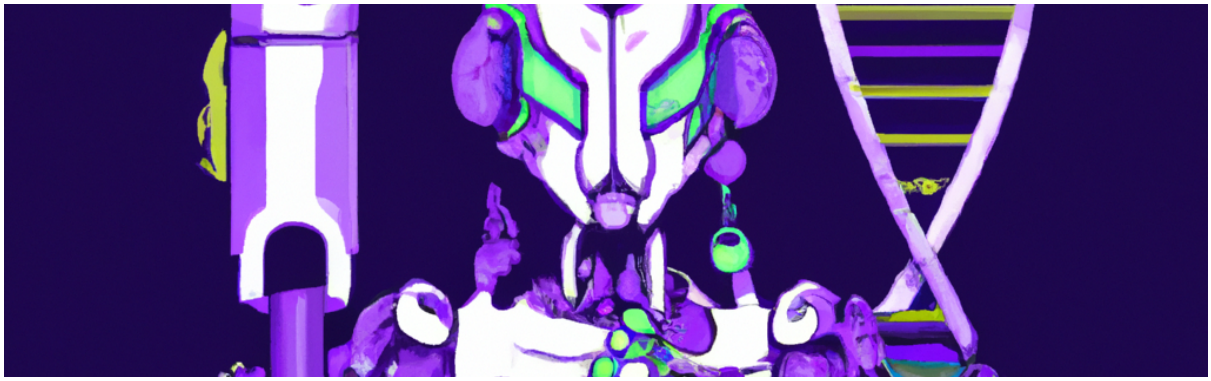


Evolutionary Computing for Optimization Problems

Genetic Algorithm for 0-1 Knapsack Problem

Objectives:

- Genetic algorithm (GA) representation;
- Design proper fitness function, crossover and mutation operators for GA to solve a given problem;
- Proper parameter setting for GA

Problem Domain:

The 0-1 knapsack problem is one of the most well known optimisation problems. Given a set of M items and a bag, each item i has a weight wi and a value vi, and the bag has a capacity Q.

The knapsack problem is to select a subset of items to be placed into the bag so that the total weight of the selected items does not exceed the capacity of the bag, and the total value of the selected items is maximised. The formal problem description can be written as follows:

$$max(v1x1 + v2x2 + \cdots + vMxM)(1)$$

$$s.t.(w1x1 + w2x2 + \cdots + wMxM \leq Q)(2)$$

$$xi \in 0, 1, i = 1, ..., M(3)$$

where xi = 1 means that item i is selected, and xi = 0 means that item i is not selected.
In this question, you are given the following three knapsack problem instances:

Dataset Evaluation:

- 10 269: with 10 items and bag capacity of 269. The optimal value is **295.**
- 23 10000: with 23 items and bag capacity of 10000. The optimal value is **9767.**
- 100 995: with 100 items and bag capacity of 995. The optimal value is **1514.**

The format of each file is as follows:

```
M Q
v1 w1
v2 w2
... ...
vM wM
```

In other words, in the first line, the first number is the number of items, and the second number is the bag capacity. From the second line, each line has two numbers, where the former is the value, and the latter is the weight of the item.

Solution:

To solve this 0-1 knapsack problem I developed a GA to solve the 0-1 knapsack problem and apply it to the provided three instances for evaluation.

This was based on the DEAP library example for this problem and utilized their library heavily.

A break down of the Genetic Operators is as follows:

Representation: For individual representation of the knapsack problem I choose a set of items (value: weight pairing) for my individual representation. This was because it is generally accepted that the best individual representation is one that is closest to the problem - here a set is as close to a bag of variables that we can realistically achieve in python.

Fitness Function: For my fitness/objective function, the individual (knapsack of items) is passed and the value and weights of each item is summed as these are the two values were are trying to optimize. We then check that the total weight and number of items does not exceed the maximum capacity of the bag, and return the total value and weight. If the capacity of the bag has been exceeded, then the maximum value is returned for the weight and minimum value for the value so this one will not be selected later.

Crossover: The Crossover operator used here is pretty simple and outputs 2 children. The first is the intersection of the the two sets and the second is the difference between them.

Mutation: The mutation operator is similarly basic and has a 50% chance of either adding or removing a random item.

Selection: NSGA-II from deap library

Population Size: For the population size I set 50 as I found more than this provided diminishing returns compared with the drastically increased run time.

