UFCFY3-15-3: The Application of Genetic Algorithms in Classification Problems

Student Number: 15024059

December 2019

1 Introduction

As hardware has advanced in the modern day so have computational approaches of carrying out heuristically inspired search. It appears Holland (1975) first used the term Genetic Algorithm in their book 'Adaption in Natural and Artificial Systems', in said book Holland explored the application of the biological phenomena which is Evolution towards tackling problems which suffer from a combinatorial explosion of permutations typically associated with non deterministic polynomial (NP) time complexity problems. The evolutionary computing metaphor is a subset of machine learning which uses inspiration from the natural world to model problem solving. Pools of candidate solutions to a given problem are generated and compete against each other in a way which models Darwinian natural selection, to survive in a manner which models natural selection. This paper discusses the assignment issued as part of the Biocomputation module in which we were tasked with developing a Genetic Algorithm which seeks to model and classify various data sets, encoded either as binary data points or as real numbers.

2 Background Research

2.1 What is Data Mining?

In the modern day passive data generation has become an omnipresent fact of life. 'We swipe our way through the world, every swipe a record in data' (Witten and Frank 2005). Witten and Frank (2005) go on to state that true understanding of vast data sets tends to fall as the amount of data rises; this leads to valuable useful information being missed in the overall noise of the data set. Witten and Eibe (2005) state that Data Mining seeks to discover previously unknown patterns within related sets of data, once these data sets are discovered they can then be used to generalise and forecast future data.

2.2 Contemporary Applications of Data Mining

2.2.1 Crime Prediction

The BBC reported in early 2019 that they were are of at least 14 UK police forces which have made use of crime-prediction software or were planning on doing so, including Avon and Somerset Constabulary (Kelion 2019). When superficially considered the application of Data Mining in conjunction with crime prevention seems both well suited and appropriate however there are a variety of social issues that must thoroughly be considered first. As aforementioned Avon and Somerset Constabulary, who police Bristol, are one of the Police Forces which is using said crime-prediction software which provides functionality such as 'predictive mapping' to identify crime hot spots in which the police force then increases patrols in identified areas, another functionality which the software provides is 'individual risk assessment' which attempts to predict an individuals likelihood of recidivism. Both of the identified software functions present potential issues especially when considered in the application of policing Bristol. In 2017 the Centre on Dynamics and Ethnicity analysed data gathered from 2001 - 2011 to attempt "to identify patterns and drivers of ethnic inequalities in Bristol". The report (Finney and Lymperopoulou 2017) highlighted that Ethnic minorities in Bristol experience greater disadvantage compared to the rest of England and Wales as a whole in relation to both education and employment. The US nonprofit organisation ProPublica in 2016 investigated an algorithm that was in use called the 'Compass Recidivism Algorithm'. 'Across the nation, judges, probation and parole officers are increasingly using algorithms to assess a criminal defendant's likelihood of becoming a recidivist' Larson et al. (2016) remarked. ProPublica found that 'black defendants were far more likely than white defendants to be incorrectly judged to be at a higher risk of recidivism, while white defendants were more likely than black defendants to be incorrectly flagged as low risk'. While the argument that race relations in the US is completely different than race relations in the UK could be made, I would argue it doesn't negate the potential for 'opaque computer programs' (Kelion 2019) to perpetuate and further entrench inequality, this is not to say that Data Mining is unsuited to the area of crime-prediction rather the algorithms implemented must be designed and monitored to ensure they are not modelling our unconscious biases, without doing so could cause societal harm.

2.2.2 Cancer Classification

Bull (2008) discusses how Frenchay Hospital, Bristol, have employed Data Mining to discover 'patterns and trends' that are relevant to aiding the diagnosis of breast cancer. The process cancer diagnosis is vulnerable to human error due to the complexity and diversity of markers which present with the illness. Early diagnosis is essential to prevent further destruction and spread of cancer sells to other parts of the body (Masood 2012). Masood (2012) utilised a Naive Bayesian classifier to classify leukemia with a 95% accuracy. Results such as this clearly highlight the high degree of suitability that Data Mining algorithms posses in the domain of illness classification.

3 Experimentation

3.1 Data Set 1

3.1.1 Representation

Data set 1 presents us with 60 lines of binary data-points. Each datapoint can be split at a fixed point to create a condition and a classification. The condition being the first 6 bits of an individual data-point and the 7^{th}, final, bit represents the classification.

3.1.2 Overview of algorithm

For data set 1 I firstly set about creating various data structures to represent entities inside the solution. As I implemented my Genetic Algorithm in the Object-Oriented programming language Python(version 2.7) I created a Candidate Solution (in the source code the class is named 'Individual') class along with a class for representing Data Points (named 'Data') extracted from the data sets provided. My algorithm generates an initial population of 50 candidate solutions, each candidate solution was made up of 10 rules, each rule mirrors the binary encoding and structure of the data set provided to ensure it can model the data set accurately. A random seed allowed both the condition and classification of my prospective candidate solutions to be generated in a pseduo-random fashion. To assess the suitability of a potential solution to a particular problem Genetic Algorithms propose a fitness function which seeks to eradicate solutions which are not considered to be quality solutions within the specified problem domain, due to this every candidate solution requires a fitness metric to be calculated on a per epoch basis. Genetic Algorithms improve gradually over time due to mirroring Darwinian evolution in generating successive differing generations, to model time running I simply ensured the program looped X number of times, x is 100 for data set one (or until it hit the highest fitness possible). If we consider each individual candidate's condition and classification to be synonymous with a chromosome described in biology with each bit which makes up our candidates chromosome being considered as individual genes we can then consider how biology introduces variation and diversity in nature. "Genetic recombination is the primary source of genetic variation" Mayr (1970), Reeves and Rowe (2003) discuss how Holland presents a 'distinctive focus on recombination' where not only do individual solutions mutate, they also are able to exchange genes. Every generation of candidate solutions in my algorithm has their fitness assessed, a candidate solution will gain 1 fitness if it is able to model an individual data-point from the provided data set perfectly, with data-set one providing 60 data points a prospective candidate solution can have a maximum fitness of 60 which would mean it is able to model the data set perfectly. After fitness has been generated for each candidate solution in the population a Tournament selection takes place where 50 initial child solutions will generated by randomly selecting two candidates solutions and preserving the fittest out of the two; this takes place 50 times to generate the correct amount of potential children. Once we have 50 potential children the Genetic Operator Crossover takes place, the population is shuffled and and two at a time they probabilistically have crossover applied to them where if a dice roll hits a specified probability (75% chance of taking place) another dice roll takes place and two children are produced by recombining each chromosome (condition) at a certain point generated by the dice roll. The rationale for recombination is that the two children which have survived tournament selection have came from parents that have exhibited some degree of fitness relative to the rest of the candidate solution pool, by recombining them we are mirroring natural selection where individuals in nature who exhibit traits considered to be indicative of some degree of fitness or success are presented with mating opportunities, if said trait is hereditary it would then be potentially inherited by offspring of said parent. After recombination the last Genetic Operator potentially takes place, again depending upon a dice roll, my initial implementation for the dice roll contained a bug where mutation may never take place, for any gene, as I generated my mutation operator using the following formula to generate a floating point in the range of

```
min(1.0/<NUM POP>), max(1.0/<NUM GENE>)
```

