Rule Induction of Binary and Real Number Datasets Using Genetic Algorithms and exploring Adaptive Parameters

Lewis Cummins - 16014766

1 INTRODUCTION

The assignment presented for this module was to research and apply artificial genetic evolutionary techniques to 'mine solutions' to the provided datasets. A dataset consists of either binary representations or real numbers, hereafter known as features, where these features match either a 0 or a 1, hereafter known as classes. A feature could be classed as a 0 or a 1, depending on the makeup of that feature, e.g. 110011 0 The 6-bit binary string being the feature, the 0 being the class. A solution is defined as a set of rules that matches the complete dataset, with the aim being that rules are generalised enough to match multiple data points, forming a subset that will pick out the valuable/important data points in order to match a feature to its class. The method of generalisation will be described in Section 3, as It varies depending on the type of data being fed into the algorithm. The aim of this report is to present the underlying challenges that exist with solving real world problems, whether it be what kind evolutionary algorithm to apply to a dataset in a given problem domain, or how the tuning of parameters for these different artificial learning techniques can be so difficult, especially due to the erratic nature of the data and the makeup of the problem space formed from the data varying dramatically; furthermore, in most cases, it is not straightforward to visualise. There is one thing that is undeniable, and it is that patterns always seem to emerge within data.

2. BACKGROUND RESEARCH

(Madni, Anwar and Shah, 2017) State that Data mining is known as Knowledge Discovery in Database (KDD) as well as the process which includes extracting the interesting, interpretable and useful information from the raw data. The methodology of extracting this information out of the vast amount of data being fed in, depends mostly on what we know about the data. For the case of this assignment, we know both the feature and classification label of that feature therefore we would use supervised learning techniques. If we did not know the class label, however, we would use unsupervised learning techniques. This is because In Data mining, the problem of unsupervised learning is that of trying to find hidden structure in unlabelled data (McNulty, 2015).

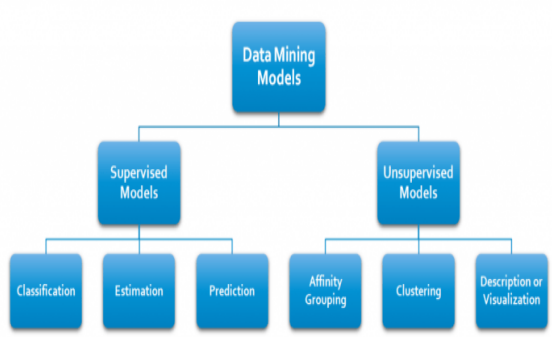


Figure 1: Showing different data mining techniques/models. (G, 2019)

With supervised learning, we can process the training data and produce a set of patterns, known as rules, which can be used for mapping new features to the correct class label (That is, as long as the new features are coming from the same environment). This process is known as rule induction. (Kotu and Deshpande, 2015) define rule induction as a data mining process of deducing if-then rules from a data set, stating that these symbolic decision rules explain an inherent relationship between the attributes and class labels in the data set. When extracting these rules, we can either directly extract the rules out of the dataset using an evolutionary learning technique and generalisation or use a Decision Tree model and extract the rules from that *(See Figure 2).* One example of a learning classifier system that uses Decision Trees is Random Forrest or Random Decision Forrest. The algorithm works by creating multiple decision trees during training, where each tree will

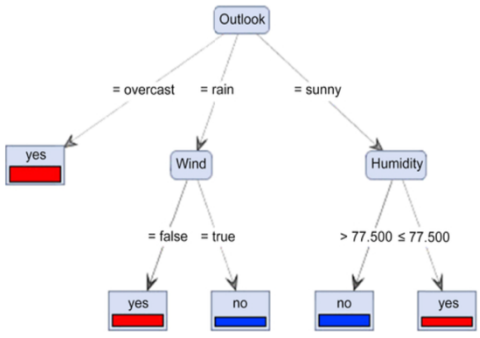


Figure 2: Example High Level Decision Tree Rule Induction (Kotu and Deshpande, 2015)

have a random subset of the features when forming questions and a random subset of the training data. To classify a new instance, each decision tree provides a classification for input data; random forest collects the classifications and chooses the most voted prediction as the result (Mao and Wang, 2012) (Also known as the mode). The approach I am going to take is

running the dataset(s) through a genetic algorithm to evolve an ordered set of rules that correctly maps features and classes to a generalised rule and a prediction.

Genetic algorithms (GAs) are search and optimization tools, which work differently compared to classical search and optimization methods (Kalyanmoy, 1999). They model the process of evolution, as a stochastic search, that implements selection, crossover and mutation and generates a new population over every iteration, known as a generation. A population consists of individuals, each with a chromosome and fitness. The 'chromosome' is a binary string representation of a solution in a problem space, and the 'fitness' is how well that solution fairs in the problem space. I will describe how the representation of the dataset and rules works in Section 3, where the algorithm and fitness function will be explained and the differences between binary numbers and real numbers evaluated.

When working with large amounts of data, one needs to be conscious of the ethical connotations of using this data. There may be sensitive information hidden within, depending on what environment it is derived from and what the numbers in the dataset represent. The central challenge that underlines ethical data mining is that there is no collective standard of data mining ethics to point to. For instance, the United States has no legal definition of personal data (Keary, 2019).

3 EXPERIMENTATION

As mentioned in *Section 2*, I am implementing a rule based genetic algorithm, which will learn to classify a dataset as a set of rules with predictions. A rule will look like this: 10#110. A prediction will be either 0 or 1 (predicting the class). The rule looks a lot like a binary string, which is exactly what it is representing. The # represents a wildcard, it does not care what bit exists in that position in the rule, as it is not pertinent to determining the class of the feature. For datasets 3 and 4, I will be using the same algorithm however the representation of the rules will be different, as the datasets consist of real numbers. Trying to come up with a rule that matches exact real numbers is futile; the possibilities are infinite. Instead, a bounds-based approach will be implemented. By that I mean a rule consists of bounds upon which a real number can reside in order for the rule to successfully predict the class. A rule will be a collection of tuples that look like (0.345123, 0.885143). This means, for this particular number at this index in the feature, the number can be between the lower and upper bound, where the lower bound is index [0] and upper bound index [1]. The basic pseudo code for a genetic algorithm looks like this.

```
Simple Genetic Algorithm ()
{
initialize population;
evaluate population ;
while convergence not achieved
        {
        scale population fitnesses ;
        select solutions for next population ;
        perform crossover and mutation ;
        evaluate population ;
        }
}
```

Figure 3: Basic GA Pseudo Code (Srinivas and Patnaik, 1994)

My implementation will follow in similar suit to the algorithm in *Figure 3*. I will implement 2 selection methods, notably Tournament (TS) and Roulette Wheel (RW) , as well as 1-point (1PC) and 2-point crossover (2PC) , coupled with a stochastic mutation function. The reasoning behind implementing multiple methods is it will allow me to compare performance of each permutation of methods, for example, compare how Tournament selection paired with 2-point crossover might compare to Tournament paired with 1 point crossover, and to discern why performance was improved/reduced based on the aforementioned methods as well as the current values of the parameters passed into the Genetic Algorithm.

Parameters:

P_M	**Probability of mutation per gene**
P_C	**Probability of crossover per 2 parents**
N	**Size of Population**
N_r	**Number of rules**
F_m	**Maximum Fitness**

To those familiar with genetic algorithms, these parameters will look familiar. At least all but N_r**,** which represents the number of rules we want to use to generalise the data set.

3.1 - Data Set 1

Dataset 1 consists of 60 binary numbers, each with a class of 0 or 1. Features with size 6 bits (maximum $2^6 = 64$) and classes being 1 bit. So, overall, a feature and class represent 7 bits of information. Extracting this information shows that our rules and predictions will be of length 7 bits (6 for rule 1 for prediction). When initialising our population, we will take N_r and multiply it by 7 (size of rule + size of prediction) to get our chromosome size. So, if we start off with $N_r = 15$ we would have a chromosome size of (7 * 15 = 105), where every 8^{th} index is the start of a new rule and every bit in a rule can be [0, 1, #] and a prediction [0, 1].

Once a population of size N has been initialised, we can now begin to evaluate the fitness of our individuals. We do this by iterating over all the individuals and for each individual, compare its rules in the chromosome to the dataset. We are looking for a rule to match the feature but also the prediction to match the class. If our rule matches the feature, we will break regardless of if the prediction matches or not. We are more interested in the rule matching the feature for now, mutation and crossover will resolve the incorrect prediction.

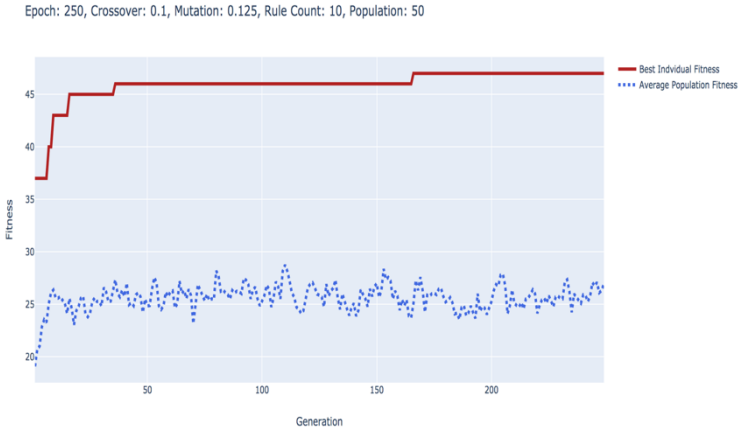
```
For Individual in Population:
   For Data in Dataset:
      For rule in individual.chromosome:
         if(rule.matches(data.feature)):
            If (data.class == rule.prediction)
                  individual.fitness ++
            Break
```

Figure 4: Example fitness function for binary rules.