Crossover and **Mutation** rates: For crossover and mutation rate I set them at complementary values of 0.6 and 0.4 respectively as I found this to be the optimal combination to replicate the optimal values as closely as possible

Results

The values recieved for running this algorithm 5 times for each dataset was as follows:

```
100 995
100_995- AVG: 747.94, STD: 416.65215276054914, MAX: 1431.0
100_995- AVG: 700.4, STD: 414.4134167712237, MAX: 1464.0
100_995- AVG: 752.78, STD: 411.8990793871722, MAX: 1419.0
100_995- AVG: 782.12, STD: 421.86413168222776, MAX: 1426.0
100_995- AVG: 717.24, STD: 442.305892341488, MAX: 1512.0

10 269
10_269- AVG: 170.32, STD: 93.67527742152674, MAX: 295.0
10_269- AVG: 124.28, STD: 99.60703589606509, MAX: 293.0
10_269- AVG: 143.24, STD: 97.69924462348725, MAX: 294.0
10_269- AVG: 136.5, STD: 96.23954488670444, MAX: 294.0
10_269- AVG: 169.38, STD: 92.65978415688222, MAX: 295.0

23 10000
23_10000- AVG: 5240.82, STD: 2902.4092936041943, MAX: 9750.0
23_10000- AVG: 5201.76, STD: 2894.243870581746, MAX: 9760.0
23_10000- AVG: 5245.66, STD: 2916.1456041151314, MAX: 9762.0
23_10000- AVG: 5372.42, STD: 2989.9078386465367, MAX: 9763.0
23_10000- AVG: 5221.42, STD: 2881.53370336007, MAX: 9760.0
```

The optimal values compared with my results were as follows:

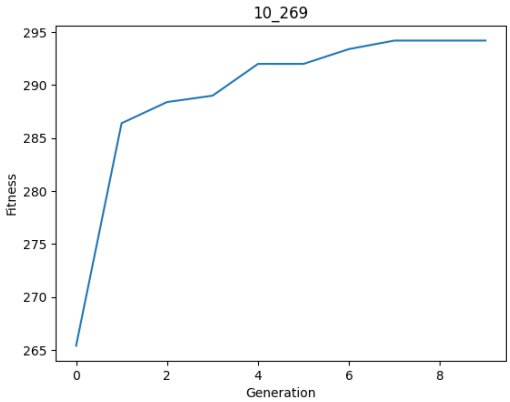
- 10 269: **295** (Optimal) **vs 295** (Mine).
- 23 10000: **9767** (Optimal) **vs 9763** (Mine).
- 100 995: **1514** (Optimal) **vs 1512** (Mine).

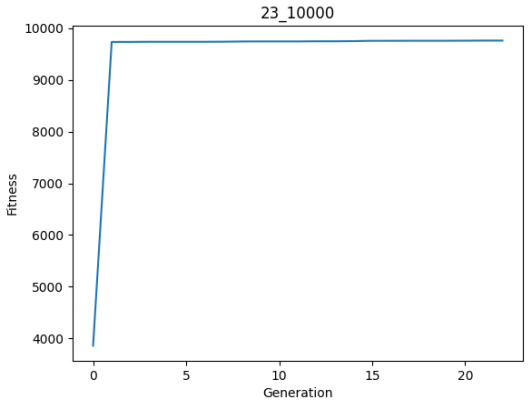
As can be seen I managed to get results that were almost the same as the optimal values. This was a result of my spending far too much time fiddling with the parameters for this question to find those that best worked. Originally I was nowhere near as close. It is also worth noting that I have only mentioned the best of each which, while over the course of the 5 runs one could be relied on to always be within a couple integer steps of the optimal value (usually the same one due to the nature of random

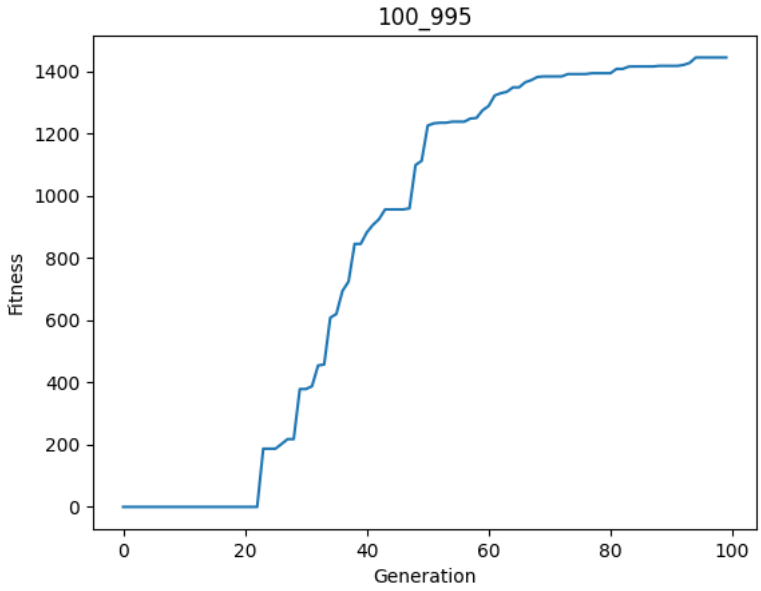
seeds) there is a significant amount of deviation between individual runs. On top of this the standard deviation is almost half of all the values which tells us that while the optimal value can be reached there is still a significant amount of spread.

