< 334951,132127 > < 502903,355580 > < 65706,192837 > < 956266,49015 > < 305730,999999 > < 743689,409767 > BIT: 1

Working with the 5 rules and the 10 rules where 95% fitness was reached I deduced if we round the float to the nearer of 0 or 1 within the bounds on the data then the pattern seems the same as the binary dataset 2 except where we convert the first 2 bits from binary to decimal and count from the right and whatever value is in that place is the action bit value we this time count from the left instead.
An example is <488312,997065> would round to a 1.
A full example is below:
< 485742,994497><14428,900000><5357,758370 > <7710,997241> <507504,993179><15078,976877 > BIT: 1
<485742,994497> = round to 1
<14428,900000> = round to 1
<5357,758370 > = round to 1
<7710,997241> = round to 1
<507504,993179> = round to 1
<15078,976877> = round to 1
Taking the first two bits 11 is 3 in decimal, looking at the full binary number 111111 and counting in 3 bits from the left shows 1, which matches the action bit. This pattern applies for most of the rules when the fitness was high.
The fitness is not 100% which means the point at which we round up or down may be 0.6 instead of 0.5. We need 4 rules to find the rule for 00, 01, 10, 11 and then 1 rule is likely to be generalisation, a common rule I saw during testing was 1,999999 > < 1,999999 > < 1,999999 > < 1,999999 > < 1,999999 > < 1,999999 > BIT: 0. This rule acts as a generalisation as it can take every value within the

float range. Also to work out if it is a 1 or a 0 we can visualise the floats as a percentage, e.g. 0.999999 could be seen as 99.9% chance of being a 1 in binary. Figure 10 shows the highest fitness I achieved on the training dataset of around 95%.

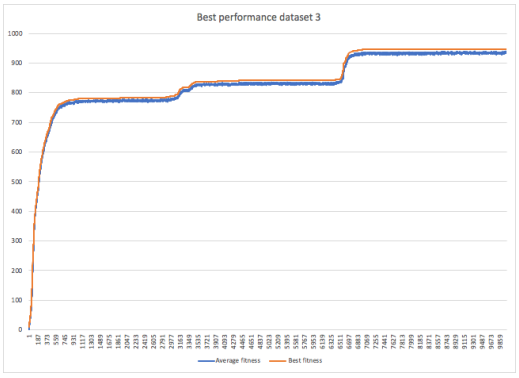

Figure 10: Best performance on dataset 3

3.3 Data Set 4

Dataset 4 was similar to dataset 3 except instead of 6 floats leading to 1 action bit there was 10 floats leading to the action bit, again I chose to apply a lower and upper bound to these action bits, being 10 floats meant there would be 20 lower and upper points with 1 action bit, each rule would be represented by 21 genes. 10 rules x21 genes means each individual should be 210 genes long. I adjusted my file reading code to handle the extra input and then attempted to run my code. I stopped running it after 1000 generations because my fitness was still at 0. Figure 11 shows my test before splitting my data into training and unseen, I ran this test to ensure the logic I had created for the GA with the new datafile would generally work.

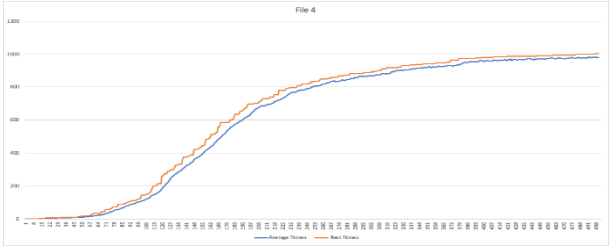

Figure 11: Running dataset 4 without separating training data from unseen data

Figure 11 reached around 1003 fitness out of 1749, it took 1050 genes (50 rules), 600 population with a 1 in 140 chance for mutation.

I then split the data into training and test data, the training comprised of of the first 700 entries and the test or unseen data was the remaining 1049 entries. Based on what I had learnt from from dataset 2 I assumed that I would need to tune the parameters to be more efficient and increase the amount of rules past 10, I decided to start at 60 rules in figure 11, figure 12 onwards I reduced the rules down to 40. I also added some logic where if a lower bound value went higher than the upper bound value then it would switch values with its pair, this improved fitness results drastically. After separating the data into 700 training figure 12 shows what fitness was achieved, 646/700 which is roughly 92% .

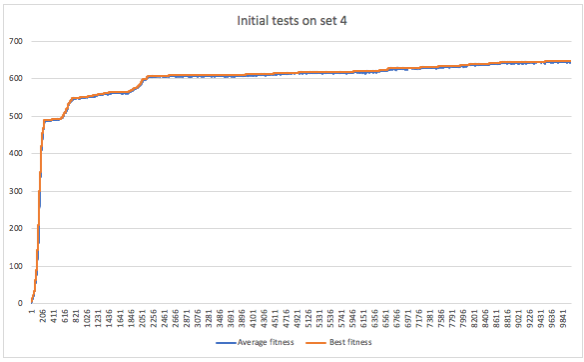

Figure 12: 92% fitness on training data set 4

Clearly the parameters I used of 840 gene length, 200 population 40 rules and a 1 in 800 mutation chance per gene were effective to reach a high fitness. The next thing I tested was to see how different amounts of training data could affect the algorithm, I changed the training data from 700 entries to 1000 and then graphed the average, figure 13 shows the results

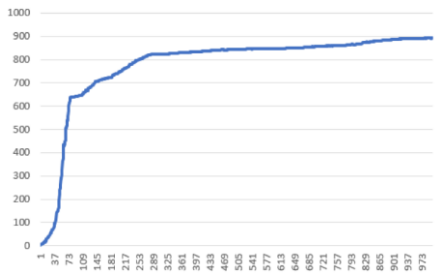

Figure 11: Dataset 4, 1000 training instead of 700

It is possible I needed to have a more drastic split between testing and training but the result was virtually the same when looking at the fitness percentage wise, it reached around 90% both times and the rules it output were the same. I would have liked more time to test parameters on dataset 4 to try reduce the amount of rules down from 40 to make it easier to find a clear and defined rule.

The rules I pulled out of the 92% fitness run can be seen on figure 12 below, I added it as an image to be clearer visually.

Figure 13: 20 rules from 92% fitness on dataset 4

Again I can apply rounding on the floats to bring them to a 0 or a 1 and undoubtably I would see a pattern in the data, it is likely again to be a lot of noise within the data and only a few of the binary values will be of importance to what the rule actually is, however given I only reduced it down to 40 rules it is too many in this instance to work them out, If I had been able to reduce the code to 5 or 10 rules like in previous datasets it is likely it would be easier to determine the rules and test them.

4 CONCLUSIONS

I found how a GA can be applied to pull a structure from data, if the data points had a relation to something such as having symptoms and a the action bit being having a disease then I could pull information from it and show which symptoms meant you would probably get the disease represented by the action bit, real world application of this is much clearer now.