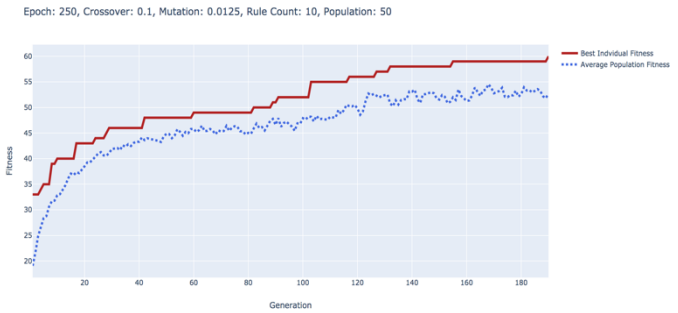
A rule is considered a match for a feature in the data set if all bits in the rule match all bits in the feature, where a hashtag is always a match, as it does not care what exists at that particular index. Once all have been evaluated, the selection process takes place to find the parents to generate offspring

from. These parents then undergo standard crossover on their chromosomes occurring with respect to P_C and mutation occurring with respect to P_m where mutation will mutate a feature gene to be [0, 1, #] and a prediction gene to be [0, 1]. This process continues until the desired number of generations has occurred, or F_m has been reached (Size of dataset, in this case F_m = 60).

On the first run of dataset 1, I set the parameters to: N = 50, P_m = 0.125, P_C = 0.9 and N_r = 10.



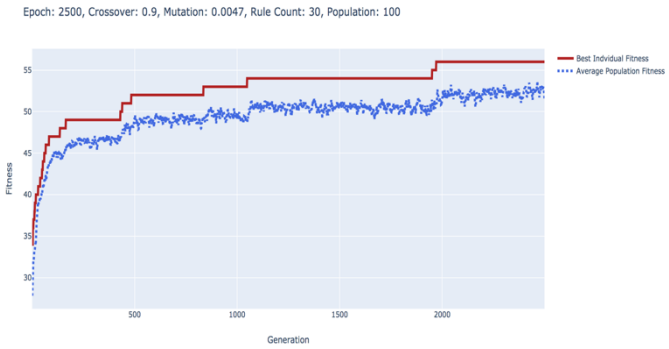
I Instantly recognized that there was a major gap between the average fitness and the best individual in my population. P_m was too high, causing too much mutation means that anything good that has been crossed over from 2 parents has a high probability of being over-mutated, causing good solutions to be discarded. Attempting to get a better fitness with a high mutation rate is very hard as you do not have the population working together to climb the search space. Instead, you are relying on random hyper mutation to hopefully produce a perfect solution, the chances being very low. After reducing the mutation rate to $1/10^{th}$ its original value, the parameters are now N = 50, P_m = 0.0125, P_C = 0.9 and N_r = 10.



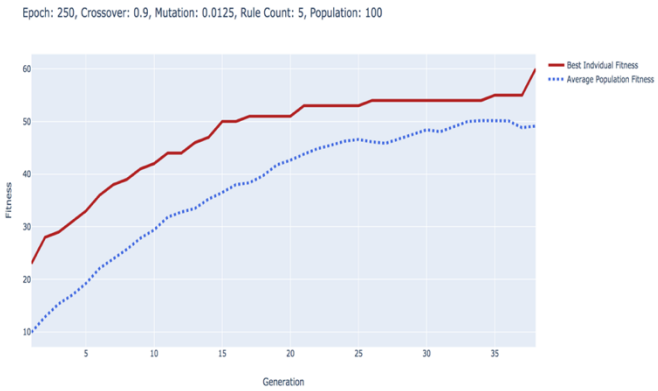
There is now a significant reduction in the distance between the average and best, which shows a much better fitness increase with relation to generations than the first run. This means it climbed the search space in a more directed manner and reached F_m for this dataset.

Rule	Prediction
[0, 0, '#', 1, '#', 0]	0
[0, '#', 0, '#', 1, 1]	1
[1, 1, 0, 1, 0, 0]	0
[0, 1, 0, '#', 1, '#']	1
[0, 0, '#', 0, '#', 1]	1
['#', 1, 0, '#', '#', '#']	0
['#', 0, '#', 0, '#', '#']	0
[0, 0, 1, '#', 0, '#']	1
[0, '#', 1, '#', 0, '#']	0
['#', '#', '#', '#', '#', '#']	1

What these represent are IF X THEN Y rules. So, if we were to pass in a feature of 001110, we would say that is classed as a 0, as it matches the first rule in the rule set. The ###### rule at the end is interesting but useful, it basically means 'If the feature does not match any of these rules, then it's class is a 1'. Which is what we want from binary classification, a small subset that represents valuable information, and if that valuable information isn't present, then we are sure that the feature is of a certain class. Of course, the less rules used then the rules explain a lot more about the data. In an effort to get the lowest rules possible, I tried setting N_r = 5 which provided the result:



As you can see, the algorithm plateaued in a local maximum, as there was not enough diversity. If ran for longer, then it would potentially move away from the local maximum, however it is not always guaranteed. I then Increased N = 50 to N = 100 and that resulted in:



And as the results show, by doubling the population it found an optimal solution in less than 40 generations. it was potentially too high of N to increase too, which could lead to overfitting the problem, however in this case it proved to really help the algorithm converge on the optimum solution.

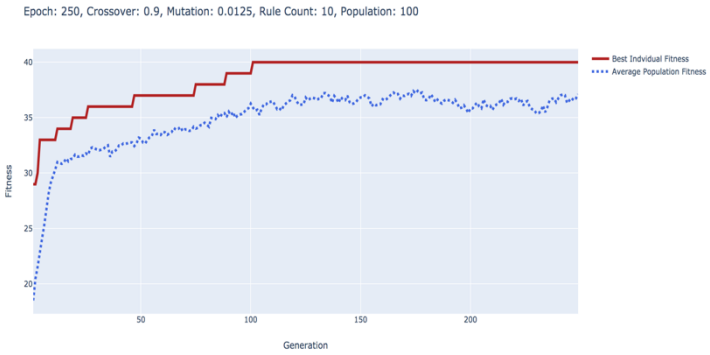
The final rule set generated from the parameters above.

Rule	Prediction
[1, 0, '#', 1, '#', '#']	1
[1, 1, 1, '#', '#', '#']	1
[0, 0, '#', '#', '#', 1]	1
[0, 1, '#', '#', 1, '#']	1
['#', '#', '#', '#', '#', '#']	0

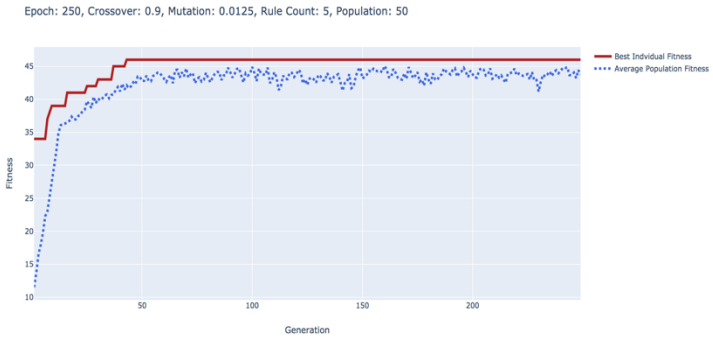
As I mentioned above, the less rules you can get to generalise a dataset, the more you can understand about the patterns formed and structure of the data. Here we have extracted the important information out of a set of data, albeit a simple one, it is still how basic data mining works. The most important datapoints in the feature are the first 2, we can see that the values in those 2, directly correlate to a definitive value in the last 4 points. For example, when the first 2 are 00, there needs to exist a 1 at the 6th bit, in order for the classification to be a 1. Anything that does not follow this pattern can automatically be classed as a 0.

3.2 - Data Set 2

Dataset 2 consists of 60 features, exactly the same as dataset 1. To begin experimentation with the dataset, I used identical parameters to the ones I used to solve dataset 1 the first time, with 10 rules. The results were surprising, as the graphs shows.



This visualisation suggests that the relationship between features and classes may be more complex than dataset 1. It achieved a much lower fitness than dataset one did in the same time, with the same parameters. Making the assumption that dataset 2 is more complex, we can only assume that more rules will be required to generate a complete ruleset for dataset 2, furthermore it will take longer to achieve a maximum fitness so I am increasing the generations from 250 to 2500 to give the algorithm time to search the space and generate an optimal solution. The revised permutation of parameters I have settled on is N = 100, P_m = 0. 0.0047, P_C = 0.9 and N_r = 30. TS + 1PC. (P_m = 1/(7 * N_r)). I ran this permutation of parameters 10 times, and the best algorithm produced a fitness of 56.