I, however, later corrected it to move away from this programmatic method as it could generate a mutation rate of 0 depending on how I tweaked the variables in the solution. Mutation is implemented in my solution as a chance for each bit of a child to be inverted, the probability of this taking place is incredibly low. Mutation presents an opportunity of escaping a locally optimal solution and discovering a better solution as relying solely crossover can lead to premature convergence upon a solution which is not the global best. Finally my algorithm implements an element of elitism by ensuring the fittest candidate solution from a a previous generation is always preserved. This ensures the 'fittest' solution is never adapted in a destructive fashion. Once all of the offspring which survived the tournament process have been through this whole process I then re-assess the fitness and repeat this whole process for 99 more generations to allow the candidates to evolve better sets of rules.

3.1.3 Data

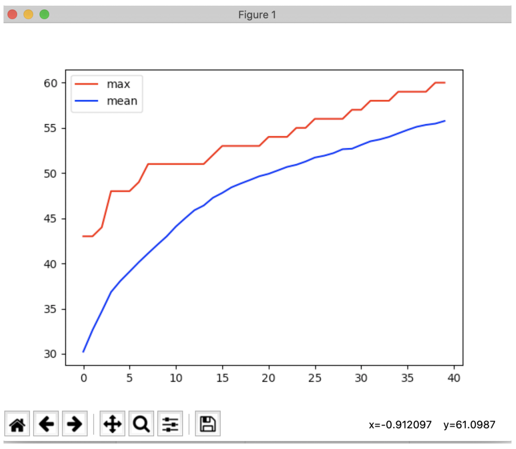

Figure 1: DS1 40 Epochs Wildcards Enabled, 500 Population Size, 75% crossover

```
2's represent wildcards
Fitness,
60

Condition, Action,
[1, 1, 0, 1, 2, 2], 0
[1, 2, 2, 1, 0, 2], 1
[2, 1, 0, 2, 0, 1], 0
[2, 0, 2, 1, 2, 1], 1
[0, 0, 1, 2, 2, 1], 1
[0, 0, 2, 2, 1, 2], 0
[0, 1, 2, 0, 1, 2], 1
[1, 2, 1, 1, 2, 0], 1
[1, 1, 0, 0, 2, 0], 0
[2, 1, 0, 0, 0, 1], 0
[1, 1, 0, 2, 2, 1], 0
[0, 1, 2, 1, 0, 2], 0
[0, 0, 2, 2, 0, 0], 0
[0, 1, 2, 0, 1, 2], 1
[1, 0, 2, 1, 0, 0], 1
[1, 1, 0, 2, 1, 1], 1
[0, 0, 0, 0, 2, 0], 1
[1, 1, 2, 0, 1, 2], 1
[0, 2, 0, 0, 1, 1], 1
[2, 0, 2, 0, 0, 0], 0
[1, 1, 1, 1, 2, 2], 1
[0, 0, 2, 1, 0, 1], 1
[2, 2, 2, 1, 2, 2], 1
[1, 1, 2, 0, 1, 0], 0
[1, 1, 1, 1, 0, 0], 0
[1, 0, 1, 1, 1, 1], 1
[0, 1, 2, 0, 2, 2], 0
[0, 0, 0, 0, 2, 0], 1
[0, 2, 2, 0, 1, 0], 0
[1, 2, 0, 0, 0, 2], 0
[0, 1, 0, 0, 2, 2], 0
[2, 1, 1, 0, 1, 1], 1
[1, 1, 1, 2, 0, 1], 1
[0, 2, 0, 0, 1, 1], 1
[1, 2, 0, 0, 1, 0], 0
[1, 1, 1, 1, 2, 2], 0
[0, 2, 2, 2, 2, 0], 0
[2, 2, 0, 0, 2, 0], 0
[0, 1, 0, 2, 1, 0], 1
[1, 0, 0, 0, 2, 1], 0
[2, 0, 2, 1, 2, 2], 1
[0, 0, 2, 2, 0, 1], 1
[0, 0, 2, 1, 2, 1], 0
[2, 2, 2, 0, 0, 0], 1
[2, 2, 2, 1, 2, 0], 1
[2, 0, 1, 0, 0, 2], 0
[1, 1, 1, 1, 2, 0], 0
[2, 2, 1, 2, 1, 2], 0
[0, 1, 1, 2, 0, 1], 1
[1, 1, 2, 0, 2, 0], 1
```

3.1.4 Analysis

Having a mutation rate which is too high cancels out the positive effect of crossover so to allow crossover to converge on a detected optima it's important that the mutation rate is kept low for data set 1.

3.2 Data Set 2

Dataset 2 has the same binary data point encoding as dataset 1 however it is immediately evident that the algorithm from dataset 1 must be adapted to be suitable. With wildcards being implemented in my dataset 1 I focused development time on adjusting my selection strategy, I switched my selection strategy from Tournament Selection to Rank Selection, my rationale for doing this was that I wanted to ensure that the least fit candidates had a much higher chance of being eliminated from the selection process whilst ensuring my search did not become biased towards premature convergence upon a local optima if a far fitter candidate solution is simultaneously present in the candidate pool.