Something I would do differently next time would be to look ahead at the format of the data, thinking of a way to handle the floating points took time that could of been saved by the algorithm being able to handle floats and ints from the start. Additionally relating back to my initial research, it is incredibly important to check the quality of data and remove any outliers of information, it is an issue that cropped up with dataset 2 where 1 value did not fit with the rules. Runtime of my algorithm is something I should consider when working with my laptop, I imagine with a powerful super computer my algorithm would run incredibly quick, but based on the hardware I am using the GA slowed down a lot with dataset 3 and 4. I tackled it initially by improving the code by flipping the upper and lower bounds of the individuals if they got the wrong way around, it meant it moved towards an overall solution faster, but each generation became visibly a lot slower. I would also like to have reduced dataset 4 rules down from 20 to 10 or below, I was not able to deduce the rules effectively from such a large collection of floats.

REFERENCES

Breiman, L. (2001) Random Forests. *Machine Learning* [online]. 45 (1), pp. 5-32. [Accessed 21 November 2019].

De Veaux, R.D., Hoerl, R.H. and Snee, R.D. (2016) Big Data and the Missing Links. *Statistical Analysis and Data Mining: The Asa Data Science Journal* [online]. 9 (6), pp. 411-416. [Accessed 18 November 2019].

Effy, V., Alessandro, B. and Glenn, C. (2018) Machine Learning in Medicine: Addressing Ethical Challenges. *Public Library of Science* [online]. 15 (11), pp. 1549-1676. [Accessed 21 November 2019].