I am unsure if 30 rules are enough for dataset 2 to converge on F_m, however it is impossible to know without understanding the data. I attempted to implement a feature that will allow individuals to evolve how many rules they have in their chromosome, with the idea being all individuals will start with 60 rules, however they will be limited to using a subset of their rules. This limit will be mutated up and down for each individual, in the hope that eventually there comes a point that individuals are not getting any more fitness just because of the number of rules they have at their disposal, because past a certain point they will become moot, as there will be enough rules in front to cover the dataset. After multiple attempts, I couldn't get this particular feature to work, whilst I believe it would be helpful it will be something I discuss in the conclusion.

An observation I have made, based on dataset 1 and 2 now, is a clear correlation between P_m and N_r. if the mutation rate is too high then it will not converge and if it is too low, then we will cause the algorithm to prematurely converge. In an attempt to automate the tweaking of P_m and P_C I went in search of an algorithm that would tweak crossover and mutation constantly. I came across the idea of an adaptive genetic algorithm, that would calculate P_m and P_C based on the fitness of the individual or parents in relation to the best and average fitness. This paper was by (Srinivas and Patnaik, 1994). The section that I incorporated into my algorithm was the adaptive crossover and mutation. (Srinivas and Patnaik, 1994) define the two equations to determine P_m and P_C as:

$$p_m = k_2(f_{\max} - f)/(f_{\max} - \overline{f}), \quad f \geq \overline{f},$$
$$p_m = k_4, \quad f < \overline{f}$$

$$p_c = k_1(f_{\max} - f')/(f_{\max} - \overline{f}), \quad f' \geq \overline{f},$$
$$p_c = k_3, \quad f' < \overline{f}$$

Figure 5: image of the equations presented in (Srinivas and Patnaik, 1994)

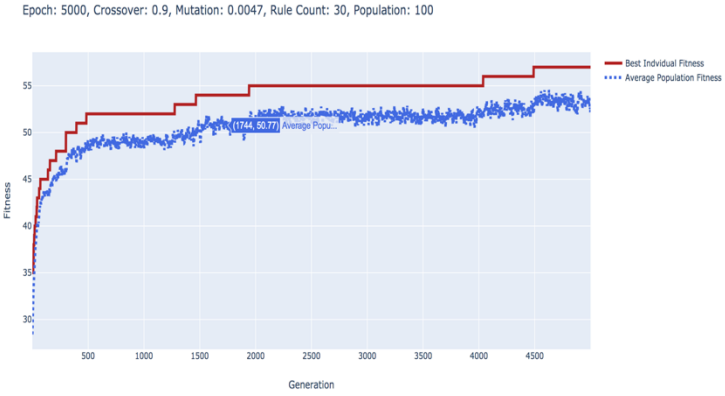
Where F_{max} Is the maximum fitness, f is the best individual for mutation and f' is the best of the 2 parents being used for crossover and $\overline{F}$ is the average fitness of the population at that point. So, for P_m where the current individual's fitness is greater than or equal to the max fitness, we take max fitness – individual fitness / max fitness – average fitness * some constant K_2, else P_m = K_4. For P_C where the best parent fitness is greater than or equal to the max fitness, we take max fitness –

parent fitness / max fitness – average fitness * some constant K_1, else P_C = K_3. The value of the constant as presented by (Srinivas and Patnaik, 1994) are:

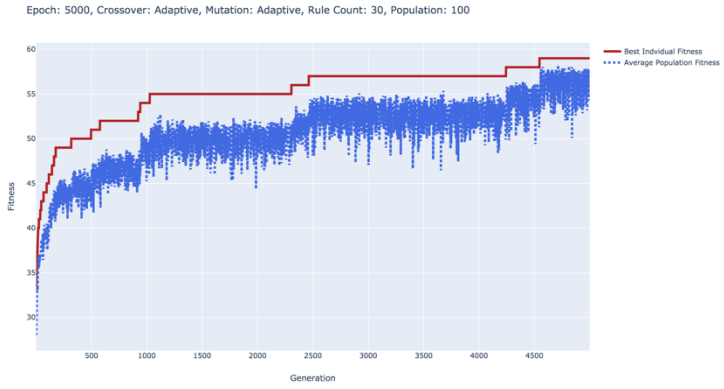
K_1 , K_3 = 1

K_2, K_4 = 0.5

To take into consideration that, when devising this equation (Srinivas and Patnaik, 1994) were dealing with a maximum fitness value of 1 in their algorithm. I have added an extra step in the equation which is to multiply the result by the inverse of the maximum fitness. That is if we are not using the constant values only**.** The whole point of using the constants when the fitness is less than the average is that we will always want to crossover these individuals as they are lower in the pack, and we also want to hypermutate the chromosome as it is subpar, so we can risk trying to find new solutions from these inferior individuals. Prior to running this, I ran dataset 2 under the same conditions again as last time N = 100, P_m = 0. 0.0047, P_C = 0.9 and N_r = 30. TS + 1PC and doubled the generations again to 5000 in a hope that it would solve. The results were only a fitness increase of 1 over double the generations.



Following this stagnation, I tried the approach described above. A genetic algorithm with adaptive properties P_m and P_C - N = 100 and N_r = 30. The results of these over 5 runs was 59 / 60. 1 off max fitness which is 98.3% rules generated.



Something interesting and different in the average population of the adaptive GA is the wider spread that the average population takes. This Is because of the hypermutation taking place, which could result in a bunch of individuals that have a bad fitness, which would bring the average fitness down.

I evaluated the rules against the dataset to find the erroneous data point. The value highlighted was 111000 with a class of 1. I ran it 20 more times, however got multiple datapoints that do not classify, I believe this was because I was trying to learn with too many rules. Whilst studying the data, I saw a pattern that was interesting and presented that 1 data point did not conform to the rest. It was following the parity bit method. A parity bit is added to the end of a binary string that confirms that the data has an odd or even number of ones (Techopedia.com, 2019). If the number of 1's is odd, then a 1 is used as the parity bit (In this case, the class), if the number of 1's is even, then a 0 is added to the end. The feature 000011 with class of 1 does not conform to this standard present throughout the entire dataset, and is therefore the erroneous datapoint causing the GA to struggle to reach F_m

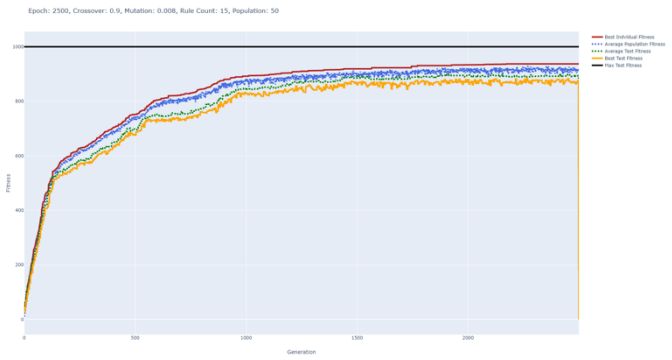
3.3 - Data Set 3

Dataset 3 is where the implementation must change to cater to real numbers. As I mentioned earlier, I will be implementing upper and lower bounds for each number in the feature. This now means that our rule will be double the size of our feature, while the class and prediction size remain the same. The overall rule + prediction size has increased from 7 to 13. Which now means our chromosome size has increased to 13 * N_R. The mutation will also have to change. Now working with real numbers, there is no point in just mutating one value to another, as you will completely disrupt the boundary. Instead, I will implement creep mutation. This works by essentially generating a number within a range and adding or subtracting it to the boundary selected for mutation. An example being

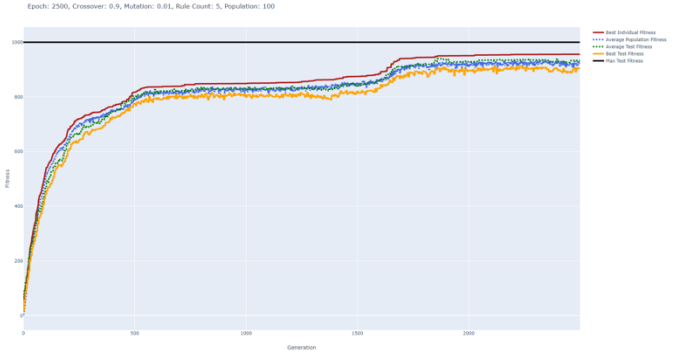
creep = +/- Random.randomRange(0.01, 0.1)

newBoundary = oldBoundary + creep

If we had -0.1 for example, we would add that to our boundary, for example 0.89 + (-0.1) = 0.79. Edge cases would have to be checked so that the 'creep' does not push the upper or lower bound over or under 1 or 0. The range will be denoted as M_{CL} and M_{CU} for the lower and upper creep boundary respectively. Dataset 3 has 2000 datapoints, which is a lot more to classify so will need a bit more time. I will also split the dataset in half, using 50% for training and 50% for testing the classified rules, aiming for above 90% classification of the dataset.
My First run with N_R = 15, N = 50 P_M = 0.008, P_C = 0.9 M_{CL} and M_{CU} as 0.01 and 0.1 gave me these results:



You can notice that the best individual fitness and the overall are quite close together. I have also incorporated checking the fitness of the rules against the test set every 5 generations, to ensure it is not overfitting too much, as you can see in the graph above, the green line is the average fitness of the population against the test set, and the orange line is the fitness of the best individual against the test set. The black line shows the maximum fitness for the test set. 15 rules look to be too many, as no discernible structure is present. The issue with classifying with more rules than is necessary, there is not much one can extrapolate from the rules, as there is less focus on the important information of the data. In a bid to reduce the classified rule count to better understand features and their relationship, I will reduce N_r to 5, I will also be increasing the mutation from 0.008 to 0.01. This is necessary as decreasing the rule count reduces the number of rules in the chromosome, which means a higher mutation rate will be needed to mutate at the same rate. Finally, I will modify M_{CU} from 0.1 to 0.15, to give the creep the ability for bigger jumps. There is a trade off with increasing this parameter. While we are able to explore the search space in bigger jumps, we may jump over an optimal solution. I would like to see the effect at which learning occurs with this modification.