3.2.1 Data

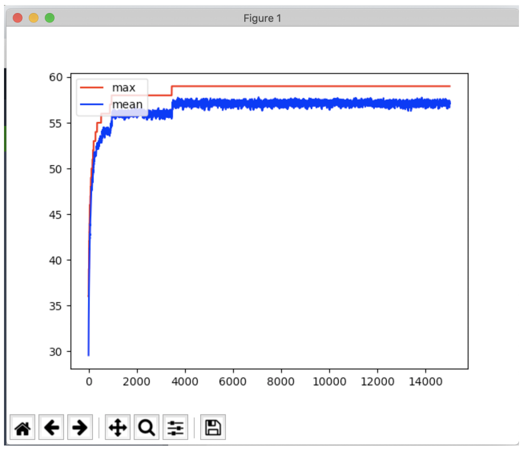

Figure 2: DS2 15000 Epochs Wildcards Enabled, 500 Population Size, 40% crossover, 0.002% Mutation, 50 Rules per candidate

```
2's represent wildcards
Fitness,
59

Condition, Action,
[1, 0, 0, 1, 1, 0], 1
[1, 1, 1, 0, 1, 0], 0
[0, 0, 0, 2, 1, 1], 1
[1, 0, 0, 0, 0, 0], 1
[0, 1, 0, 0, 0, 1], 0
[1, 1, 0, 0, 0, 0], 0
[0, 0, 2, 0, 1, 1], 1
[0, 2, 1, 1, 1, 0], 0
[0, 2, 1, 0, 1, 1], 0
[0, 1, 2, 1, 2, 0], 1
[1, 1, 2, 0, 2, 0], 1
[2, 1, 1, 2, 1, 2], 1
[1, 0, 2, 1, 1, 0], 0
[1, 0, 1, 1, 0, 1], 0
[0, 0, 0, 1, 0, 1], 0
[1, 1, 2, 1, 1, 1], 1
[0, 0, 1, 0, 0, 0], 1
[2, 0, 0, 1, 1, 2], 0
[1, 2, 1, 0, 0, 0], 0
[0, 1, 2, 0, 2, 1], 1
[2, 1, 1, 2, 0, 0], 0
[1, 2, 1, 0, 2, 0], 1
[2, 1, 0, 2, 1, 2], 0
[2, 0, 2, 0, 0, 2], 0
[0, 1, 1, 2, 1, 1], 0
[2, 0, 2, 1, 0, 1], 1
[0, 2, 0, 2, 2, 2], 1
[0, 2, 1, 2, 2, 0], 0
[2, 2, 0, 0, 2, 1], 1
[1, 2, 1, 1, 2, 2], 1
[0, 2, 0, 2, 0, 2], 0
[1, 2, 1, 0, 2, 0], 0
[2, 0, 2, 2, 2, 2], 0
[2, 1, 2, 2, 2, 1], 0
[2, 0, 1, 0, 1, 1], 0
[2, 1, 0, 1, 2, 0], 1
[2, 1, 1, 1, 0, 1], 1
[2, 0, 1, 2, 2, 0], 1
[0, 1, 2, 2, 0, 2], 1
[1, 0, 1, 2, 1, 2], 0
[1, 2, 0, 1, 1, 0], 1
[1, 0, 0, 0, 1, 2], 0
[0, 1, 2, 1, 0, 2], 0
[2, 1, 0, 0, 2, 2], 0
[2, 1, 0, 0, 0, 1], 0
[1, 1, 2, 2, 0, 2], 1
[2, 1, 1, 0, 2, 2], 0
[1, 0, 2, 1, 2, 2], 1
[1, 1, 1, 2, 2, 2], 0
[1, 1, 1, 2, 0, 0], 1
```

3.2.2 Analysis

As is evident by the number of epochs my Genetic Algorithm struggles to model this data set, I believe this is related to somewhere between 5% and 10 % of the data points present in the training set being unrelated to the general structure of the rest of the data set, this combined with my algorithms incredibly low mutation rate means it converges at a local optima instead of being able to represent the entire data set. Implementing an adaptive mutation rate would potentially have helped me escape from the local optima by ramping up the mutation rate when selection appears to be stagnating; this most likely would lead to a decrease in average fitness of the candidate pool however, as elitism is taking place on a per epoch basis, an overall decrease would have little negative effect and the increased rate of mutation could stimulate the selection process. The wildcards are useful in terms of visualising the data set, if you visually observe the rules of the fittest candidate you are able to gain an understanding of what can be abstracted away between data points (in data set 2's case if the bit is a wildcard) and what is an essential difference (i.e a bit that must be set in a particular state rather than wildcard state). This would be helpful if we were trying to classify in as few rules as possible and also if this was a real world problem it could potentially lead to insight in understanding the problem domain in potentially new ways.

3.3 Data Set 3

Data set 3 introduces genes which are encoded as floating point values with a corresponding integer classification, to accommodate this the overall algorithm had to be adapted to be tolerant towards processing of floating point numbers. Inside data set three we are presented with 2000 data points, this set was split in half with the first half being used as a training set and the second half being used to validate the effectiveness of the training process (test set). To accommodate floating point values the mutation operator had to change as simply generating a different integer value would no longer suffice due to the differing representation, in it's place a a function was created that either increases or decreases the float depending on a set of dice rolls and constraints.