Source code as an appendix

Binary data set (1&2)

```
package algorithmga;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.Random;
import java.util.Scanner;

/**
 *
 * @author jordandraper
 */
public class AlgorithmGA {

   static int N = 35; //Length of 1s and 0s
   static int P = 200; //population
   static int generations = 100; //Amount of generations to complete
   static int mutationRate = 70; //This is a 1 in x for the mutation rate
   static int conL = 6; //Length of the int from textfile 1 and 2
   static int numR = 5; //Amount of rules within the gene length 70/7 = 10.
   Population population = new Population(); //Creates the population of individuals
   Rulebase rulebase = new Rulebase(); //Creates the population of individuals
   int indI = 0;
   static int file1Len = 60; //Length of file 1
   static int wildCard = 2; //Wildcard 2

   int indexWorstOffspring;

   public static void main(String[] args) throws FileNotFoundException, UnsupportedEncodingException {
      PrintWriter writer = new PrintWriter("/Users/jordandraper/Desktop/Files/University/Year
3/BioComp/algorithmGA/src/algorithmga/GAoutput.txt", "UTF-8");
      writer.println("Average fitness, Best fitness");
      AlgorithmGA demo = new AlgorithmGA();

      //Need to calculate and track best fitness in population and mean fitness in population
      //Initialize population
      demo.population.initializePopulation(P);
      demo.rulebase.initializeRulebase(numR);
      demo.rulebase.initializeDataFile(file1Len);

      demo.loadDataFile(); //Load data1.txt into array
      //demo.showRulesDataFile(); //Shows rules loaded from data 1 - test purposes.

      demo.calculateFitness();  //call function to calc fitness

      demo.totalPopulationFitness(); //Populations fitness

      demo.fittestIndividual();

      for (int generationCount = 0; generationCount < generations; generationCount++) {
         System.out.println("Generation: " + generationCount);

         //Find index of best parent
         demo.indexBestParent(); //need to copy best parent out somewhere
         //Do selection of parents
         demo.selection(); //Perform selection of parents randomly into offsprings
         //Do crossover (recombine parents)
         demo.crossover();
         //Do mutation and results goes to offspring
         demo.mutation();

         //Find index of worst offspring
         demo.indexWorstOffspring();
```

```
            //Swap worst offspring for best parent
            demo.swapBestParentWorstOffspring();
            //Load the offspring to replace parents
            demo.survivorSelection();
            //Recalculate fitness of the population
            demo.calculateFitness();

            //Print average population fitness and top fitness
            writer.println(demo.totalPopulationFitness() + "," + demo.fittestIndividualFitness());
            //Prints fittest individual in this generation
            demo.fittestIndividual();
        }

        //Termination
        writer.println("Fittest individual below: ");
        writer.println(demo.fittestIndividual());
        writer.close(); //Stop writing to file
    }

    //Class for Individual
    class Individual {

        int[] gene = new int[N]; //This needs to be equal to N
        int fitness;

        public Individual() { //default construcor setting everything to 0
            Random rn = new Random();

            //Set genes randomly for each individual
            for (int i = 0; i < N; i++) {

                if ((i + 1) % 7 == 0) { //If i is in the action place we must allow it to only be 0 or 1.
                    gene[i] = Math.abs(rn.nextInt() % 2);
                } else {
                    gene[i] = Math.abs(rn.nextInt() % 3); //gene values can be 2 sometimes

                }

            }
            fitness = 0;
        }
    }
//Class for population

    class Population {

        Individual[] individuals = new Individual[P];
        Individual[] offspring = new Individual[P];
        Individual[] bestParent = new Individual[P];

        //Initialize population
        public void initializePopulation(int size) {
            for (int i = 0; i < P; i++) {
                individuals[i] = new Individual();
                offspring[i] = new Individual(); //array to store offsprings
                bestParent[i] = new Individual(); //array to store the best parent
            }
        }
    }

    //Class for Conditions
    class Condition {

        int[] cond = new int[conL]; //This has length 6 for the first 6 bits
        int out; //Fitness is the 7th bit

        public Condition() { //default construcor setting everything to 0
            Random rn = new Random();
```

```
            //Set genes randomly for each individual
            for (int i = 0; i < conL; i++) {
               cond[i] = Math.abs(rn.nextInt() % 2);
            }
            out = 0;
         }
      }
   //Class for Rulebase

      class Rulebase {

         Condition[] conditions = new Condition[numR]; //Creates 10 rule spaces
         Condition[] dataFile1 = new Condition[file1Len]; //Creates an array

         //Initialize Rulebase
         public void initializeRulebase(int size) {
            for (int i = 0; i < numR; i++) {
               conditions[i] = new Condition();
            }
         }

         public void initializeDataFile(int size) {
            for (int i = 0; i < file1Len; i++) {
               dataFile1[i] = new Condition();
            }
         }
      }

      public void calculateFitness() {

         for (int i = 0; i < P; i++) {
            population.individuals[i].fitness = 0; //Reset fitness to 0
            population.offspring[i].fitness = 0; //Reset fitness sum to 0
            population.bestParent[i].fitness = 0; //Reset fitness sum to 0

            parentToRulebase(i);
            fitnessOfRulesInd(i);

            offSpringToRulebase(i);
            fitnessOfRulesOffspring(i);

            bestParentToRulebase(i);
            fitnessOfBestParent(i);
         }

      }

      public int totalPopulationFitness() {
         calculateFitness();
         int totalFitness = 0;
         int populationFitness = 0;
         for (int i = 0; i < P; i++) { //Cycling population
            totalFitness = totalFitness + population.individuals[i].fitness;
         }

         populationFitness = totalFitness / P;

         System.out.println("Average population fitness: " + populationFitness);
         return populationFitness;
      }

      public void selection() {
         for (int i = 0; i < P; i++) { //Will create same amount of offspring as there is parents
            int parent1 = (int) (Math.random() * P); //Randomly select 2 parents
            int parent2 = (int) (Math.random() * P);
            if (population.individuals[parent1].fitness >= population.individuals[parent2].fitness) {
               //Whichever parent is more fit is added to an array called offspring
               for (int x = 0; x < N; x++) {
                  population.offspring[i].gene[x] = population.individuals[parent1].gene[x];
```

```
            }
        } else {
            for (int x = 0; x < N; x++) {
                population.offspring[i].gene[x] = population.individuals[parent2].gene[x];
            }
        }
    }
    calculateFitness();
}

public void crossover() { //This is an explorative function, makes a big jump between parent areas
    //Choose a random point on the two parents, Split parents at this crossover point
    int crossoverPoint = (int) (Math.random() * N);
    //Create children by exchanging tails
    //For now just crossing over between 2 parents randomly - can change this to cycle all parents if needed
    int parent1 = (int) (Math.random() * P); //Randomly select 2 parents
    int parent2 = (int) (Math.random() * P);
    //int parent1 = 0;
    //int parent2 = 1;
    //System.out.println("CROSSOVER " + crossoverPoint);
    //Create an N digit temp array
    for (int i = N - 1; i > crossoverPoint; i--) {

        //Put parent array 1 genes up to crossover the temp array
        int temp = population.offspring[parent1].gene[i];
        //Put parent array 2 genes up to crossover point into parent array 1
        population.offspring[parent1].gene[i] = population.offspring[parent2].gene[i];
        //Put temp array genes up to the crossover into parent array 2
        population.offspring[parent2].gene[i] = temp;
    }
    calculateFitness();
}

public void mutation() { //This is an explorative function creating random small perpubations
    //Alter each gene independently with a probability Pm
    //int mutationRate
    //Pm is called the mutation rate. Typically between 1/pop_size and 1/ chromosome_length
    for (int i = 0; i < P; i++) {
        for (int x = 0; x < N; x++) {

            boolean probability = new Random().nextInt(mutationRate) == 0; //Probability 1/chromosome_length applied here
            if (probability) {
                if ((x + 1) % 7 == 0) { //If x mod 7 = 0 then it's the 7th bit (action) and must not be edited
                    if (population.offspring[i].gene[x] == 0) {
                        population.offspring[i].gene[x] = 1;
                    } else if (population.offspring[i].gene[x] == 1) {
                        population.offspring[i].gene[x] = 0;
                    }
                }
             else {
                    int randomChance = new Random().nextInt(3);
                    switch (randomChance) {
                        case 1:
                            population.offspring[i].gene[x] = 0;
                            break;
                        case 2:
                            population.offspring[i].gene[x] = 1;
                            break;
                        default:
                            population.offspring[i].gene[x] = 2;
                            break;
                    }
                    }
                }
            }
        }
    calculateFitness();
}
```

```
public void survivorSelection() {//Replace offspring with parents
   for (int i = 0; i < P; i++) {
      for (int x = 0; x < N; x++) {
         population.individuals[i].gene[x] = population.offspring[i].gene[x];
      }
   }
}