The result from this parameter change gave me a better classification of 93% of the test set, which is above the threshold for what is considered an acceptable solution. The fact that $N_r = 5$ shows that the algorithm is as performant with less rules. There is a better correlation with this number of rules between the best individual of train and test than compared with more rules earlier. I would say this is the case because with more generalisation, the test features are hitting the rules with the correct label more often, as there are less rules to match, therefore more likely to not overfit. Real number rule sets are harder to interpret. There aren't just concrete rules stating what numbers have to be in what position to correctly identify a class. A few steps have to be taken to be able to interpret the data in a way where it is easier to explain. Firstly, any upper and lower bound pair with a difference of 0.9 or greater, I will convert into a '#', the same as was used for datasets 1 and 2. This is to indicate a 'Don't care' boundary. The other step will be to convert the boundary to a 1 or a 0, where a boundary is ~ 0 > 0.5 then it is a 0, if it is ~ 0.5 > 1 then it is a 1.

Feature	Class
(0, 0.494966), (0.007513, 0.498510), (0.4862709, 0.999739) # # #	1
# (0.491430, 0.997729) # (0.504498, 0.996295) # #	1
(0.488798, 0.993567) # # # (0.498617, 0.990273) #	1
(0.491229, 0.997595) (0.489636, 0.992434) # # # (0.490329, 0. 999642)	1
[#, #, #, #, #, #]	0

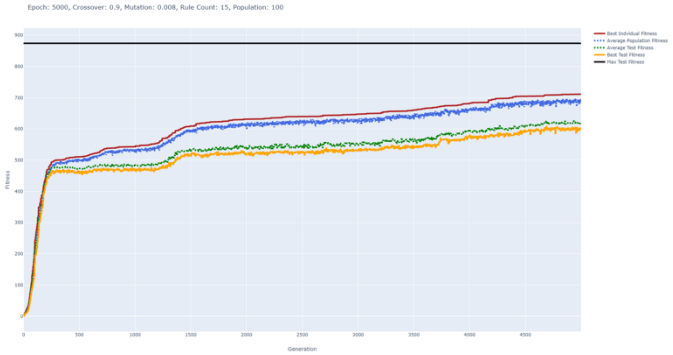
Above is the rule set before the bounds are converted to 1 or 0.

Feature	Class
[0, 0, 1, #, #, #]	1
[0, 1, #, 1, #, #]	1
[1, 0, #, #, 1, #]	1
[1, 1, #, #, #, 1]	1
[#, #, #, #, #, #]	0

This is the rule set now, after converting boundaries to 0's or 1's. What can be seen here is a clear separation between the first 2 and last 4 bits, just like dataset 1. However, the representation is in reverse, where 0, 0 in the first 2 would represent #, #, #, 1 in dataset 1, it now represents 1, #, #, #. Essentially the inverse of the function used.

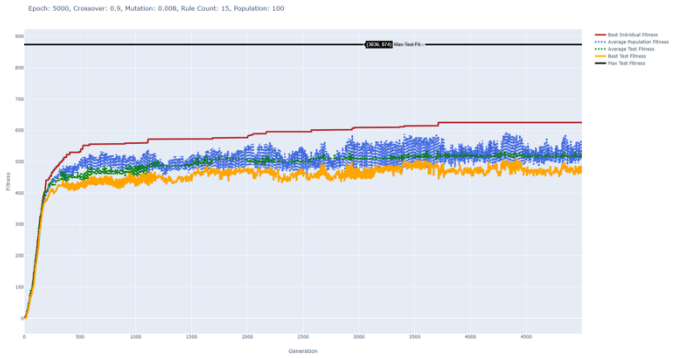
3.4 - Data Set 4

Dataset 4 is also a dataset full of real numbers. However, in contrast to dataset 3, it only has 1749 datapoints. It has 10 numbers that make up a feature, as opposed to 6 in the previous 3 datasets. With increased datapoints, it will be harder to classify but there may be columns that are completely useless when it comes to understanding what the data represents. Noise is always a problem in any classification problem, whether it be linear classification, image recognition etc. When interpreting large amounts of data, you need to be able to recognise noise, to avoid skewing your evaluation of the data. I split the training and test data by using even numbers for train and odd to test and used the same parameters for my first run on this dataset as I did for dataset 3, $N_R = 15$, $N = 100$ $P_M = 0.008$, $P_C = 0.9$ but M_{CL} and M_{CU} as 0.01 and 0.15.

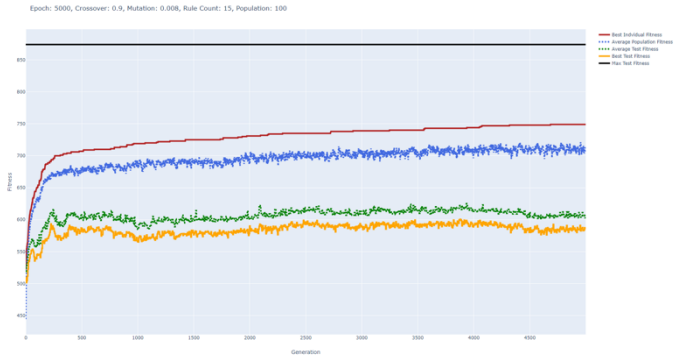


715 datapoints were classified using 15 rules during training, where the test set matched 603 against the finished rule set. Considering 600 is about 66% of the 875 datapoints in the training set, we can see there is quite a difference, and a fair bit of overfitting occurring. Averaging this over 5 runs, I got a mean test classification rate of 76%. This is not enough, especially with 15 rules. The target would be around 85-90% averaged. I implemented another feature which essentially rotates the index of rules in a chromosome each iteration. This means the first rule in the chromosome will be pushed to the back of the chromosome of the individual generated for the next generation. This allows for extreme rules that are full of boundaries on the edge of the search space to be at the back of the list of rules. The thought process behind introducing it means that there will not be stagnation in fitness if there is an all-encompassing rule that matches all of the datapoints. It didn't work that well, the graph on the next page shows the

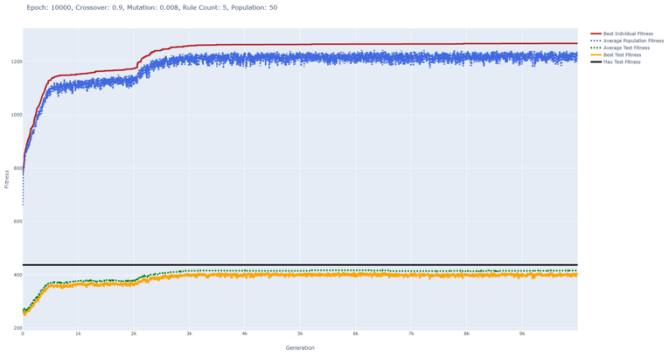
best fitness was 625 (71.5%) and the best test fitness was 490 (56%) of the test set. So, while it may stop extremely generic rules from staying at the front of the rule set, a better approach may be to move one to the back based on a random percentage, not every time. There is quite a large disparity occuring between the test and train set. This shows that the algorithm may need to train on a larger sample of the data, and test a smaller amount, to prevent overfitting and missing key features.



The genetic algorithm does struggle when classifying where the initial population are all random boundaries, it starts quite slow and stagnates usually around 650-700 fitness. As a different approach, to explore different methods of classification, I generated the initial population where the upper and lower bound were 0 and 1, seeding all individuals to encompass the entire search space, where it could then refine its boundary rules to be more specific. This does speed up the process, and using the same parameters as above, garnered the results below:



As you can see, the initial population starts with a much higher fitness; with extreme boundaries, it encases much of the search space. However, it does refine the boundaries to be more focused on a particular area. The overall percentage passed on train and test is 84%. This is still too many rules, though. I think the best approach for me would be to set N_R = 5, increase the generations to 10000 and initialise the population with the edge case boundaries, as it seems to learn a lot quicker and fit the test set better. One last modification I am going to make is to split 75% for training and 25% for testing. By splitting the data into odds [1] and evens [2], then take the evens of [1], and append that to [2], essentially halving half of the data. Running these parameters gives an overall fitness of 1240 and covers the test set by 91%.



This also removes the disparity between the best fitness of train and test. With the best fitness of training being 1268 (96%) and the best fitness of the training being (95.5%), there is an accurate sample size in the training to generalise it into rules. The one issue I have with this implementation would be using 10000 generations. The algorithm requires half that to get to an accurate representation of the dataset.