3.3.1 Data

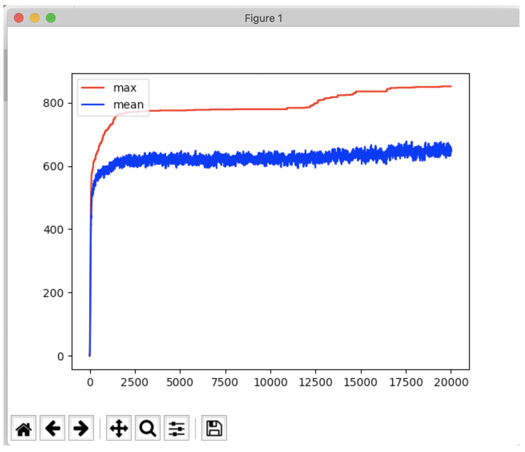

Figure 3: DS03 20000 Epochs Wildcards Enabled, 500 Population Size, 70% crossover, 35% Mutation Rate

```
Training Set Fitness,
Testing Set Fitness
851, 763

Condition, Action,
[0.504118, 0.999308,
0.000573, 0.998944,
0.00075, 0.996616,
0.001154, 0.983748,
0.000977, 0.549892,
0.000645, 0.664011],
0

[0.346115, 0.473144,
0.817922, 0.923132,
0.23426, 0.951071,
0.891161, 0.249946,
0.476905, 0.343093,
0.543996, 0.650446],
0

[0.675879, 0.093277,
0.684756, 0.179947,
0.699989, 0.520613,
0.663385, 0.846306,
0.515037, 0.759904,
0.409813, 0.453981],
1
```

```
[0.554285, 0.314721,
0.227386, 0.202532,
0.854683, 0.895975,
0.156837, 0.577916,
0.572332, 0.492848,
0.157994, 0.288687],
0

[0.002006, 0.508475,
0.000786, 0.997729,
0.000419, 0.501502,
0.000152, 0.663722,
0.000415, 0.993048,
0.006192, 0.99516],
0

[0.009191, 0.984751,
0.493405, 0.984593,
0.015704, 0.969923,
0.002636, 0.603667,
0.023366, 0.979507,
0.015615, 0.662055],
0

[0.309372, 0.373363,
0.221201, 0.445192,
0.223468, 0.713037,
0.536362, 0.146738,
0.190376, 0.817594,
0.289931, 0.20271],
1

[0.409147, 0.998785,
0.002832, 0.997279,
0.001836, 0.999829,
0.01022, 0.999312,
0.001043, 0.998366,
0.007263, 0.999848], 1

[0.182206, 0.687651,
0.608846, 0.918365,
0.816653, 0.963279,
0.151254, 0.289826,
0.920162, 0.048008,
0.452781, 0.305609], 0

[0.60541, 0.421331,
0.297571, 0.869029,
0.664816, 0.753723,
0.252259, 0.664705,
0.556933, 0.802227,
0.531256, 0.442194], 0
```

3.3.2 Data

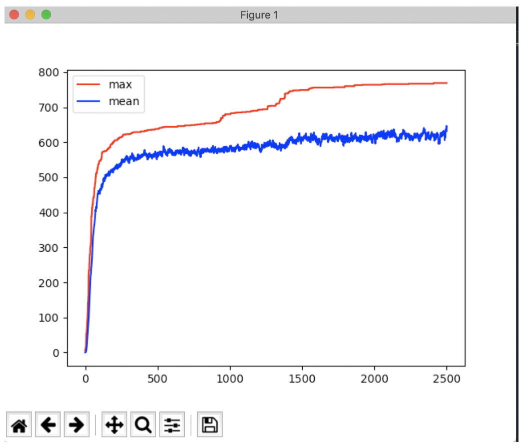

Figure 4: DS03 2500 Epochs Wildcards Enabled, 500 Population Size, 70% crossover, 35% Mutation Rate

```
Training Set Fitness,
Testing Set Fitness
769, 724

Condition, Action,
[0.001343, 0.995696,
0.00361, 0.494769,
0.508781, 0.999693,
0.010044, 0.998619,
0.011298, 0.994844,
0.007141, 0.991687],
1

[0.031828, 0.266297,
0.367112, 0.324484,
0.429601, 0.666833,
0.338205, 0.797362,
0.724772, 0.514571,
0.86219, 0.475549],
1

[0.60257, 0.600609,
0.246268, 0.594193,
0.289242, 0.521059,
0.134707, 0.408204,
0.592636, 0.636986,
0.676081, 0.891266],
1

[0.004039, 0.981625,
0.498746, 0.998904,
0.004961, 0.990436,
0.336957, 0.978327,
0.012593, 0.942566,
0.496064, 0.991133],
1

[0.779717, 0.483775,
0.720779, 0.305927,
0.911829, 0.135816,
0.809999, 0.520221,
0.439636, 0.414678,
0.743555, 0.223583],
1

[0.001404, 0.999025,
0.000388, 0.99723,
0.040404, 0.999933,
0.001878, 0.999809,
0.002506, 0.999729,
0.003324, 0.996632],
0

[0.518371, 0.44392,
0.154758, 0.206665,
0.314097, 0.852165,
0.420287, 0.211608,
0.3543, 0.0365,
0.747502, 0.426015],
0

[0.813998, 0.474895,
0.355001, 0.492279,
0.759109, 0.798344,
0.850846, 0.683264,
0.478374, 0.39247,
0.435029, 0.485331],
1

[0.226585, 0.517746,
0.546258, 0.49964,
0.786022, 0.069093,
0.595312, 0.793585,
0.452546, 0.171492,
0.930811, 0.537671],
1

[0.632965, 0.62371,
0.17567, 0.414505,
0.310684, 0.517185,
0.733614, 0.536165,
0.90765, 0.355542,
0.561686, 0.524704],
1
```

4 Conclusion

Genetic Algorithms are suited to a variety of tasks including classification. While being computationally expensive, genetic operators grant an impressive ability to traverse and sample various solutions in hope of finding a globally optimal solution or a solution which is 'good enough'.

References

Bull, Larry (2008). *Learning classifier systems in data mining.* English. Vol. 125. ISBN: 9783540789789;3540789782;

Finney, Farah Elahi andNissa and Kitty Lymperopoulou (Jan. 1, 2017). *Bristol: a city divided?* Centre on Dynamics of Ethnicity.

Holland, John H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press. ISBN: 0262082136.

Kelion, Leo (Feb. 4, 2019). "Crime prediction software 'adopted by 14 UK police forces'". In: *BBC.* URL: `https://www.bbc.co.uk/news/technology-47118229` (visited on 01/12/2019).

Larson, Jeff et al. (Mar. 23, 2016). "How We Analyzed the COMPAS Recidivism Algorithm". In: *Propublica.* URL: `https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm` (visited on 01/12/2019).

Masood, Dr Syed (Sept. 2012). "Cancer diagnosis using data mining technology". In: *Life Science Journal* 1, pp. 308–313.

Mayr, Ernst (1970). *Populations, Species, and Evolution.* Cambridge, MA, USA: Harvard University Press.

Reeves, Colin R. and Jonathan E. Rowe (2003). *Genetic algorithms: principles and perspectives : a guide to GA theory.* English. Vol. ORCS 20. London;Boston; Kluwer Academic. ISBN: 9781402072406;1402072406;

Witten, I. H. and Frank Eibe (2005). *Data mining: practical machine learning tools and techniques.* English. 2nd. Elsevier. ISBN: 9780120884070;0120884070;

Witten, I. H. and Eibe Frank (2005). *Data Mining : Practical Machine Learning Tools and Techniques, Second Edition.* Vol. 2nd ed. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann. ISBN: 9780120884070. URL: `http://search.ebscohost.com.ezproxy.uwe.ac.uk/login.aspx?direct=true&db=nlebk&AN=130260&site=ehost-live`.