public void indexBestParent() { //Find the parent with best fitness and write it to bestParent[0]
   calculateFitness();
   int actualFitness = 0;
   int indexTopParent = 0;
   for (int i = 0; i < P; i++) {
      if (population.individuals[i].fitness > actualFitness) {
         actualFitness = population.individuals[i].fitness;
         indexTopParent = i; //Can use this index
      }
   }
   for (int x = 0; x < N; x++) {
      population.bestParent[0].gene[x] = population.individuals[indexTopParent].gene[x];
   }
}

public int indexWorstOffspring() { //Replace the worst fitness in new gen from best fitness in last gen
   calculateFitness();
   int indexWorstOffspring = 0;
   int actualFitness = N;
   for (int i = 0; i < P; i++) {
      if (population.offspring[i].fitness < actualFitness) {
         actualFitness = population.offspring[i].fitness;
         indexWorstOffspring = i; //Can use this index
      }
   }
   return indexWorstOffspring;
}

public void swapBestParentWorstOffspring() {
   int indexWorstOffspring = indexWorstOffspring();

   if (population.bestParent[0].fitness > population.offspring[indexWorstOffspring].fitness) {
      for (int x = 0; x < N; x++) {
         population.offspring[indexWorstOffspring].gene[x] = population.bestParent[0].gene[x];
      }
   }
}

public String fittestIndividual() {
   String individual = "";
   int index = 0;
   int topFitness = 0;
   for (int i = 0; i < P; i++) {
      if (population.individuals[i].fitness > topFitness) {
         index = i;
         topFitness = population.individuals[i].fitness;
      }

   }
   System.out.println("The best rulebase is: ");
   for (int x = 0; x < N; x++) {
      individual = individual + population.individuals[index].gene[x];
      System.out.print(population.individuals[index].gene[x]);
   }

   System.out.println(" The fittest individuals fitness is " + topFitness);

   return individual;
}

public int fittestIndividualFitness() {
```

```
        int index = 0;
        int topFitness = 0;
        for (int i = 0; i < P; i++) {
            if (population.individuals[i].fitness > topFitness) {
                index = i;
                topFitness = population.individuals[i].fitness;
            }
        }
        return topFitness;
    }
    public void loadDataFile() throws FileNotFoundException {
        Scanner scanner = new Scanner(new File("/Users/jordandraper/Desktop/Files/University/Year
3/BioComp/algorithmGA/src/algorithmga/data2.txt"));
        int i = 0;
        while (scanner.hasNextLine()) {

            String line = scanner.nextLine();

            rulebase.dataFile1[i].cond[0] = Integer.parseInt(String.valueOf(line.charAt(0)));
            rulebase.dataFile1[i].cond[1] = Integer.parseInt(String.valueOf(line.charAt(1)));
            rulebase.dataFile1[i].cond[2] = Integer.parseInt(String.valueOf(line.charAt(2)));
            rulebase.dataFile1[i].cond[3] = Integer.parseInt(String.valueOf(line.charAt(3)));
            rulebase.dataFile1[i].cond[4] = Integer.parseInt(String.valueOf(line.charAt(4)));
            rulebase.dataFile1[i].cond[5] = Integer.parseInt(String.valueOf(line.charAt(5)));

            //We have a space within the string format "xxxxxx x"
            rulebase.dataFile1[i].out = Integer.parseInt(String.valueOf(line.charAt(7))); //7th including space.

            // System.out.println(line); //line
            i++;

            //Should load the file into an array
        }
        //System.out.println("Data file load complete");
    }

    public void parentToRulebase(int indI) {
        int out = 0;
        int k = 0;
        for (int i = 0; i < numR; i++) {
            for (int j = 0; j < conL; j++) {
                rulebase.conditions[i].cond[j] = population.individuals[indI].gene[k++];
            }
            rulebase.conditions[i].out = population.individuals[indI].gene[k++];
        }
    }

    public void showparentToRulebase() {
        for (int x = 0; x < numR; x++) {

            for (int i = 0; i < conL; i++) {
                System.out.print(rulebase.conditions[x].cond[i]);
            }
            System.out.print(" 7th bit: ");
            System.out.print(rulebase.conditions[x].out);
            System.out.println(" ");
        }
        return;
    }

    public void showRulesDataFile() {
        for (int i = 0; i < file1Len; i++) {

            for (int x = 0; x < conL; x++) {
                System.out.print(rulebase.dataFile1[i].cond[x]);
            }
            System.out.print(" 7th bit: ");
            System.out.print(rulebase.dataFile1[i].out);
            System.out.println(" ");
```

```
        }
        return;
    }

    public void fitnessOfRulesInd(int indI) {
        population.individuals[indI].fitness = 0;
        for (int i = 0; i < file1Len; i++) { //Cycles through the amount of rules from data file e.g. 60
            for (int j = 0; j < numR; j++) {  //Cycles through amount of rules in rules from the individual e.g. 10
                if (((rulebase.dataFile1[i].cond[0] == rulebase.conditions[j].cond[0]) || (rulebase.conditions[j].cond[0] == wildCard))
                        && ((rulebase.dataFile1[i].cond[1] == rulebase.conditions[j].cond[1]) || (rulebase.conditions[j].cond[1] == wildCard))
                        && ((rulebase.dataFile1[i].cond[2] == rulebase.conditions[j].cond[2]) || (rulebase.conditions[j].cond[2] == wildCard))
                        && ((rulebase.dataFile1[i].cond[3] == rulebase.conditions[j].cond[3]) || (rulebase.conditions[j].cond[3] == wildCard))
                        && ((rulebase.dataFile1[i].cond[4] == rulebase.conditions[j].cond[4]) || (rulebase.conditions[j].cond[4] == wildCard))
                        && ((rulebase.dataFile1[i].cond[5] == rulebase.conditions[j].cond[5]) || (rulebase.conditions[j].cond[5] == wildCard))) {
//Matches cond
                    if (rulebase.dataFile1[i].out == rulebase.conditions[j].out) { //Matches output
                        //If all 5 bits are the same and the output is the same then fitness ++
                        population.individuals[indI].fitness++;
                    }
                    break;
                }
            } //When the rule matches should we move on??? fitness is going above 10 it shouldnt?
        }
        //System.out.println("Individual fitness based on rules: " + population.individuals[indI].fitness);
    }

    public void offSpringToRulebase(int indI) {
        int out = 0;
        int k = 0;
        for (int i = 0; i < numR; i++) {
            for (int j = 0; j < conL; j++) {
                rulebase.conditions[i].cond[j] = population.offspring[indI].gene[k++];
            }
            rulebase.conditions[i].out = population.offspring[indI].gene[k++];
        }
    }

    public void fitnessOfRulesOffspring(int indI) {
        population.offspring[indI].fitness = 0;
        for (int i = 0; i < file1Len; i++) { //Cycles through the amount of rules from data file
            for (int j = 0; j < numR; j++) {  //Cycles through amount of rules in rules from the individual
                if (((rulebase.dataFile1[i].cond[0] == rulebase.conditions[j].cond[0]) || (rulebase.conditions[j].cond[0] == wildCard))
                        && ((rulebase.dataFile1[i].cond[1] == rulebase.conditions[j].cond[1]) || (rulebase.conditions[j].cond[1] == wildCard))
                        && ((rulebase.dataFile1[i].cond[2] == rulebase.conditions[j].cond[2]) || (rulebase.conditions[j].cond[2] == wildCard))
                        && ((rulebase.dataFile1[i].cond[3] == rulebase.conditions[j].cond[3]) || (rulebase.conditions[j].cond[3] == wildCard))
                        && ((rulebase.dataFile1[i].cond[4] == rulebase.conditions[j].cond[4]) || (rulebase.conditions[j].cond[4] == wildCard))
                        && ((rulebase.dataFile1[i].cond[5] == rulebase.conditions[j].cond[5]) || (rulebase.conditions[j].cond[5] == wildCard))) {
//Matches cond
                    if (rulebase.dataFile1[i].out == rulebase.conditions[j].out) { //Matches output
                        //If all 5 bits are the same and the output is the same then fitness ++
                        population.offspring[indI].fitness++;
                    }
                    break; // note it is important to get the next data item after a match
                }
            }
        }
    }

    public void bestParentToRulebase(int indI) {
        int out = 0;
        int k = 0;
        for (int i = 0; i < numR; i++) {
            for (int j = 0; j < conL; j++) {
                rulebase.conditions[i].cond[j] = population.bestParent[indI].gene[k++];
            }
            rulebase.conditions[i].out = population.bestParent[indI].gene[k++];
        }
    }
```

```
   public void fitnessOfBestParent(int indI) {
      population.bestParent[indI].fitness = 0;
      for (int i = 0; i < file1Len; i++) { //Cycles through the amount of rules from data file
         for (int j = 0; j < numR; j++) {  //Cycles through amount of rules in rules from the individual
            if (((rulebase.dataFile1[i].cond[0] == rulebase.conditions[j].cond[0]) || (rulebase.conditions[j].cond[0] == wildCard))
                  && ((rulebase.dataFile1[i].cond[1] == rulebase.conditions[j].cond[1]) || (rulebase.conditions[j].cond[1] == wildCard))
                  && ((rulebase.dataFile1[i].cond[2] == rulebase.conditions[j].cond[2]) || (rulebase.conditions[j].cond[2] == wildCard))
                  && ((rulebase.dataFile1[i].cond[3] == rulebase.conditions[j].cond[3]) || (rulebase.conditions[j].cond[3] == wildCard))
                  && ((rulebase.dataFile1[i].cond[4] == rulebase.conditions[j].cond[4]) || (rulebase.conditions[j].cond[4] == wildCard))
                  && ((rulebase.dataFile1[i].cond[5] == rulebase.conditions[j].cond[5]) || (rulebase.conditions[j].cond[5] == wildCard))) {
//Matches cond
               if (rulebase.dataFile1[i].out == rulebase.conditions[j].out) { //Matches output
                  //If all 5 bits are the same and the output is the same then fitness ++
                  population.bestParent[indI].fitness++;
               }
               break; // note it is important to get the next data item after a match
            }
         }
      }

   }

} //end
```