Convergence:

A convergence curve for each dataset plotted with fitness against generations can be seen below:







Whats interesting to observe is the relative speed with which these datasets converged over generations. I also increased the number of generations as the size of the dataset increased. The most interesting is the 23_10000 dataset which converges incredibly fast and then stabilizes, yet is still the furthest from the optimal value by the end. this suggests to me that the algorithm got stuck in a local maxima.

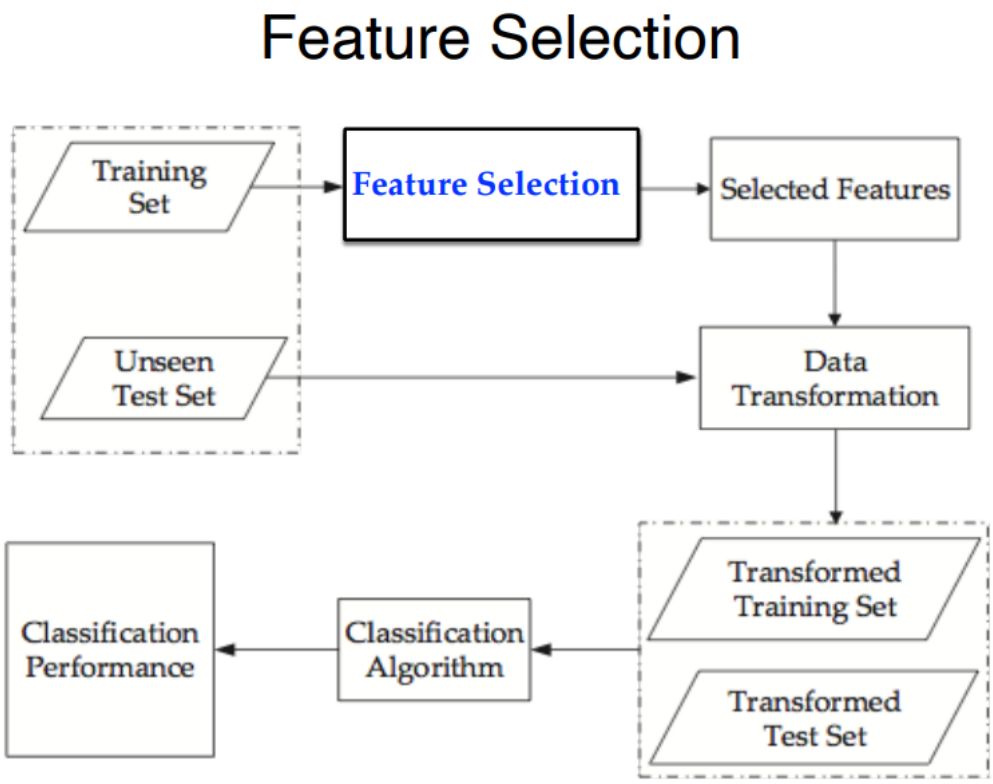
2 Genetic Algorithm for Feature Selection

Objectives:

- Genetic algorithm (GA) representation;
- Design proper fitness function, crossover and mutation operators for GA to solve a given problem;
- Proper parameter setting for GA

Problem Domain:

Selecting Features for Datasets is an important part of Data Engineering - and can be done using GA. The process for this is as below:



https://ecs.wgtn.ac.nz/foswiki/pub/Courses/AIML426_2022T2/LectureSchedule/lec03-GA.pdf