Following the same method of boundary reduction I took for dataset 3, whereby any upper and lower bound pair with a difference of 0.9 or greater is converted to a '#', only that I will use a difference of 0.8, as any important boundaries will usually have a max variance of 0.55. Seeding the initial population with max boundaries seems to produce rules that are still quite generic but lower than a 0.9 threshold.
The structure of dataset 4 is challenging to decode. One interesting thing about it, as I predicted at the start, is that there seems to be columns of data that are essentially just noise. They have no bearing on any rules that are generated. These columns are the 1st, 2nd, 5th and 9th columns of the dataset. After averaging my results over 5 runs of the GA, with 5 rules, every time these four columns for have had extreme boundaries that cover pretty much the entire space. The structure of Dataset 4 is that, when removing the noisy columns, the first and last numbers indicate what the 4 numbers in the middle columns will be.

3.5 – Weka Comparison

Weka is a piece of software with a collection of algorithms used for classification, regression and other modelling methodologies. I am going to use it to compare how the various algorithms classify dataset 4, so I could see how other algorithms faired without implementing them myself.

First, I converted dataset four into WEKA readable format (.arff). The two algorithms I wanted to compare performance with are Random Forrest, which I mentioned earlier and Decision Tables. Decision Tables are essentially a way to represent logic, in the form of actions and conditions, where an action is executed depending on the conditions that are met. I would definitely expect Random Forrest to classify dataset 4 with a high accuracy, as it is an algorithm really suited to this sort of data and problem space. Decision tables I am unsure about, as they may not be a complex enough method of representation for dataset 4. I will split the dataset 75% and 25% for test; the same amount I used for my final tuned algorithm (The selection method may be different, though).

The first algorithm I tested through WEKA was Random Forrest. The results it provided were:

```
=== Summary ===

Correctly Classified Instances         428               97.9405 %
Incorrectly Classified Instances         9                2.0595 %
Kappa statistic                          0.9577
Mean absolute error                      0.1374
Root mean squared error                  0.1779
Relative absolute error                 28.2936 %
Root relative squared error             35.9315 %
Total Number of Instances              437
```

A 97% classification. Slightly better than my algorithm, however the speed of which it did this is remarkably faster. My algorithm is quite slow and takes a lot longer. This could be refined by rewriting my algorithm in C, using structs and manual memory allocation. This only took the algorithm 100 iterations of training, too. This means my algorithm was not optimally tuned.

The Results from the decision tables was a lot worse in terms of managing to classify the data.

```
=== Summary ===

Correctly Classified Instances         322               73.6842 %
Incorrectly Classified Instances       115               26.3158 %
Kappa statistic                          0.4363
Mean absolute error                      0.3571
Root mean squared error                  0.4183
Relative absolute error                 73.5521 %
Root relative squared error             84.5039 %
Total Number of Instances              437
```

The results from the decision tables was a lot worse in terms of classifying the data. Having only classified 73% of the data, it is not an optimal algorithm for this data.

4 – Conclusions

Exploring the field of data mining, it's fascinating to see how various patterns can come from what is just on the surface large amounts of data. Without knowing the environment that the data is generated from I cannot execute the final step to try and evaluate why these patterns occur in an environment.
Trying to understand the structure of the datasets has changed the way I look at data. When trying to understand the structure I was looking at the feature as a whole, not how a third of the feature can influence the other two thirds, for example. It's not just plugging numbers into an algorithm and expecting results. You have to meticulously study these results, compare them to the environment and recognise the complex relationships that exist. Things I would have done differently, first, I would have liked to implement a neural network with backpropagation and use a genetic algorithm to tune the weights. I currently lack the complete knowledge on how to do that. Given more time, this would be something I would like to learn and explore the positives and negatives of this approach, to see if it would classify datasets 3 and 4 in a more performant manner. If I was to redo the assignment, I would write the code in C over python. Not that python isn't a great language, and has many use cases especially in AI, Big Data, data mining etc. But a smaller algorithm like this would have been better structured in C and the performance would have been improved, especially when training for a long time due to the fact that C is a lot faster and operates at a lower level.
The assignment has allowed me to learn a lot about the field of AI and data mining and has sparked an interest in me to want to learn more about it and explore more supervised learning techniques, and eventually look into unsupervised learning.

REFERENCES

G, M. (2019). *Data mining techniques*. [online] Data Engineer. Available at: https://www.data-engineer.in/analytics-2/data-mining-techniques/

Kalyanmoy, D. (1999). An introduction to genetic algorithms. *Sadhana*, [online] 24(4-5), pp.293-315. Available at: https://link.springer.com/article/10.1007/BF02823145

Keary, T. (2019). Finding a balance: what are the challenges of ethical data mining. [online] Information Age. Available at: https://www.information-age.com/data-mining-123481736/

Kotu, V. and Deshpande, B. (2015). *Predictive analytics and data mining*. Waltham, MA: Morgan Kaufmann, pp.88-89.

McNulty, E. (2015). *What's The Difference Between Supervised and Unsupervised Learning? - Dataconomy*. [online] Dataconomy. Available at: https://dataconomy.com/2015/01/whats-the-difference-between-supervised-and-unsupervised-learning/

Madni, H., Anwar, Z. and Shah, M. (2017). Data mining techniques and applications — A decade review. *2017 23rd International Conference on Automation and Computing (ICAC)*. [online] Available at: https://www.researchgate.net/publication/49616224_DATA_MINING_TECHNIQUES_AND_APPLICATIONS

Mao, W. and Wang, F. (2012). *Advances in intelligence and security informatics*. Oxford, UK: Academic Press, pp.91-102.

Srinivas, M. and Patnaik, L. (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems,* Man*, and Cybernetics*, [online] 24(4), pp.656-667. Available at: https://ieeexplore.ieee.org/abstract/document/286385

Techopedia.com. (2019). *What is a Parity Check? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/1803/parity-check