Floating point data set (3&4)

```
package datafile4;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.Random;
import java.util.Scanner;

/**
 *
 * @author jordandraper
 */
public class datafile4 {

   static int N = 840; //Length of 1s and 0s - datafile 4 is 10 U/L bounds and 1 action bit
   static int P = 200; //population
   static int generations = 10000; //Amount of generations to complete
   static int mutationChance = 800; //This is a 1 in x for the mutation chance
   static int mutationJump = 200000; //This is the range + or - that the mutation can jump... imagine there is a "0." in front of the value
   static int conL = 10; //Length of the int from textfile 1 and 2
   static int numR = 40; //Amount of rules within the gene length 70/7 = 10.
   static int ruleLength = (N / numR) - 1; //Rule length, EXCLUDES the action bit.
   Population population = new Population(); //Creates the population of individuals
   Rulebase rulebase = new Rulebase(); //Creates the population of individuals
   int indI = 0;
   static int file1Len = 700; //Length of file 1

   int indexWorstOffspring;

   public static void main(String[] args) throws FileNotFoundException, UnsupportedEncodingException {

      PrintWriter writer = new PrintWriter("/Users/jordandraper/Desktop/Files/University/Year
3/BioComp/algorithmGA/src/datafile4/GAoutput.txt", "UTF-8");
      writer.println("Average fitness, Best fitness");
      datafile4 demo = new datafile4();
      //Need to calculate and track best fitness in population and mean fitness in population
      //Initialize population
      demo.population.initializePopulation(P);
      demo.rulebase.initializeRulebase(numR);
      demo.rulebase.initializeDataFile(file1Len);
      demo.loadDataFile(); //Load data1.txt into array
      //demo.showRulesDataFile(); //Shows rules loaded from data 1 - test purposes.
```

```
        demo.calculateFitness();  //call function to calc fitness
        demo.totalPopulationFitness(); //Populations fitness
        demo.fittestIndividual();

        for (int generationCount = 0; generationCount < generations; generationCount++) {
            System.out.println("Generation: " + generationCount);

            //Find index of best parent
            demo.indexBestParent(); //need to copy best parent out somewhere
            //Do selection of parents
            demo.selection(); //Perform selection of parents randomly into offsprings
            //Do crossover (recombine parents)
            demo.crossover();
            //Do mutation and results goes to offspring
            demo.mutation();

            //Find index of worst offspring
            demo.indexWorstOffspring();
            //Swap worst offspring for best parent
            demo.swapBestParentWorstOffspring();
            //Load the offspring to replace parents

            demo.survivorSelection();
            //Recalculate fitness of the population
            demo.calculateFitness();
            //Print average population fitness and top fitness
            writer.println(demo.totalPopulationFitness() + "," + demo.fittestIndividualFitness());
            //Prints fittest individual in this generation
            demo.fittestIndividual();
        }
        //Termination
        writer.println("Fittest individual below: ");
        writer.println(demo.fittestIndividual());
        writer.close(); //Stop writing to file

        //Test last value on unseen data.
        demo.fitnessUnseenData();
    }

    //Class for Individual
    class Individual {

        int[] gene = new int[N]; //This needs to be equal to N
        int fitness;

        public Individual() { //default construcor setting everything to 0
            Random rn = new Random();

            //Set genes randomly for each individual
            for (int i = 0; i < N; i++) {
                if ((i + 1) % 21 == 0) {
                    gene[i] = Math.abs(rn.nextInt() % 2);
                } else {
                    gene[i] = Math.abs(rn.nextInt() % 999999); //Changed from 2 to 999999
                }
            }
            fitness = 0;
        }
    }
//Class for population

    class Population {

        Individual[] individuals = new Individual[P];
        Individual[] offspring = new Individual[P];
        Individual[] bestParent = new Individual[P];

        //Initialize population
        public void initializePopulation(int size) {
```

```
        for (int i = 0; i < P; i++) {
            individuals[i] = new Individual();
            offspring[i] = new Individual(); //array to store offsprings
            bestParent[i] = new Individual(); //array to store the best parent
        }

    }

}

//Class for Conditions this is used just for the datafile conditions
class Condition {

    int[] cond = new int[conL]; //This has length 10

    int out; //Fitness is the 21st bit

    public Condition() { //default construcor setting everything to 0
        Random rn = new Random();

        //Set genes randomly for each individual
        for (int i = 0; i < conL; i++) {
            cond[i] = Math.abs(rn.nextInt() % 999999); //random between 0 to 999999
        }
        out = 0;
    }
}

//This is used to put into conditions the individuals etc that I randomly create
class upperLowerCondition {

    int[] cond = new int[ruleLength]; //This has length 21 for the upper and lower

    int out; //Fitness is the 13th bit

    public upperLowerCondition() { //default construcor setting everything to 0
        Random rn = new Random();

        //Set genes randomly for each individual
        for (int i = 0; i < ruleLength; i++) {
            cond[i] = Math.abs(rn.nextInt() % 999999);
        }
        out = 0;
    }
}
//Class for Rulebase

class Rulebase {

    upperLowerCondition[] conditions = new upperLowerCondition[numR]; //Creates 10 rule spaces
    Condition[] dataFile1 = new Condition[file1Len]; //Creates an array

    //Initialize Rulebase
    public void initializeRulebase(int size) {
        for (int i = 0; i < numR; i++) {
            conditions[i] = new upperLowerCondition();
        }
    }

    public void initializeDataFile(int size) {
        for (int i = 0; i < file1Len; i++) {
            dataFile1[i] = new Condition();
        }
    }
}

public void calculateFitness() {
```

```
    for (int i = 0; i < P; i++) {
        population.individuals[i].fitness = 0; //Reset fitness to 0
        population.offspring[i].fitness = 0; //Reset fitness sum to 0
        population.bestParent[i].fitness = 0; //Reset fitness sum to 0
        switchBoundsInd();
        parentToRulebase(i);
        fitnessOfRulesInd(i);
        switchBoundsOffspring();//Trying to switch all bounds
        offSpringToRulebase(i);
        fitnessOfRulesOffspring(i);
        bestParentToRulebase(i);
        fitnessOfBestParent(i);
    }

}

public int totalPopulationFitness() {
    calculateFitness();
    int totalFitness = 0;
    int populationFitness = 0;
    for (int i = 0; i < P; i++) { //Cycling population
        totalFitness = totalFitness + population.individuals[i].fitness;
    }
    populationFitness = totalFitness / P;
    System.out.println("Average population fitness: " + populationFitness);
    return populationFitness;
}

public void selection() {