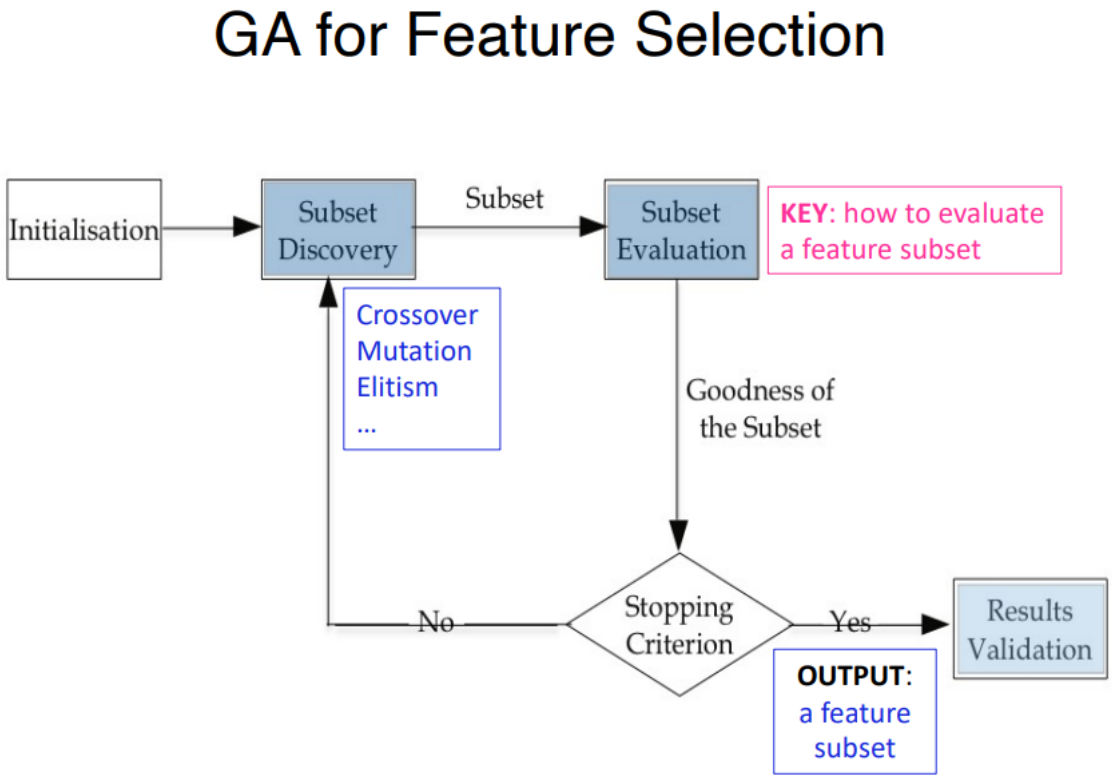
Datasets:

In this question, two datasets are given as follows:

- Winsconsin Breast Cancer (wbcd.data and wbcd.names), with 30 features and 2 classes.
- Sonar (sonar.data and sonar.names), with 60 features and 2 classes.

Solution:

To solve this problem I used only the libraries given by Sklearn in python. The flow of data for my solution is given below:



https://ecs.wgtn.ac.nz/foswiki/pub/Courses/AIML426_2022T2/LectureSchedule/lec03-GA.pdf

Representation/Encoding: the represention I choose for my individuals was as a string of n chars ('1' and '0') representing whether a feature active or not and where n is the number of features is of length equal to the number of features. Because python stores one char per byte, this will be occupy only n bytes of memory. A depiction of this is shown below:

https://miro.medium.com/max/875/1*ywCU0nmv8XhnTzUrz9DKJA.png

While this is not major for these datasets a common problem for big data is the sheer amount of space the immense numbers of features occupy.

Fitness Function(s): For this problem I used both Filter and Wrapper fitness functions. To briefly recap the difference between the two: Filter methods check how much influence each feature has on the label whereas wrapper methods actually train a model and measure the usefuleness of a given model. This makes filter models much faster but wrapper models more accurate.

Filter: For the filter function I used SelectKBest to return the k (5) best features using mutual info classifier as its scoring function which measures the dependency between two values. In this case it will measure how much the label depends on each column and is used for univariate feature selection.

Wrapper: For the wrapper function I used a forward Sequential Feature Selector with KNN classifier. The SFS adds features to form a feature subset in a greedy fashion. At each stage the KNN estimator chooses the best feature to add based on its cross validation score.