APPENDIX

SOURCE CODE

```
from classes.individual import Individual
import random
from copy import deepcopy

class GeneticAlgorithmBase():

    def __init__(self, population_size=10, chromosome_size=10,
        mutation_probability=0.01, crossover_probability=0.9):
        self._population_size = population_size
        self._chromosome_size = chromosome_size
        self._mutation_probabilty = mutation_probability
        self._crossover_probability = crossover_probability
        self._population = []
        self._generation = 0


    def _get_worst_individual(self, population, with_index=False):
        worst = population[0]
        idx = 0
        worst_idx = 0
        for individual in population:
            if (individual.fitness < worst.fitness):
                worst = individual
                worst_idx = idx
            idx +=1

        worst = deepcopy(worst)
        return (worst, worst_idx) if with_index else worst

    def _get_best_individual(self, population, with_index=False):
        best = population[0]
        idx = 0
        best_idx = 0
        for individual in population:
            if (individual.fitness > best.fitness):
                best = individual
                best_idx = idx
            idx +=1

        best = deepcopy(best)
        return (best, best_idx) if with_index else best
```

```python
from classes.base_ga import GeneticAlgorithmBase
from classes.rule import Rule
from classes.individual import Individual
from modules.data import initialise_data, draw_graph
from modules.selection import tournament_selection
from modules.crossover import one_point_crossover
from modules.mutation import rule_based_mutate
from copy import deepcopy
import random


class RuleBasedGeneticAlgorithm(GeneticAlgorithmBase):

    def __init__(self, dataset:list, population_size=10,
        mutation_probability=0.05, crossover_probability=1, rule_count=20):

        super().__init__(population_size=population_size,
            mutation_probability=mutation_probability,
            crossover_probability=crossover_probability)

        self._rule_count = rule_count
        self._data, self._feature_size, self._label_size = initialise_data(dataset)
        self._rule_size  = self._feature_size + self._label_size
        self._chromosome_size = self._rule_count * self._rule_size
        self._generation_info = []
        self._max_fitness = len(dataset)
        self.__initialise_population()


    def __initialise_population(self):
        for i in range(self._population_size):
            chromosome = self._generate_chromosome()
            individual = Individual(i, chromosome=chromosome,
                fitness=0)
            self._population.insert(i, individual)
        self._generation = 1


    def _generate_chromosome(self):
        chromosome = []
        while len(chromosome) < self._chromosome_size:
            for i in range(self._feature_size):
                chromosome.append(random.choice([0,1,'#']))

            for j in range(self._label_size):
                chromosome.append(random.choice([0,1]))

        return chromosome


    def _evaluate_fitness(self, individual):
        rules = Rule.generate_rules_from_chromosome(individual.chromosome,
            self._feature_size, self._rule_size)

        for data in self._data:
            for rule in rules:
```

```
                if(self._does_rule_match_data(rule.feature, data.feature)):
                    if(rule.label == int(data.label)):
                        individual.fitness += 1
                    break


    def _does_rule_match_data(self, rule, data):
        return all(a == str(b) or str(b) == '#' for a, b in zip(data, rule))


    def evolve(self, epochs=2000):
        while(self._generation < epochs):
            for individual in self._population:
                self._evaluate_fitness(individual)
            self._display_population_fitness(self._population)
            self._population = self._generate_offspring(self._population)
            best = self._get_best_individual(self._population)
            if(best.fitness == self._max_fitness):
                break
            self._reset_population_fitness(  self._population)
            self._generation +=1


        rules = Rule.generate_rules_from_chromosome(best.chromosome, self._feature_size,
            self._rule_size)

        print("[BEST RULE BASE]")
        for rule in rules:
            print("RULE: {} {}".format(rule.feature, rule.label))

        draw_graph(self._generation_info, epochs, self._crossover_probability,
            self._mutation_probabilty, self._rule_count, self._population_size)


    def _display_population_fitness(self, population):
        best_individual = super()._get_best_individual(population)
        average_fitness = round(self._get_average_fitness(population), 2)
        print("[Gen] {} [Average] {} [Best] {}".format(
            self._generation, average_fitness, best_individual.fitness))
        self._generation_info.append([self._generation, average_fitness,
best_individual.fitness])


    def _reset_population_fitness(self, population):
        for individual in population:
            individual.fitness = 0


    def _get_average_fitness(self, population):
        fitness_total = 0
        for individual in population:
            fitness_total += individual.fitness

        return fitness_total / self._population_size


    def _generate_offspring(self, population):
```

```
        best_individual = self._get_best_individual(population)
        offspring = tournament_selection(deepcopy(population))
        offspring = one_point_crossover(offspring, self._crossover_probability,
            self._chromosome_size)

        for individual in offspring:
            rule_based_mutate(individual,
                self._mutation_probabilty, self._rule_size)

        _, worst_idx = self._get_worst_individual(population=population,
            with_index=True)

        offspring[worst_idx] = best_individual
        return offspring


    def _test_rules(self, rule_set):
        total_rules_matched = 0
        for data in self._data:
            for rule in rule_set:
                    # Rule has matched
                    if(data.label == str(rule.prediction)):
                        # The rule is good
                        print("Rule {} {} matched {} {} successfully".format(rule.feature,
rule.prediction, data.feature, data.label))
                        total_rules_matched += 1
                        break
                    else:
                        print("Rule {} matched data {} but gave the wrong
result.".format(rule.feature, data.feature))
                        break
        print("[TOTAL DATA POINTS CORRECTLY MATCHED => {}]".format(total_rules_matched))
```

```
from classes.base_ga import GeneticAlgorithmBase
from modules.data import initialise_floating_point_data
from classes.individual import Individual
from classes.rule import Rule
from modules.selection import tournament_selection
from modules.crossover import one_point_crossover, two_point_crossover
from modules.mutation import floating_point_boundary_mutate
from modules.data import draw_graph
from copy import deepcopy
import random
import uuid


class RuleBasedFloatingPointGA(GeneticAlgorithmBase):

    def __init__(self, dataset, test_data, population_size=50,
                 mutation_probability=0.001, crossover_probability=0.8, rule_count=10,
                 seed_edge_cases=False):

        super().__init__(population_size=population_size,
                         mutation_probability=mutation_probability,
                         crossover_probability=crossover_probability)

        self._rule_count = rule_count
```

```
        (self._train_data, self._feature_size,
            self._label_size) = initialise_floating_point_data(dataset)
        self._test_data = initialise_floating_point_data(test_data)[0]
        self._rule_size = self._feature_size + self._label_size
        self._chromosome_size = self._rule_size * self._rule_count
        self._seed_edge_cases = seed_edge_cases
        self._initialise_population()
        self._generation_info = []
        self._max_fitness = len(self._train_data)

    def _initialise_population(self):
        for i in range(0, self._population_size):
            chromosome = self._generate_chromosome()
            self._population.insert(i,
                                    Individual(uuid.uuid1().hex, chromosome, 0))

    def _generate_chromosome(self):
        chromosome = []
        while len(chromosome) < self._chromosome_size:
            for i in range(self._feature_size):
                def gene(): return random.random()
                bounds = list((0, 1)) if self._seed_edge_cases else list(
                    (gene(), gene()))
                chromosome.append((min(bounds), max(bounds)))

            for j in range(self._label_size):
                chromosome.append(random.choice([0, 1]))

        return chromosome

    def evolve(self, epoch=1500):
        while(self._generation < epoch):
            for individual in self._population:
                self._evaluate_fitness(individual)
            self._display_population_fitness(self._population)
            self._population = self._generate_offspring(self._population)
            best = self._get_best_individual(self._population)
            if(best.fitness == self._max_fitness):
                break
            self._reset_population_fitness(self._population)

            if(self._generation != 0 and (self._generation + 1) % 5 == 0):
                self.evalute_test(self._population)
                self._reset_population_fitness(self._population)

            self._generation += 1

        rules = Rule.generate_rules_from_chromosome(best.chromosome, self._feature_size,
                                                    self._rule_size)

        self.evalute_test(self._population)
        print("[BEST RULE BASE]\n")
        for rule in rules:
            print("RULE: {} {}".format(rule.feature, rule.label))

        draw_graph(self._generation_info, epoch, self._crossover_probability,
```

```
                  self._mutation_probabilty, self._rule_count, self._population_size,
len(self._test_data))
        self.test_rules(rules)
        self.floating_point_to_rules(rules)

    def evalute_test(self, population):
        for individual in population:
            self._evaluate_fitness(individual, use_test=True)
        best_individual = self._get_best_individual(population)
        average_fitness = round(self._get_average_fitness(population), 2)
        print("[Gen] {} [Test Average] {} [Test Best] {}".format(
            self._generation, average_fitness, best_individual.fitness))
        for i in range(self._generation - 5, self._generation):
            self._generation_info[i][-1], self._generation_info[i][-2] = (
                average_fitness, best_individual.fitness)

    def _evaluate_fitness(self, individual, use_test=False):
        rules = Rule.generate_rules_from_chromosome(individual.chromosome,
                                                    self._feature_size, self._rule_size)

        dataset = self._test_data if use_test else self._train_data
        for data in dataset:
            for rule in rules:
                if(self._does_rule_match(data.features, rule.feature)):
                    if(data.prediction == rule.label):
                        individual.fitness += 1
                    break

    def _does_rule_match(self, data, rule):
        return all(b <= a <= c for a, (b, c) in zip(data, rule))

    def _display_population_fitness(self, population):
        best_individual = self._get_best_individual(population)
        average_fitness = round(self._get_average_fitness(population), 2)
        print("[Gen] {} [Average] {} [Best] {}".format(
            self._generation, average_fitness, best_individual.fitness))
        self._generation_info.append(
            [self._generation, average_fitness, best_individual.fitness, 0, 0])

    def _generate_offspring(self, population):
        best_individual = self._get_best_individual(population)
        offspring = tournament_selection(population)
        offspring = two_point_crossover(offspring, self._crossover_probability,
                                        self._chromosome_size)

        for individual in offspring:
            floating_point_boundary_mutate(individual,
                                           self._mutation_probabilty, self._rule_size)
            # individual.chromosome = (individual.chromosome[self._rule_size:] +
            #                          individual.chromosome[:self._rule_size])

        _, worst_idx = self._get_worst_individual(population=population,
                                                  with_index=True)

        offspring[worst_idx] = best_individual
        return offspring
```

```
    def _get_average_fitness(self, population):
        fitness_total = 0
        for individual in population:
            fitness_total += individual.fitness

        return fitness_total / self._population_size

    def _reset_population_fitness(self, population):
        for individual in population:
            individual.fitness = 0

    def __calculate_crossover_probability(self, best_parent_fitness, max_fitness,
                                          average_population_fitness):

        if(best_parent_fitness >= average_population_fitness):
            calc = ((max_fitness - best_parent_fitness) / (
                max_fitness - average_population_fitness))
            calc = calc * 1/max_fitness
            return (self.CROSSOVER_CONST_ONE * calc)
        else:
            return self.CROSSOVER_CONST_TWO

    def __calculate_mutation_probability(self, individual_fitness, max_fitness,
                                         average_population_fitness):

        if(individual_fitness >= average_population_fitness):
            calc = ((max_fitness - individual_fitness) /
                    (max_fitness - average_population_fitness))
            calc = calc * 1/max_fitness
            return(self.MUTATION_CONST_ONE * calc)
        else:
            return self.MUTATION_CONST_TWO

    def test_rules(self, finished_ruleset):
        overall_fitness = 0
        failed = 0
        for data in self._test_data:
            data_matched = False
            for rule in finished_ruleset:
                if(self._does_rule_match(data.features, rule.feature) and
                        data.prediction == rule.label):
                    overall_fitness += 1
                    data_matched = True
                    break
            if(not data_matched):
                failed += 1

        percentage_passed = (overall_fitness / len(self._train_data)) * 100
        print("TEST COMPLETE: PASSED: {} FAILED: {} | OVERALL: {}".format(
            overall_fitness, failed, percentage_passed))

    def floating_point_to_rules(self, finished_ruleset):

        for rule in finished_ruleset:

            print("RULE: {} ----- {}".format(
                [(lb, ub) if ((ub - lb) < 0.8) else ('#', '#')
```

```
                for (lb, ub) in rule.feature], rule.label))

from classes.rule_based_ga import RuleBasedGeneticAlgorithm
from modules.selection import tournament_selection
from modules.crossover import do_one_point_crossover, do_two_point_crossover
from modules.mutation import rule_based_mutate
from modules.data import draw_graph
from classes.rule import Rule
from copy import deepcopy
import random

class AdaptiveRuleBasedGeneticAlgorithm(RuleBasedGeneticAlgorithm):

    def __init__(self, dataset: list, population_size=10,
        mutation_probability=0.005, rule_count=10):

        super().__init__(dataset, population_size=population_size,
            mutation_probability = mutation_probability, rule_count=rule_count)

        self.CROSSOVER_CONST_ONE = 1
        self.CROSSOVER_CONST_TWO = 1
        self.MUTATION_CONST_ONE = 0.5
        self.MUTATION_CONST_TWO = 0.5


    def evolve(self, epoch=1000):

        while(self._generation < epoch):
            self.__evaluate_population_fitness(self._population)
            self._display_population_fitness(self._population)
            self._population = self._generate_offspring(self._population)
            best = self._get_best_individual(self._population)
            if(best.fitness == self._max_fitness):
                break
            self._reset_population_fitness(self._population)
            self._generation +=1

        rules = Rule.generate_rules_from_chromosome(best.chromosome, self._feature_size,
            self._rule_size)

        print("[BEST RULE BASE]")
        for rule in rules:
            rule.feature.append(rule.label)
            print("{}".format(rule.feature))

        draw_graph(self._generation_info, epoch, "Adaptive",
            "Adaptive", self._rule_count, self._population_size)


    def __evaluate_population_fitness(self, population):
          for individual in population:
                self._evaluate_fitness(individual)


    def _generate_offspring(self, population):
        best_individual = self._get_best_individual(population)
        average_fitness_of_population = self._get_average_fitness(population)
```

```
        offspring = tournament_selection(deepcopy(population))
        offspring = self.__adaptive_crossover(offspring, average_fitness_of_population)
        self.__adaptive_mutation(offspring, average_fitness_of_population)
        _, worst_idx = self._get_worst_individual(population=offspring,
            with_index=True)
        offspring[worst_idx] = best_individual

        return offspring

    def __adaptive_crossover(self, population, average_fitness):
        new_population = []
        for i in range(0, self._population_size, 2):
            parent_one = population[i]
            parent_two = population[i+1]
            best_parent = (parent_one if
                parent_one.fitness > parent_two.fitness else parent_two)
            crossover_probability = self.__calculate_crossover_probability(best_parent.fitness,
            self._max_fitness, average_fitness)
            chance = random.uniform(0, 1)
            if(chance <= crossover_probability):
                parent_one, parent_two = do_one_point_crossover(parent_one, parent_two)

            new_population.append(parent_one)
            new_population.append(parent_two)

        return new_population

    def __adaptive_mutation(self, population, average_fitness):
        for individual in population:
            mutation_probability = self.__calculate_mutation_probability(
                individual.fitness, self._max_fitness, average_fitness)
            rule_based_mutate(individual,
                mutation_probability, self._rule_size)


    def __calculate_crossover_probability(self, best_parent_fitness, max_fitness,
        average_population_fitness):

        if(best_parent_fitness >= average_population_fitness):
            calc = ((max_fitness - best_parent_fitness) / (
                max_fitness - average_population_fitness))
            calc = calc * 1/max_fitness
            return (self.CROSSOVER_CONST_ONE * calc)
        else:
            return self.CROSSOVER_CONST_TWO


    def __calculate_mutation_probability(self, individual_fitness, max_fitness,
        average_population_fitness):

        if(individual_fitness >= average_population_fitness):
            calc = ((max_fitness - individual_fitness) /
                (max_fitness - average_population_fitness))
            calc = calc * 1/max_fitness
            return(self.MUTATION_CONST_ONE * calc)
        else:
            return self.MUTATION_CONST_TWO
```

```
    def __adaptive_population(self, best_fitness,
        average_fitness, current_population_size):
        new_pop_size = (current_population_size + (best_fitness / average_fitness) - 1)
        print("[NEW POPULATION SIZE] = {}".format(new_pop_size))
```

```
class BinaryData:

    feature = None
    label = None

    def __init__(self, feature: list, label: str):
        self.feature = feature
        self.label = label
```

```
class FloatingPointData():

    def __init__(self, features, prediction):
        self.features = features
        self.prediction = prediction
```

```
class Individual():

    fitness = 0
    chromosome = None
    prediction = 0
    uid = None

    def __init__(self, _id, chromosome=[], fitness=0, prediction=None, ):
        '''
            An individual represents one candidate solution to a problem.

        '''
        self.uid = _id
        self.chromosome = chromosome
        self.fitness = fitness

    def __gt__(self, other):
        return self.fitness > other.fitness

    def __lt__(self, other):
        return self.fitness < other.fitness

    def __eq__(self, other):
        return self.fitness == other.fitness

    def __ge__(self, other):
        return self.fitness >= other.fitness

    def __le__(self, other):
        return self.fitness <= other.fitness

    def __str__(self):
        return 'Individual [ID: {} chromosome: {} fitness: {}]\n'.format(self.uid,
                                                                         self.chromosome,
self.fitness)
```

```
class Rule():

    def __init__(self, feature, label, match_count=0):
        self.feature = feature
        self.label = label
        self.match_count = match_count

    @staticmethod
    def generate_rules_from_chromosome(chromosome, feature_size,
        rule_size):

        rules = []

        for i in range(0, len(chromosome), rule_size):
            rule_stop = (i + feature_size)
            feature = chromosome[i:rule_stop]
            prediction = chromosome[rule_stop: rule_stop + 1][0]
            rules.append(Rule(feature=feature, label=prediction))

        return rules
```