5 Appendix

5.1 Data Set 1 Source Code

5.1.1 geneticAlgorithm.py

```
import const
import DataExtract
import random
# Matplotlib virtualenv hack
import matplotlib
matplotlib.use('TkAgg')

import matplotlib.pyplot as plt

from copy import deepcopy

random.seed(a=None)

MUTATION_PROB = random.uniform(1.0/const.NUM_POP, 1.0/const.NUM_GENE)

mean_fit = []
best_fit = []

fittest_cand = -1

class Individual:
    def __init__(self, gene):
        self.gene = gene
        self.fitness = 0
        self.ruleList = []

def main():
    # Create Population
    popPool = []
    max_num = 2
    first = False
    for _ in range(const.NUM_POP):
        count = 1
        gene = []
        for _ in range(const.NUM_GENE):
            if first:
                first = False
            if count == 7:
                max_num = 1
                count = 1
                first = True
            gene.append(random.randint(0, max_num))
            if count == 1:
                max_num = 2
            if not first:
                count += 1
        popPool.append(Individual(gene))

    # Selection
    for i in range(const.NUM_EPOCH):
        init = False
        if i == 0:
            init = True
        inspectPop(popPool, init=init)
        popPool = selection(popPool)
    inspectPop(popPool)
    write_csv()
    plot()

def plot():
    x = range(const.NUM_EPOCH + 1)
    plt.plot(x, best_fit, color='r', label='max')
    plt.plot(x, mean_fit, color='b', label='mean')
    plt.legend(loc="upper left")
    plt.show()


def write_csv():
    global fittest_cand
    with open("fit.csv", 'w') as f:
        f.write("Epoch,Best Candidate,Mean Fitness,\n")
        for i in range(const.NUM_EPOCH + 1):
            f.write("{e},{bc},{mf},\n".format(
                e=i, bc=best_fit[i], mf=mean_fit[i]))
    with open("best_cand.csv", 'w') as f:
        for rule in fittest_cand.ruleList:
            f.write(str(rule.condition) + " " + str(rule.classification))
            f.write("\n")
        f.write(str(fittest_cand.fitness))

def assessFitness(individual):
    def check_condition(rule, datapoint):
        match = True
        for i, dp in enumerate(datapoint):
            if rule[i] == dp or rule[i] == 2:
                pass
            else:
                match = False
                break
        return match
    # Create Rules
    k = 0
    fitness = 0
    individual.ruleList = []
    for i in range(const.NUM_RULES):
        tempRule = DataExtract.Data()
        for j in range(const.COND_LENGTH):
            tempRule.condition.append(individual.gene[k])
            k = k + 1
        tempRule.classification = individual.gene[k]
        k = k + 1
        individual.ruleList.append(tempRule)
    # Assess the fitness
    for dp in DataExtract.DataHelper.data_list:
        for rule in individual.ruleList:
            if check_condition(rule.condition, dp.condition):
                if rule.classification == dp.classification:
                    fitness = fitness + 1
                break
    individual.fitness = fitness



def inspectPop(p, init=False):
    global fittest_cand
    meanFitness = 0.0
    fittist = -1
    if init:
        # Generate current population fitness
        for ind in p:
            assessFitness(ind)
    for ind in p:
        # print "Genes:\n{g}\nFitness:\n{f}".format(g=ind.gene, f=ind.fitness)
        try:
            if ind.fitness > fittest_cand.fittness:
                fittest_cand = ind
        except Exception:
            fittest_cand = ind

        if ind.fitness > fittist:
            fittist = ind.fitness
        meanFitness = meanFitness + ind.fitness
    meanFitness = meanFitness / const.NUM_POP

    print "Fittest Candidate: {f}".format(f=fittist)
    print "Mean Fitness: {mf}".format(mf=meanFitness)
    mean_fit.append(meanFitness)
    best_fit.append(fittist)
```

```python
def selection(population):
    def crossover(cand_a, cand_b):
        child_a = None
        child_b = None
        if random.random() <= const.CROSSOVER_PROB:
            while True:
                cross_point = random.randint(1, const.NUM_GENE - 1)
                child_a = Individual(
                    cand_a.gene[:cross_point] + cand_b.gene[cross_point:])
                child_b = Individual(
                    cand_b.gene[:cross_point] + cand_a.gene[cross_point:])
                if vali_wc(child_a.gene) and vali_wc(child_b.gene):
                    break
            return (True, child_a, child_b)
        else:
            return (False, cand_a, cand_b)

    def vali_wc(ele):
        for e in ele:
            if e != 2:
                return True
        return False

    def mutate_offspring(child):
        first = False
        while True:
            new_gene = []
            count = 1
            b_range = 2
            for bit in child.gene:
                if first:
                    first = False
                if count == 7:
                    # Only allow action bit to be 1 and 0
                    count = 1
                    b_range = 1
                    first = True
                # Mutate bit
                if random.random() <= MUTATION_PROB:
                    new_gene.append(random.randint(0, b_range))
                else:
                    new_gene.append(bit)
                if count == 1:
                    b_range = 2
                if not first:
                    count = count + 1
            if vali_wc(new_gene):
                break
        child.gene = new_gene
        return child

    def swap_lowest(pop, prev_fittest):
        # Mutate original list
        pop = sorted(pop, key=lambda x: x.fitness, reverse=True)
        if prev_fittest.fitness > pop[-1].fitness:
            # Remove tail
            pop.pop()
            pop.append(prev_fittest)
        return pop

    shuffle_pop = lambda p : random.shuffle(p)
    offspring = []
    new_pop = []
    # Add the fittest candidate to the offspring
    fittest = deepcopy(sorted(population, key=lambda x: x.fitness, reverse=True)[0])

    for _ in range(const.NUM_POP):
        parentOne = population[random.randint(0, const.NUM_POP - 1)]
        parentTwo = population[random.randint(0, const.NUM_POP - 1)]
        if parentOne.fitness > parentTwo.fitness:
            offspring.append(parentOne)
        else:
            offspring.append(parentTwo)

    shuffle_pop(offspring)
    # Crossover
    for i in range(0, len(offspring), 2):
        temp_children = crossover(offspring[i], offspring[i+1])
        new_pop.append(mutate_offspring(temp_children[1]))
        new_pop.append(mutate_offspring(temp_children[2]))
    for ele in new_pop:
        assessFitness(ele)
    new_pop = swap_lowest(new_pop, fittest)
    return new_pop


if __name__ == "__main__":
    DataExtract.DataHelper.load_file_data(
        "/Users/User/Repos/BioComp/Biocomp_GeneticAlgorithm_CW/train/"
        "data1.txt")
    main()
```