    for (int i = 0; i < P; i++) { //Will create same amount of offspring as there is parents
        int parent1 = (int) (Math.random() * P); //Randomly select 2 parents
        int parent2 = (int) (Math.random() * P);
        if (population.individuals[parent1].fitness >= population.individuals[parent2].fitness) {
            //Whichever parent is more fit is added to an array called offspring
            for (int x = 0; x < N; x++) {
                population.offspring[i].gene[x] = population.individuals[parent1].gene[x];
            }
        } else {
            for (int x = 0; x < N; x++) {
                population.offspring[i].gene[x] = population.individuals[parent2].gene[x];
            }
        }
    }
    calculateFitness();
}

public void crossover() { //This is an explorative function, makes a big jump between parent areas
    //Choose a random point on the two parents, Split parents at this crossover point
    int crossoverPoint = (int) (Math.random() * N);
    //Create children by exchanging tails
    //For now just crossing over between 2 parents randomly - can change this to cycle all parents if needed
    int parent1 = (int) (Math.random() * P); //Randomly select 2 parents
    int parent2 = (int) (Math.random() * P);
    //int parent1 = 0;
    //int parent2 = 1;
    //System.out.println("CROSSOVER " + crossoverPoint);
    //Create an N digit temp array
    for (int i = N - 1; i > crossoverPoint; i--) {
        //Put parent array 1 genes up to crossover the temp array
        int temp = population.offspring[parent1].gene[i];
        //Put parent array 2 genes up to crossover point into parent array 1
        population.offspring[parent1].gene[i] = population.offspring[parent2].gene[i];
        //Put temp array genes up to the crossover into parent array 2
        population.offspring[parent2].gene[i] = temp;
    }
    calculateFitness();
}

public void mutation() { //This is an explorative function creating random small perpubations
```

```
    //Alter each gene independently with a probability Pm
    //int mutationRate
    //Pm is called the mutation rate. Typically between 1/pop_size and 1/ chromosome_length
    for (int i = 0; i < P; i++) {
        for (int x = 0; x < N; x++) {
            boolean probability = new Random().nextInt(mutationChance) == 0; //Probability 1/chromosome_length applied here
            if (probability) {

                if ((x + 1) % 21 == 0) {
                    //Need to edit the ==0 to be the upper/lower bounds of a 0 or a 1 int
                    if (population.offspring[i].gene[x] == 0) {
                        population.offspring[i].gene[x] = 1;
                    } else if (population.offspring[i].gene[x] == 1) {
                        population.offspring[i].gene[x] = 0;
                    }
                } else {
                    //need mutation here to handle other numbers.
                    int change = new Random().nextInt(mutationJump); //Gets the amount to mutate by
                    int plusOrMinus = new Random().nextInt(2); //Decides whether we add or minus the value
                    //Need to make sure it stays in bounds of 0.000001 and 0.999999
                    if (plusOrMinus == 0) { //If we got 0 then we add the change
                        population.offspring[i].gene[x] = population.offspring[i].gene[x] + change;
                        if (population.offspring[i].gene[x] > 999999) { //If the value goes above bounds reset it
                            population.offspring[i].gene[x] = 999999;
                        }
                        if (population.offspring[i].gene[x] < 1) { //If the value goes below bounds reset it
                            population.offspring[i].gene[x] = 1;
                        }
                    } else { //If we got 1 then we are minusing the change
                        population.offspring[i].gene[x] = population.offspring[i].gene[x] - change;
                        if (population.offspring[i].gene[x] > 999999) { //If the value goes above of bounds reset it
                            population.offspring[i].gene[x] = 999999;
                        }
                        if (population.offspring[i].gene[x] < 1) { //If the value goes below bounds reset it
                            population.offspring[i].gene[x] = 1;
                        }
                    }
                }
            }
        }
    }
    calculateFitness();
}

public void survivorSelection() {//Replace offspring with parents
    for (int i = 0; i < P; i++) {
        for (int x = 0; x < N; x++) {
            population.individuals[i].gene[x] = population.offspring[i].gene[x];
        }
    }
}

public void indexBestParent() { //Find the parent with best fitness and write it to bestParent[0]
    calculateFitness();
    int actualFitness = 0;
    int indexTopParent = 0;
    for (int i = 0; i < P; i++) {
        if (population.individuals[i].fitness > actualFitness) {
            actualFitness = population.individuals[i].fitness;
            indexTopParent = i; //Can use this index
        }
    }
    for (int x = 0; x < N; x++) {
        population.bestParent[0].gene[x] = population.individuals[indexTopParent].gene[x];
    }
}

public int indexWorstOffspring() { //Replace the worst fitness in new gen from best fitness in last gen
    calculateFitness();
```

```
    int indexWorstOffspring = 0;
    int actualFitness = N;
    for (int i = 0; i < P; i++) {
      if (population.offspring[i].fitness < actualFitness) {
        actualFitness = population.offspring[i].fitness;
        indexWorstOffspring = i; //Can use this index
      }
    }
    return indexWorstOffspring;
  }

  public void swapBestParentWorstOffspring() {
    int indexWorstOffspring = indexWorstOffspring();
    if (population.bestParent[0].fitness > population.offspring[indexWorstOffspring].fitness) {
      for (int x = 0; x < N; x++) {
        population.offspring[indexWorstOffspring].gene[x] = population.bestParent[0].gene[x];
      }
    }
  }

  public String fittestIndividual() {
    String individual = "";
    int index = 0;
    int topFitness = 0;
    for (int i = 0; i < P; i++) {
      if (population.individuals[i].fitness > topFitness) {
        index = i;
        topFitness = population.individuals[i].fitness;
      }

    }

    System.out.println("The best rulebase is: ");
    for (int j = 0; j < N; j = j + 20) {
      individual = "";
      individual = "< " + population.individuals[index].gene[j] + " , " + population.individuals[index].gene[j + 1] + " > "
          + " < " + population.individuals[index].gene[j + 2] + " , " + population.individuals[index].gene[j + 3] + " > "
          + " < " + population.individuals[index].gene[j + 4] + " , " + population.individuals[index].gene[j + 5] + " > "
          + " < " + population.individuals[index].gene[j + 6] + " , " + population.individuals[index].gene[j + 7] + " > "
          + " < " + population.individuals[index].gene[j + 8] + " , " + population.individuals[index].gene[j + 9] + " > "
          + " < " + population.individuals[index].gene[j + 10] + " , " + population.individuals[index].gene[j + 11] + " > "