SELECTION MODULE:

```
import random
from classes.individual import Individual

def tournament_selection(population):
    offspring = []
    select_parent = lambda x: random.randrange(x)

    population_size = len(population)
    for i in range (population_size):
        parent_one = population[select_parent(population_size)]
        parent_two = population[select_parent(population_size)]

        parent_one = Individual(_id=parent_one.uid,
                chromosome=parent_one.chromosome.copy(),
                fitness=parent_one.fitness)
        parent_two = Individual(_id=parent_two.uid,
                chromosome=parent_two.chromosome.copy(),
                fitness=parent_two.fitness)

        if(parent_one.fitness > parent_two.fitness):
            offspring.insert(i, parent_one)
        elif(parent_one.fitness < parent_two.fitness):
            offspring.insert(i, parent_two)
        else:
            choice = random.randint(0,1)
            if(choice):
                offspring.insert(i, parent_one)
            else:
                offspring.insert(i, parent_two)

    return offspring
```

MUTATION MODULE

```
import random
from classes.individual import Individual


def bit_flip_mutate(individual: Individual, mutation_probabilty: float):
    idx = -1
    for cell in individual.chromosome:
        idx += 1
        chance = random.uniform(0, 1)
        mutate = (chance <= mutation_probabilty)
        if(mutate):
            cell ^= 1
            individual.chromosome[idx] = cell
        else:
            continue


def rule_based_mutate(individual: Individual,
                      mutation_probability: float, rule_size: int):
    idx = -1
    for cell in individual.chromosome:
        idx += 1
        chance = random.uniform(0, 1)
        mutate = (chance <= mutation_probability)
        if(mutate):
            possibilities = [0, 1, '#']
            if ((idx + 1) % rule_size == 0):
                possibilities = [0, 1]
            possibilities.remove(cell)
            cell = random.choice(possibilities)

            individual.chromosome[idx] = cell


def floating_point_boundary_mutate(individual: Individual,
                                   mutation_probability: float, rule_size: int):
    idx = -1
    for gene in individual.chromosome:
        idx += 1

        bound_idx = 0
        chance = random.uniform(0, 1)
        mutate = (chance <= mutation_probability)

        if (not isinstance(gene, tuple)):
            if(mutate):
                gene ^= 1
                individual.chromosome[idx] = gene
            continue

        bounds = []
        for bound in gene:
            chance = random.uniform(0, 1)
            mutate = (chance <= mutation_probability)
```

```
            bounds.append(bound)
            if(mutate):
                operator = -1 if random.randint(0, 1) else 1
                mutation = random.uniform(0.01, 0.15) * operator
                bounds[bound_idx] = mutate_bound(bound, mutation)
            bound_idx += 1

        individual.chromosome[idx] = (min(bounds), max(bounds))


def mutate_bound(bound, mutation):
    if((bound + mutation) < 0 or (bound + mutation) > 1):
        mutation *= -1
    mutated_boundary = bound + mutation
    return mutated_boundary
```