5.1.2 DataExtract.py

```
class Data:
    def __init__(self, condition=None, classification=-1):
        if condition is None:
            condition = []
        self.condition = condition
        self.classification = classification

class DataHelper:
    '''
    Class which helps parse the text file of data and sort it into relevant lists.
    '''
    data_list = []
    '''
    Extracts raw data from file into Data objects
    :param dataset_path - String: Absolute path to where a dataset should be.
    '''
    @staticmethod
    def load_file_data(dataset_path):
        with open(dataset_path, 'r') as f:
            for line in f:
                tempCondition = []
                for bit in line.split()[0]:
                    tempCondition.append(int(bit))
                DataHelper.data_list.append(
                    Data(
                        tempCondition, int(line.split()[1])))
```

5.1.3 const.py

```
NUM_POP = 500 # P
NUM_GENE = 70  # N
NUM_EPOCH = 100
CROSSOVER_PROB = 0.75
NUM_RULES = 10
COND_LENGTH = 6
```

5.2 Data Set 2 Source Code

5.2.1 geneticAlgorithm.py

```
import const
import DataExtract
import random

# Matplotlib virtualenv hack
import matplotlib
matplotlib.use('TkAgg')

import matplotlib.pyplot as plt
```

```
from copy import deepcopy
from bisect import bisect

random.seed(a=None)

mean_fit = []
best_fit = []

class Individual:
    def __init__(self, gene):
        self.gene = gene
        self.fitness = 0
        self.ruleList = []

def main():
    # Create Population
    popPool = []
    max_num = 2
    first = False
    for _ in range(const.NUM_POP):
        count = 1
        gene = []
        for _ in range(const.NUM_GENE):
            if first:
                first = False
            if count == 7:
                max_num = 1
                count = 1
                first = True
            gene.append(random.randint(0, max_num))
            if count == 1:
                max_num = 2
            if not first:
                count += 1
        popPool.append(Individual(gene))
    global counter
    counter = 0
    # Selection
    for i in range(const.NUM_EPOCH):
        counter += 1
        print("Epoch {c}".format(c=str(counter)))
        init = False
        if i == 0:
            init = True
        inspectPop(popPool, init=init)
        popPool = selection(popPool)
        if fittest_individual.fitness >= (const.MAX_FIT):
            break
    inspectPop(popPool)
    print "Fittest Candidate"
    for e in fittest_individual.ruleList:
        print e.condition, e.classification
    print "Fittest: {f}".format(
        f=fittest_individual.fitness)

    write_csv()
    plot()

def plot():
    x = range(counter + 1)
    plt.plot(x, best_fit, color='r', label='max')
    plt.plot(x, mean_fit, color='b', label='mean')
    plt.legend(loc="upper left")
    plt.show()


def write_csv():
    with open("fit.csv", 'w') as f:
        f.write("Epoch,Best Candidate,Mean Fitness,\n")
        for i in range(counter + 1):
            f.write("{e},{bc},{mf},\n".format(
                e=i, bc=best_fit[i], mf=mean_fit[i]))
    with open("fittest_cand.csv", 'w') as f:
        f.write("Fitness,\n")
        f.write(str(fittest_individual.fitness))
        f.write("\n\n")
        f.write("Condition, Action,\n")
        for e in fittest_individual.ruleList:
            f.write("{cond}, {clas}\n".format(
                cond=e.condition,
                clas=e.classification))


def assessFitness(individual):
    def check_condition(rule, datapoint):
        match = True
        for i, dp in enumerate(datapoint):
            if rule[i] == dp or rule[i] == 2:
                pass
            else:
                match = False
                break
        return match

    # Create Rules
    k = 0
    fitness = 0
    individual.ruleList = []
    for i in range(const.NUM_RULES):
        tempRule = DataExtract.Data()
        for j in range(const.COND_LENGTH):
            tempRule.condition.append(individual.gene[k])
            k = k + 1
        tempRule.classification = individual.gene[k]
        k = k + 1
        individual.ruleList.append(tempRule)
    # Assess the fitness
    for dp in DataExtract.DataHelper.data_list:
        for rule in individual.ruleList:
            if check_condition(rule.condition, dp.condition):
                if rule.classification == dp.classification:
                    fitness = fitness + 1
                break
    individual.fitness = fitness


def inspectPop(p, init=False):
    global fittest_individual
    meanFitness = 0.0
    fittist = -1
    if init:
        # Generate current population fitness
        for ind in p:
            assessFitness(ind)
    for ind in p:
        # print "Genes:\n{g}\nFitness:\n{f}".format(g=ind.gene, f=ind.fitness)
        if ind.fitness > fittist:
            fittest_individual = ind
            fittist = ind.fitness
        meanFitness = meanFitness + ind.fitness
    meanFitness = meanFitness / const.NUM_POP

    print "Fittest Candidate: {f}".format(f=fittist)
    print "Mean Fitness: {mf}".format(mf=meanFitness)
    mean_fit.append(meanFitness)
    best_fit.append(fittist)

def selection(population):
    def crossover(cand_a, cand_b):
        child_a = None
        child_b = None
        if random.random() <= const.CROSSOVER_PROB:
            cross_point = random.randint(1, const.NUM_GENE - 1)
            child_a = Individual(
                cand_a.gene[:cross_point] + cand_b.gene[cross_point:])
            child_b = Individual(
                cand_b.gene[:cross_point] + cand_a.gene[cross_point:])
            return (True, child_a, child_b)
        else:
            return (False, cand_a, cand_b)

    def mutate_offspring(child):
        first = False
        new_gene = []
        count = 1
        b_range = 2
        for bit in child.gene:
            if first:
                first = False
            if count == 7:
                # Only allow action bit to be 1 and 0
                count = 1
                b_range = 1
                first = True
            # Mutate bit
            if random.random() <= const.MUTATION_PROB:
                new_gene.append(random.randint(0, b_range))
            else:
                new_gene.append(bit)
            if count == 1:
                b_range = 2
            if not first:
                count = count + 1
        child.gene = new_gene
        return child

    def swap_lowest(pop, prev_fittest):
        # Mutate original list
        pop = sorted(pop, key=lambda x: x.fitness, reverse=True)
        if prev_fittest.fitness > pop[-1].fitness:
            # Remove tail
            pop.pop()
            pop.append(prev_fittest)
        return pop

    shuffle_pop = lambda p : random.shuffle(p)
    offspring = []
    new_pop = []
    # Add the fittest candidate to the offspring
    population.sort(key=lambda x: x.fitness, reverse=False)
    # print [e.fitness for e in population]
```

```
    fittest = deepcopy(population[-1])
    z = 0
    rank_select = []
    max_rel_fit = 0
    rank_indexes = []

    for i in range(len(population)):
        z+= 1
        max_rel_fit += z
        rank_select.append(max_rel_fit)

    for i in range(len(population)):
        rank_indexes.append(bisect(rank_select, random.random() * max_rel_fit))
    [(offspring.append(population[index])) for index in rank_indexes]
    offspring.sort(key=lambda x: x.fitness, reverse=True)
    # print[e.fitness for e in offspring]
    shuffle_pop(offspring)
    # Crossover
    for i in range(0, len(offspring), 2):
        temp_children = crossover(offspring[i], offspring[i+1])
        new_pop.append(mutate_offspring(temp_children[1]))
        new_pop.append(mutate_offspring(temp_children[2]))
    for ele in new_pop:
        assessFitness(ele)
    new_pop = swap_lowest(new_pop, fittest)
    return new_pop


if __name__ == "__main__":
    DataExtract.DataHelper.load_file_data(
        "/Users/User/Repos/BioComp/Biocomp_GeneticAlgorithm_CW/train/"
        "data2.txt")
    main()
```