          + " < " + population.individuals[index].gene[j + 12] + " , " + population.individuals[index].gene[j + 13] + " > "
          + " < " + population.individuals[index].gene[j + 14] + " , " + population.individuals[index].gene[j + 15] + " > "
          + " < " + population.individuals[index].gene[j + 16] + " , " + population.individuals[index].gene[j + 17] + " > "
          + " < " + population.individuals[index].gene[j + 18] + " , " + population.individuals[index].gene[j + 19] + " > ";
      System.out.println(individual);
    }
    System.out.println(" The fittest individuals fitness is " + topFitness);
    return individual;
  }

  public int fittestIndividualFitness() {
    int index = 0;
    int topFitness = 0;
    for (int i = 0; i < P; i++) {
      if (population.individuals[i].fitness > topFitness) {
        index = i;
        topFitness = population.individuals[i].fitness;
      }
    }
    return topFitness;
  }

  //data file 3 structure is
  //0.803662 0.981136 0.369132 0.498354 0.067417 0.422276 0
  public void loadDataFile() throws FileNotFoundException {
    Scanner scanner = new Scanner(new File("/Users/jordandraper/Desktop/Files/University/Year
3/BioComp/algorithmGA/src/datafile4/data4_training.txt"));
    int i = 0;
```

```
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            String[] details = line.split(" ");
            //Take the whole line and split by space, then on the 6 initial floats drop the 0. to make it int
            rulebase.dataFile1[i].cond[0] = Integer.parseInt(details[0].substring(2)); //value1
            rulebase.dataFile1[i].cond[1] = Integer.parseInt(details[1].substring(2)); //value2
            rulebase.dataFile1[i].cond[2] = Integer.parseInt(details[2].substring(2)); //value3
            rulebase.dataFile1[i].cond[3] = Integer.parseInt(details[3].substring(2)); //value4
            rulebase.dataFile1[i].cond[4] = Integer.parseInt(details[4].substring(2)); //value5
            rulebase.dataFile1[i].cond[5] = Integer.parseInt(details[5].substring(2)); //value6
            rulebase.dataFile1[i].cond[6] = Integer.parseInt(details[6].substring(2)); //value7
            rulebase.dataFile1[i].cond[7] = Integer.parseInt(details[7].substring(2)); //value8
            rulebase.dataFile1[i].cond[8] = Integer.parseInt(details[8].substring(2)); //value9
            rulebase.dataFile1[i].cond[9] = Integer.parseInt(details[9].substring(2)); //value10
            rulebase.dataFile1[i].out = Integer.parseInt(details[10]);
            i++; //increment data
        }
    }

            //Load the unseen data
    public void loadDataFileUnseenData() throws FileNotFoundException {
        Scanner scanner = new Scanner(new File("/Users/jordandraper/Desktop/Files/University/Year
3/BioComp/algorithmGA/src/datafile4/data4_test.txt"));
        int i = 0;
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            String[] details = line.split(" ");
            //Take the whole line and split by space, then on the 6 initial floats drop the 0. to make it int
            rulebase.dataFile1[i].cond[0] = Integer.parseInt(details[0].substring(2)); //value1
            rulebase.dataFile1[i].cond[1] = Integer.parseInt(details[1].substring(2)); //value2
            rulebase.dataFile1[i].cond[2] = Integer.parseInt(details[2].substring(2)); //value3
            rulebase.dataFile1[i].cond[3] = Integer.parseInt(details[3].substring(2)); //value4
            rulebase.dataFile1[i].cond[4] = Integer.parseInt(details[4].substring(2)); //value5
            rulebase.dataFile1[i].cond[5] = Integer.parseInt(details[5].substring(2)); //value6
            rulebase.dataFile1[i].cond[6] = Integer.parseInt(details[6].substring(2)); //value7
            rulebase.dataFile1[i].cond[7] = Integer.parseInt(details[7].substring(2)); //value8
            rulebase.dataFile1[i].cond[8] = Integer.parseInt(details[8].substring(2)); //value9
            rulebase.dataFile1[i].cond[9] = Integer.parseInt(details[9].substring(2)); //value10
            rulebase.dataFile1[i].out = Integer.parseInt(details[10]);
            i++; //increment data
        }
    }

    public void parentToRulebase(int indI) {
        int out = 0;
        int k = 0;

        for (int i = 0; i < numR; i++) {
            for (int j = 0; j < ruleLength; j++) {
                rulebase.conditions[i].cond[j] = population.individuals[indI].gene[k++];
            }
            rulebase.conditions[i].out = population.individuals[indI].gene[k++];
        }
    }

    public void showparentToRulebase() {
        for (int x = 0; x < numR; x++) {

            for (int i = 0; i < ruleLength; i++) {
                System.out.print(rulebase.conditions[x].cond[i]);
            }
            System.out.print(" 21st bit: ");
            System.out.print(rulebase.conditions[x].out);
            System.out.println(" ");
        }
        return;
    }

    public void showRulesDataFile() {
```

```
    for (int i = 0; i < file1Len; i++) {

      for (int x = 0; x < conL; x++) {
        System.out.print(rulebase.dataFile1[i].cond[x]);
      }
      System.out.print(" 7th bit: ");
      System.out.print(rulebase.dataFile1[i].out);
      System.out.println(" ");
    }
    return;
  }

  //<L1, U1>, <L2, U2> ... <[0],[1]>, <[2],[3]>...
  public void fitnessOfRulesInd(int indI) {
    population.individuals[indI].fitness = 0;
    for (int i = 0; i < file1Len; i++) { //Cycles through the amount of rules from data file e.g. 60
      for (int j = 0; j < numR; j++) {  //Cycles through amount of rules in rules from the individual e.g. 10

        if (((rulebase.conditions[j].cond[0] < rulebase.dataFile1[i].cond[0]) && (rulebase.dataFile1[i].cond[0] <
rulebase.conditions[j].cond[1]))
            && ((rulebase.conditions[j].cond[2] < rulebase.dataFile1[i].cond[1]) && (rulebase.dataFile1[i].cond[1] <
rulebase.conditions[j].cond[3]))
            && ((rulebase.conditions[j].cond[4] < rulebase.dataFile1[i].cond[2]) && (rulebase.dataFile1[i].cond[2] <
rulebase.conditions[j].cond[5]))
            && ((rulebase.conditions[j].cond[6] < rulebase.dataFile1[i].cond[3]) && (rulebase.dataFile1[i].cond[3] <
rulebase.conditions[j].cond[7]))
            && ((rulebase.conditions[j].cond[8] < rulebase.dataFile1[i].cond[4]) && (rulebase.dataFile1[i].cond[4] <
rulebase.conditions[j].cond[9]))
            && ((rulebase.conditions[j].cond[10] < rulebase.dataFile1[i].cond[5]) && (rulebase.dataFile1[i].cond[5] <
rulebase.conditions[j].cond[11]))
            && ((rulebase.conditions[j].cond[12] < rulebase.dataFile1[i].cond[6]) && (rulebase.dataFile1[i].cond[6] <
rulebase.conditions[j].cond[13]))
            && ((rulebase.conditions[j].cond[14] < rulebase.dataFile1[i].cond[7]) && (rulebase.dataFile1[i].cond[7] <
rulebase.conditions[j].cond[15]))
            && ((rulebase.conditions[j].cond[16] < rulebase.dataFile1[i].cond[8]) && (rulebase.dataFile1[i].cond[8] <
rulebase.conditions[j].cond[17]))
            && ((rulebase.conditions[j].cond[18] < rulebase.dataFile1[i].cond[9]) && (rulebase.dataFile1[i].cond[9] <
rulebase.conditions[j].cond[19]))) { //Matches cond
          if (rulebase.dataFile1[i].out == rulebase.conditions[j].out) { //Matches output
            //If all 5 bits are the same and the output is the same then fitness ++