DATA MODULE:

```
import plotly.graph_objects as go
from classes.floating_point_data import FloatingPointData
from classes.binary_data import BinaryData


def draw_graph(generation_info, epochs, crossover_probability,
               mutation_probabilty, rule_count, population_size, max_test_fitness=None):
    def flatten_point(l, i): return [sublist[i] for sublist in l]

    if (len(generation_info[0]) == 3):
        generation = flatten_point(generation_info, 0)
        averages = flatten_point(generation_info, 1)
        best = flatten_point(generation_info, 2)
    else:
        generation = flatten_point(generation_info, 0)
        averages = flatten_point(generation_info, 1)
        best = flatten_point(generation_info, 2)
        average_test = flatten_point(generation_info, 3)
        best_test = flatten_point(generation_info, 4)

    figure = go.Figure()

    figure.add_trace(go.Scatter(y=best, x=generation, name="Best Indvidual Fitness",
                                line=dict(color='firebrick', width=4)))
    figure.add_trace(go.Scatter(y=averages, x=generation, name="Average Population Fitness",
                                line=dict(color='royalblue', width=4, dash='dot')))
    figure.add_trace(go.Scatter(y=average_test, x=generation, name="Average Test Fitness",
                                line=dict(color='green', width=4, dash='dot')))
    figure.add_trace(go.Scatter(y=best_test, x=generation, name="Best Test Fitness",
                                line=dict(color='orange', width=4)))
    if(max_test_fitness):
        figure.add_trace(go.Scatter(y=([max_test_fitness] * epochs), x=generation, name="Max
Test Fitness",
                                    line=dict(color='black', width=4)))

    title = 'Epoch: {}, Crossover: {}, Mutation: {}, Rule Count: {}, Population: {} '.format(
        epochs, crossover_probability, mutation_probabilty,
        rule_count, population_size)
```

```
    figure.update_layout(title=title,
                         xaxis_title='Generation',
                         yaxis_title='Fitness')

    figure.show()


def initialise_data(dataset):
    data_props = dataset[0].split(" ")
    feature_size = len(data_props[0])
    prediction_size = len(data_props[1])
    data = []
    for row in dataset:
        data.append(BinaryData(*row.split(" ")))

    return(data, feature_size, prediction_size)


def initialise_floating_point_data(dataset):
    data_props = dataset[0].split(" ")
    amount_of_points = len(data_props) - 1
    prediction_size = len(data_props[-1])
    data = []
    for row in dataset:
        data_props = row.split(" ")
        features = list(map(lambda x: float(x), data_props[:-1]))
        data.append(FloatingPointData(features, int(data_props[-1])))

    return(data, amount_of_points, prediction_size)


def open_and_sanitize_data(data_set_location):
    with open(data_set_location, mode="r") as f:
        data = f.readlines()
        for i in range(len(data)):
            line = data[i]
            data[i] = line.strip("\n")
        return data
```

CROSSOVER MODULE

```
from classes.individual import Individual
import random
from copy import deepcopy


def do_one_point_crossover(parent_one: Individual, parent_two: Individual):
    crossover = lambda i1, i2, point: i1.chromosome[:point].copy() +
i2.chromosome[point:].copy()
    crossover_point = random.randrange(len(parent_one.chromosome))
    parent_one_chromosome = crossover(parent_one,
        parent_two, crossover_point)
    parent_two_chromosome = crossover(parent_two,
        parent_one, crossover_point)
    parent_one.chromosome = parent_one_chromosome
    parent_two.chromosome = parent_two_chromosome
```

```
    return (parent_one, parent_two)


def one_point_crossover(population: list,
    crossover_probability: float, chromosome_size: int):

    crossed_over_population = []
    population_size = len(population)
    for i in range(0, population_size, 2):
        chance = random.uniform(0, 1)
        do_crossover = (chance <= crossover_probability)
        parent_one = population[i]
        parent_two = population[(i + 1)]

        if(do_crossover):
            parent_one, parent_two = do_one_point_crossover(parent_one,
                parent_two)

        crossed_over_population.append(parent_one)
        crossed_over_population.append(parent_two)

    return crossed_over_population

def do_two_point_crossover(parent_one: Individual, parent_two: Individual):
    crossover_point_one = 0
    crossover_point_two = 0
    crossover = lambda i1, i2, p1, p2: (i1.chromosome[:p1].copy() +
        i2.chromosome[p1:p2].copy() + i1.chromosome[p2:].copy())

    chromosome_size = len(parent_two.chromosome)
    while((crossover_point_one) == (crossover_point_two)):
        crossover_point_one = random.randrange(chromosome_size)
        crossover_point_two = random.randrange(chromosome_size)

    biggest = (crossover_point_one if crossover_point_one > crossover_point_two
        else crossover_point_two)

    smallest = (crossover_point_one if crossover_point_two == biggest
        else  crossover_point_two)

    parent_one_chromosome = crossover(parent_one,
        parent_two, smallest, biggest)

    parent_two_chromosome = crossover(parent_two,
        parent_one, smallest, biggest)

    parent_one.chromosome = parent_one_chromosome
    parent_two.chromosome = parent_two_chromosome

    return (parent_one, parent_two)


def two_point_crossover(population: list, crossover_probability: float,
    chromosome_size, invert=False):
    crossed_over_population = []
    population_size = len(population)
    for i in range(0, population_size, 2):
```

```
        chance = random.uniform(0, 1)
        do_crossover = (chance <= crossover_probability)
        parent_one = population[i]
        parent_two = population[(i + 1)]

        if(do_crossover):
            parent_one, parent_two = do_two_point_crossover(parent_one,
                parent_two)

        crossed_over_population.append(parent_one)
        crossed_over_population.append(parent_two)

    return crossed_over_population


def uniform_crossover(population, crossover_probability):
    crossed_over_population = []
    for i in range(0, population, 2):
        parent_one = population[i]
        parent_two = population[i+1]
        chance = random.uniform(0, 1)
        do_crossover = (chance <= crossover_probability)

        if(do_crossover):
            parent_one_chromosome = uniform_inner(parent_one, parent_two)
            parent_two_chromosome = uniform_inner(parent_one, parent_two)
            parent_one.chromosome = parent_one_chromosome
            parent_two.chromosome = parent_two_chromosome

        crossed_over_population.append(parent_one)
        crossed_over_population.append(parent_two)

    return crossed_over_population


def uniform_inner(parent_one, parent_two):
    new = []
    for i in range(0, len(parent_one.chromosome)):
            chance = random.randint(0, 1)
            if(chance == 0):
                new.insert(i, parent_one.chromosome[i])
            else:
                new.insert(i, parent_two.chromosome[i])
    return new
```

ADVANCED MODULE:

```
def __cull_population(population, percent=20):
    # To sort the list in place...
    population.sort(key=lambda x: x.fitness, reverse=False)
    # To return a new list, use the sorted() built-in function...
    population = sorted(population, key=lambda x: x.fitness, reverse=False)
    population_length = len(population)
    cull_prop = int(population_length * (percent / 100))
    return population[cull_prop]
```

MAIN:

```
from classes.rule_based_ga import RuleBasedGeneticAlgorithm
from classes.adaptive_rule_based_ga import AdaptiveRuleBasedGeneticAlgorithm as ARBGA
from classes.rule_based_fp_ga import RuleBasedFloatingPointGA
from modules.data import open_and_sanitize_data, initialise_data

import plotly.express as px
import plotly.graph_objects as go
import numpy as np
import os
import sys


def main():
    dirname = os.path.dirname(__file__)
    sys.path.append(dirname)
    data_four(dirname)


def data_one(dirname):
    data_path = os.path.join(dirname, "data/dataset1.txt")
    data = open_and_sanitize_data(data_path)
    ga = RuleBasedGeneticAlgorithm(dataset=data, mutation_probability=0.0047,
                                   crossover_probability=0.9, population_size=100,
rule_count=30)
    ga.evolve(epochs=5000)


def data_two(dirname):
    data_path = os.path.join(dirname, "data/dataset2.txt")
    data = open_and_sanitize_data(data_path)
    ga = ARBGA(dataset=data, mutation_probability=0.005,
               population_size=100, rule_count=30)
    ga.evolve(epoch=5000)


def data_three(dirname):
    data_path = os.path.join(dirname, "data/dataset3.txt")
    data = open_and_sanitize_data(data_path)
    training_set = data[0::2]
    test_set = data[1::2]
    ga = RuleBasedFloatingPointGA(dataset=training_set, test_data=test_set,
                                  mutation_probability=0.01, crossover_probability=0.9,
                                  population_size=100, rule_count=5)
    ga.evolve(epoch=2500)


def data_four(dirname):
    data_path = os.path.join(dirname, "data/dataset4.txt")
    data = open_and_sanitize_data(data_path)
    training_set = data[0::2]
    test_set = data[1::2]
    training_set = training_set + test_set[0::2]
    test_set = test_set[1::2]
```

```
    # training_set = data[0::2]
    # test_set = data[1::2]
    ga = RuleBasedFloatingPointGA(dataset=training_set, test_data=test_set,
                                  mutation_probability=0.008, crossover_probability=0.9,
                                  population_size=50, rule_count=5, seed_edge_cases=True)
    ga.evolve(epoch=10000)


def evaluate_rules(dirname):
    data_two = os.path.join(dirname, "data/dataset2.txt")
    rules = os.path.join(dirname, "observations/data2rules.txt")
    data_two = open_and_sanitize_data(data_two)
    data_two = initialise_data(data_two)[0]

    with open(rules, mode="r") as f:
        data = f.readlines()
        for i in range(len(data)):
            line = data[i]
            data[i] = line.strip("\n")

    data = [(d[:len(d) - 1], d[-1]) for d in data]

    unmatched = []
    for bd in data_two:
        fm = False
        for (x, y) in data:
            if(match(bd.feature, x) and bd.label == y):
                fm = True
                break
        if(fm == False):
            unmatched.append(bd)
    print(unmatched)


def match(data, rule):
    return all((a == b) or b == '#' for a, b in zip(data, rule))


main()
```