5.2.2 const.py

```
'''
 Condition Length + 1 action bit
 Rule = (010101  0)
 Num Genes = 1x Rule Length * Number of Rules to be generated to ensure
 enough genes present.
 '''
NUM_POP = 500
NUM_EPOCH = 5000
CROSSOVER_PROB = 0.4
NUM_RULES = 50
COND_LENGTH = 6
MUTATION_PROB = 0.0021
```

```
NUM_GENE = (COND_LENGTH + 1) * NUM_RULES
MAX_FIT = 60
```

5.3 Data Set 3 Source Code

5.3.1 GeneticAlgorithm.py

```
import const
import DataExtract
import random

# Matplotlib virtualenv hack
import matplotlib
matplotlib.use('TkAgg')

import matplotlib.pyplot as plt

from copy import deepcopy
from bisect import bisect

random.seed(a=None)

mean_fit = []
best_fit = []
counter = 0
test_fitness = None

class Individual:
    def __init__(self, gene):
        self.gene = gene
        self.fitness = 0
        self.ruleList = []


def main():
    global counter
    cout = 0
    # Create Population
    popPool = []
    for _ in range(const.NUM_POP):
        count = 1
        gene = []
        for _ in range(const.NUM_GENE):
            action = False
            if count == const.COND_LENGTH + 1:
                # Generating action bit (classification)
                # then reset count for new rule
                count = 1
                action = True
            if action:
                gene.append(random.randint(0, const.MAX_ACTION))
            else:
                # condition
                gene.append(
                    float(
                        '%.{p}f'.format(
                            p=const.FLOAT_PRECISION) % random.random()))
                cout += 1
                if count == 2:
                    cout = 0
            if not action:
                count += 1
        popPool.append(Individual(gene))

    # Selection
    for i in range(const.NUM_EPOCH):
        counter += 1
        print("Epoch {c}".format(c=str(counter)))
        init = False
        if i == 0:
            init = True
        inspectPop(popPool, init=init)
        popPool = selection(popPool)
        if fittest_individual.fitness >= (const.MAX_FIT):
            break

    inspectPop(popPool)
    print "Fittest Candidate"
    for e in fittest_individual.ruleList:
        print e.condition, e.classification
    print "Fittest: {f}".format(
        f=fittest_individual.fitness)

    # Run against test set
    global test_fitness
    test_fitness = compare_against_set(
        fittest_individual, DataExtract.DataHelper.test_data)
    print(
        "Produced a fitness of {f} on test set".format(f=str(test_fitness)))
    write_csv()
    plot()

def plot():
    x = range(counter + 1)
    plt.plot(x, best_fit, color='r', label='max')
    plt.plot(x, mean_fit, color='b', label='mean')
    plt.legend(loc="upper left")
    plt.show()


def write_csv():
    global test_fitness
    with open("fit.csv", ’w’) as f:
        f.write("Epoch,Best Candidate,Mean Fitness,\n")
        for i in range(counter + 1):
            f.write("{e},{bc},{mf},\n".format(
                e=i, bc=best_fit[i], mf=mean_fit[i]))
        f.write("\ntest fitness\n{f}".format(f=str(test_fitness)))
    with open("fittest_cand.csv", ’w’) as f:
        f.write("Fitness,\n")
        f.write(str(fittest_individual.fitness))
        f.write("\n\n")
        f.write("Condition, Action,\n")
        for e in fittest_individual.ruleList:
            f.write("{cond}, {clas}\n".format(
                cond=e.condition,
                clas=e.classification))


def compare_against_set(individual, dataset):
    # Assess the fitness
    fitness = 0
    for dp_rule in dataset:
        # Get one DP
        for trial_rule in individual.ruleList:
            # Run Trial DP down my gen rules and try ot match
            dp_matched = 0
            match = False
            lookup = None
            for i, dp_gene in enumerate(dp_rule.condition):
                if i == 0:
                    lookup = i
                else:
                    lookup = i * 2
                if (
                    trial_rule.condition[lookup] < dp_gene <
                        trial_rule.condition[lookup + 1]):
                    dp_matched += 1
                else:
                    break
                if dp_matched == const.TRAIN_COND_LENGTH:
                    # We’ve matched every gene in a DP with a generated rule
                    if dp_rule.classification == trial_rule.classification:
                        fitness += 1
                    match = True
                if match:
                    # We've matched with a created rule so stop looking for more rules
                    break
        return fitness

def assessFitness(individual):
    # Create Rules
    k = 0
    individual.ruleList = []
    for i in range(const.NUM_RULES):
        tempRule = DataExtract.Data()
        for j in range(const.COND_LENGTH):
            tempRule.condition.append(individual.gene[k])
            k = k + 1
        tempRule.classification = individual.gene[k]
        k = k + 1
        individual.ruleList.append(tempRule)

    individual.fitness = compare_against_set(
        individual, DataExtract.DataHelper.train_data)


def inspectPop(p, init=False):
    global fittest_individual
    meanFitness = 0.0
    fittist = -1
    if init:
        # Generate current population fitness
        for ind in p:
            assessFitness(ind)
    for ind in p:
        # print "Genes:\n{g}\nFitness:\n{f}".format(g=ind.gene, f=ind.fitness)
        if ind.fitness > fittist:
            fittest_individual = ind
            fittist = ind.fitness
        meanFitness = meanFitness + ind.fitness
    meanFitness = meanFitness / const.NUM_POP

    print "Fittest Candidate: {f}".format(f=fittist)
    print "Mean Fitness: {mf}".format(mf=meanFitness)
    mean_fit.append(meanFitness)
    best_fit.append(fittist)

def selection(population):
    def crossover(cand_a, cand_b):
        child_a = None
        child_b = None
        if random.random() <= const.CROSSOVER_PROB:
```

```
        cross_point = random.randint(1, const.NUM_GENE - 1)
        child_a = Individual(
            cand_a.gene[:cross_point] + cand_b.gene[cross_point:])
        child_b = Individual(
            cand_b.gene[:cross_point] + cand_a.gene[cross_point:])
        return (True, child_a, child_b)
    else:
        return (False, cand_a, cand_b)

def mutate_offspring(child):
    get_six_prec = lambda p: float('%.{p}f'.format(p=const.FLOAT_PRECISION) % p)
    action = False
    new_gene = []
    count = 0
    for bit in child.gene:
        count += 1
        if action:
            action = False
        if count == const.COND_LENGTH + 1:
            # Only allow action bit to be 1 and 0
            count = 0
            action = True
        # Mutate bit
        if random.random() <= const.MUTATION_PROB:
            if action:
                new_gene.append(1 - bit)
            else:
                mut_val = random.uniform(0, const.MUTATION_AMOUNT)
                if random.random() > 0.5:
                    # Try to add if possible
                    if get_six_prec(bit + mut_val) < 1:
                        new_gene.append(get_six_prec(bit + mut_val))
                    else:
                        new_gene.append(get_six_prec(bit - mut_val))
                else:
                    # try to minus if possible
                    if get_six_prec(bit - mut_val > 0):
                        new_gene.append(get_six_prec(bit - mut_val))
                    else:
                        new_gene.append(get_six_prec(bit + mut_val))
        else:
            new_gene.append(bit)
    child.gene = new_gene
    return child

def swap_lowest(pop, prev_fittest):
    # Mutate original list
    pop = sorted(pop, key=lambda x: x.fitness, reverse=True)
```

```
        if prev_fittest.fitness > pop[-1].fitness:
            # Remove tail
            pop.pop()
            pop.append(prev_fittest)
        return pop

    shuffle_pop = lambda p : random.shuffle(p)
    offspring = []
    new_pop = []
    # Add the fittest candidate to the offspring
    population.sort(key=lambda x: x.fitness, reverse=False)
    # print [e.fitness for e in population]
    fittest = deepcopy(population[-1])
    z = 0
    rank_select = []
    max_rel_fit = 0
    rank_indexes = []

    for i in range(len(population)):
        z+= 1
        max_rel_fit += z
        rank_select.append(max_rel_fit)

    for i in range(len(population)):
        rank_indexes.append(bisect(rank_select, random.random() * max_rel_fit))
    [(offspring.append(population[index])) for index in rank_indexes]
    shuffle_pop(offspring)
    # Crossover
    for i in range(0, len(offspring), 2):
        temp_children = crossover(offspring[i], offspring[i+1])
        new_pop.append(mutate_offspring(temp_children[1]))
        new_pop.append(mutate_offspring(temp_children[2]))
    for ele in new_pop:
        assessFitness(ele)
    new_pop = swap_lowest(new_pop, fittest)
    return new_pop


if __name__ == "__main__":
    DataExtract.DataHelper.load_file_data(
        "/Users/User/Repos/BioComp/Biocomp_GeneticAlgorithm_CW/train/"
        "data3.txt")
    main()
```