            population.individuals[indI].fitness++;
          }
          break;
        }
      } //When the rule matches should we move on??? fitness is going above 10 it shouldnt?
    }
    //System.out.println("Individual fitness based on rules: " + population.individuals[indI].fitness);
  }

  public void offSpringToRulebase(int indI) {
    int out = 0;
    int k = 0;
    for (int i = 0; i < numR; i++) {
      for (int j = 0; j < ruleLength; j++) {
        rulebase.conditions[i].cond[j] = population.offspring[indI].gene[k++];
      }
      rulebase.conditions[i].out = population.offspring[indI].gene[k++];
    }
  }

  public void fitnessOfRulesOffspring(int indI) {
    population.offspring[indI].fitness = 0;
    for (int i = 0; i < file1Len; i++) { //Cycles through the amount of rules from data file
      for (int j = 0; j < numR; j++) {  //Cycles through amount of rules in rules from the individual

        if (((rulebase.conditions[j].cond[0] < rulebase.dataFile1[i].cond[0]) && (rulebase.dataFile1[i].cond[0] <
rulebase.conditions[j].cond[1]))
            && ((rulebase.conditions[j].cond[2] < rulebase.dataFile1[i].cond[1]) && (rulebase.dataFile1[i].cond[1] <
rulebase.conditions[j].cond[3]))
```

```
                && ((rulebase.conditions[j].cond[4] < rulebase.dataFile1[i].cond[2]) && (rulebase.dataFile1[i].cond[2] <
rulebase.conditions[j].cond[5]))
                && ((rulebase.conditions[j].cond[6] < rulebase.dataFile1[i].cond[3]) && (rulebase.dataFile1[i].cond[3] <
rulebase.conditions[j].cond[7]))
                && ((rulebase.conditions[j].cond[8] < rulebase.dataFile1[i].cond[4]) && (rulebase.dataFile1[i].cond[4] <
rulebase.conditions[j].cond[9]))
                && ((rulebase.conditions[j].cond[10] < rulebase.dataFile1[i].cond[5]) && (rulebase.dataFile1[i].cond[5] <
rulebase.conditions[j].cond[11]))
                && ((rulebase.conditions[j].cond[12] < rulebase.dataFile1[i].cond[6]) && (rulebase.dataFile1[i].cond[6] <
rulebase.conditions[j].cond[13]))
                && ((rulebase.conditions[j].cond[14] < rulebase.dataFile1[i].cond[7]) && (rulebase.dataFile1[i].cond[7] <
rulebase.conditions[j].cond[15]))
                && ((rulebase.conditions[j].cond[16] < rulebase.dataFile1[i].cond[8]) && (rulebase.dataFile1[i].cond[8] <
rulebase.conditions[j].cond[17]))
                && ((rulebase.conditions[j].cond[18] < rulebase.dataFile1[i].cond[9]) && (rulebase.dataFile1[i].cond[9] <
rulebase.conditions[j].cond[19]))) { //Matches cond
              if (rulebase.dataFile1[i].out == rulebase.conditions[j].out) { //Matches output
                 //If all 5 bits are the same and the output is the same then fitness ++
                 population.offspring[indI].fitness++;
              }
              break; // note it is important to get the next data item after a match
           }
        }
     }
  }

  public void bestParentToRulebase(int indI) {
     int out = 0;
     int k = 0;
     for (int i = 0; i < numR; i++) {
        for (int j = 0; j < ruleLength; j++) {
           rulebase.conditions[i].cond[j] = population.bestParent[indI].gene[k++];
        }
        rulebase.conditions[i].out = population.bestParent[indI].gene[k++];
     }
  }

  public void fitnessOfBestParent(int indI) {
     population.bestParent[indI].fitness = 0;
     for (int i = 0; i < file1Len; i++) { //Cycles through the amount of rules from data file
        for (int j = 0; j < numR; j++) {  //Cycles through amount of rules in rules from the individual

           if (((rulebase.conditions[j].cond[0] < rulebase.dataFile1[i].cond[0]) && (rulebase.dataFile1[i].cond[0] <
rulebase.conditions[j].cond[1]))
                && ((rulebase.conditions[j].cond[2] < rulebase.dataFile1[i].cond[1]) && (rulebase.dataFile1[i].cond[1] <
rulebase.conditions[j].cond[3]))
                && ((rulebase.conditions[j].cond[4] < rulebase.dataFile1[i].cond[2]) && (rulebase.dataFile1[i].cond[2] <
rulebase.conditions[j].cond[5]))
                && ((rulebase.conditions[j].cond[6] < rulebase.dataFile1[i].cond[3]) && (rulebase.dataFile1[i].cond[3] <
rulebase.conditions[j].cond[7]))
                && ((rulebase.conditions[j].cond[8] < rulebase.dataFile1[i].cond[4]) && (rulebase.dataFile1[i].cond[4] <
rulebase.conditions[j].cond[9]))
                && ((rulebase.conditions[j].cond[10] < rulebase.dataFile1[i].cond[5]) && (rulebase.dataFile1[i].cond[5] <
rulebase.conditions[j].cond[11]))
                && ((rulebase.conditions[j].cond[12] < rulebase.dataFile1[i].cond[6]) && (rulebase.dataFile1[i].cond[6] <
rulebase.conditions[j].cond[13]))
                && ((rulebase.conditions[j].cond[14] < rulebase.dataFile1[i].cond[7]) && (rulebase.dataFile1[i].cond[7] <
rulebase.conditions[j].cond[15]))
                && ((rulebase.conditions[j].cond[16] < rulebase.dataFile1[i].cond[8]) && (rulebase.dataFile1[i].cond[8] <
rulebase.conditions[j].cond[17]))
                && ((rulebase.conditions[j].cond[18] < rulebase.dataFile1[i].cond[9]) && (rulebase.dataFile1[i].cond[9] <
rulebase.conditions[j].cond[19]))) { //Matches cond
              if (rulebase.dataFile1[i].out == rulebase.conditions[j].out) { //Matches output
                 //If all 5 bits are the same and the output is the same then fitness ++
                 population.bestParent[indI].fitness++;
              }
              break; // note it is important to get the next data item after a match
           }
        }
```

```
    }

  }

  public void fitnessUnseenData() throws FileNotFoundException {
    int index = 0;
    int topFitness = 0;
    for (int i = 0; i < P; i++) {
      if (population.individuals[i].fitness > topFitness) {
        index = i;
        topFitness = population.individuals[i].fitness;
      }


      //Gets index as the top fitness
      parentToRulebase(index);
      //Switch to data file 2 before calculating fitness
      loadDataFileUnseenData();
      //Calculate fitness
      file1Len = 1049; //Have to set it to 1049 because training was 700 entries
      fitnessOfRulesInd(index);
      System.out.println("Fitness on unseen dataset below: " + population.individuals[index].fitness);
    }
  }

    public void switchBoundsOffspring() {
    for (int i = 0; i < P; i++) {
      for (int x = 0; x < N;) {
        if ((x + 1) % 21 == 0) {
          //this is an action bit do nothing
          x++;
        } else {
          if (population.offspring[i].gene[x] > population.offspring[i].gene[x + 1]) {
            int temp = population.offspring[i].gene[x];
            population.offspring[i].gene[x] = population.offspring[i].gene[x + 1];
            population.offspring[i].gene[x + 1] = temp;
          }
          x = x + 2;
        }
      }
    }
  }
  public void switchBoundsInd() {
    for (int i = 0; i < P; i++) {
      for (int x = 0; x < N;) {
        if ((x + 1) % 21 == 0) {
          //this is an action bit do nothing
          x++;
        } else {
          if (population.individuals[i].gene[x] > population.individuals[i].gene[x + 1]) {
            int temp = population.individuals[i].gene[x];
            population.individuals[i].gene[x] = population.individuals[i].gene[x + 1];
            population.individuals[i].gene[x + 1] = temp;
          }
          x = x + 2;
        }
      }
    }
  }


} //end
```