5.3.2 DataExtract.py

```
import const
```

```
class Data:
    def __init__(self, condition=None, classification=-1):
        if condition is None:
            condition = []
        self.condition = condition
        self.classification = classification

class DataHelper:
    '''
    Class which helps parse the text file of data and sort it into relevant lists.
    '''
    train_data = []
    test_data = []

    '''
    Extracts raw data from file into Data objects
    :param dataset_path - String: Absolute path to where a dataset should be.
    '''
    @staticmethod
    def load_file_data(dataset_path):
        temp_list = []
        with open(dataset_path, 'r') as f:
            for line in f:
                tempCondition = []
                lineSplit = line.split()
                i = 0
                for i in range(const.TRAIN_COND_LENGTH):
                    tempCondition.append(float(lineSplit[i]))
                temp_list.append(Data(
                        tempCondition, int(lineSplit[i+1])))
        DataHelper.train_data = temp_list[0:(len(temp_list) / 2)]
        DataHelper.test_data = temp_list[len(temp_list)/2:]
```

5.3.3 const.py

```
'''
NUM_POP = 50 # P
NUM_EPOCH = 15000
CROSSOVER_PROB = 0.75
NUM_RULES = 55
COND_LENGTH = 6
MUTATION_PROB = 0.02

'''
'''
 Condition Length + 1 action bit
 Rule = (010101  0)
 Num Genes = 1x Rule Length * Number of Rules to be generated to ensure
 enough genes present.
 '''
'''
NUM_GENE = (COND_LENGTH + 1) * NUM_RULES
# MUTATION_PROB = random.uniform(1.0/const.NUM_GENE, 1.0/const.NUM_POP,)
'''
NUM_POP = 500
NUM_EPOCH = 5000
CROSSOVER_PROB = 0.7
NUM_RULES = 10
COND_LENGTH = 12
TRAIN_COND_LENGTH = 6
# DS3 Fitness is assessing if each gene is between a certain range. ie.
# gene1 < testGene < gene2 (so cond = 6 * 2)
NUM_GENE = (COND_LENGTH + 1) * NUM_RULES
MAX_FIT = 1000
FLOAT_PRECISION = 6
MAX_ACTION = 1
MUTATION_PROB = 0.02
MUTATION_AMOUNT = 0.35
```