Crossover: My crossover operation is two point crossover as depicted below and described in depth in later sections - with a probability of 0.5:

```
def crossover(parent1, parent2):
    """
    Crossover using two point crossover
    :param parent1:
    :param parent2:
    :return:
    """
    # Choose two random points
    point1 = random.randint(0, len(parent1) - 1)
    point2 = random.randint(0, len(parent1) - 1)
    # Make sure point1 is the lower index
    if point1 > point2:
        point1, point2 = point2, point1
    # Create the children
    child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
    child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]
    return child1, child2
```

Mutation: My Mutation Operator was bit flip mutation which is depicted below and described in depth later. I also used a rate of 0.5:

```
def mutation(child, MUTATION_RATE=0.5):
"""
    Mutation using bit flip
:param child:
:return:
    """
for i in range(len(child)):
        if random.random() < MUTATION_RATE:
            child[i] = 1 - child[i]
    return child
```

Population size: 300

Termination criteria: Number of generations (40)

Selection: Tournament

Results

For each instance I ran the GA with the filter-based fitness function (called FilterGA) and the GA with the wrapper-based fitness function (called WrapperGA) 5 times with different random seeds. The Mean and STD of the time taken for this is shown below:

```
GA TIMES:
filter,
sonar
Mean:  0.09983286857604981
Standard Deviation:  0.013286388908495036

wbcd
Mean:  0.08124990463256836
Standard Deviation:  0.00624816908815229

wrapper
sonar
Mean:  3.4552278995513914
Standard Deviation:  0.23211602448322827

wbcd
Mean:  2.7250061988830567
Standard Deviation:  0.11464798442437629
```

- Compare the mean and standard deviation of the computational time of the FilterGA and WrapperGA and draw your conclusions.

What we can see here supports what has been discussed earlier which is that the filter function is much quicker than the wrapper function. I would expect to see a very different result if testing for accuracy - and as such that is the next step.

For each selected feature subset (5 subsets selected by FilterGA and 5 subsets selected by WrapperGA), I transformed the dataset by removing the unselected features and, using a Gasussian Naive Bayes classifier I performed a classification on the dataset. The mean and std of the accuracy of this classifier is shown below:

```
CLASSIFICATION ACCURACY:
filter,
sonar
Mean:  0.7761904761904762
Standard Deviation:  0.019047619047619025

wbcd
Mean:  0.9736842105263157
Standard Deviation:  1.1102230246251565e-16

wrapper
sonar
Mean:  0.6904761904761905
Standard Deviation:  0.0

wbcd
Mean:  0.956140350877193
Standard Deviation:  0.0
```

- Compare the mean and standard deviation of the classification accuracy of the 5 subsets selected by FilterGA and the 5 subsets selected by WrapperGA, and draw your conclusions.

Again, As was expected, the wrapper functions performed much better than the filter function. While this is not obvious in the mean accuracy itself, it can be seen definitively in the standard deviation of the the accuracy where the wrapper is entirely deterministic for both datasets. While the std of the filter is not high per say, it is still relatively poor compared with that of the wrapper.

It is interestng to note that both functions performed markedly poorly on the sonar dataset in their respective strengths. I attribute this mostly to the number of features contained in this dataset. While the actual amount of data is less than that of the wbcd dataset, the sheer volume of features means that removing all but 5 makes a significant impact on the accuracy of the classification model. Additionally performing the feature selection takes significantly more time, even for the filter function, as the number of comparisons to perform increases exponentially while simultaneously reducing the significance of each individual feature.

Non-dominated Sorting Genetic Algorithm-II

Objectives:

- Fitness assignment and selection for Non-dominated Sorting Genetic Algorithm II (NSGAII) for multi-objective optimisation;
- Design proper representation, crossover and mutation operators for NSGA-II for solving a problem;
- Proper parameter setting for NSGA-II.

Problem Domain:

In the previous task where we used GA to perform single-objective optimization for feature selection, , the number of selected features was not considered, and it is
possible that selecting all the features will lead to the highest classification accuracy (as it does not miss any information).

Extending this we will now aim to minimise the following two objectives in feature selection:

1. Minimise the classification error rate,
2. Minimise the ratio of selected features, i.e., number of selected features divided by the total number of features.

The two objectives are within the range [0, 1], and therefore have the same scale.

Dataset Evaluation

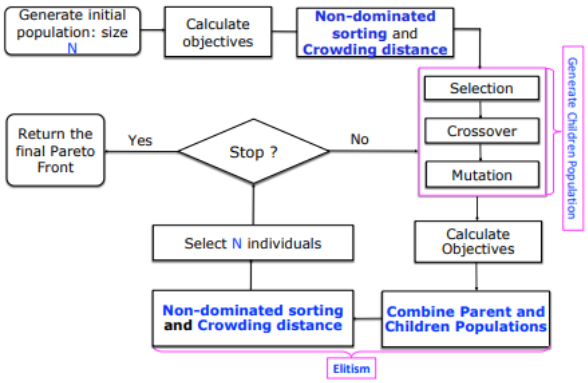
For this problem we will be training our algorithm against two datasets:

- Vehicle dataset (vehicle.dat and vehicle.doc), with 846 instances, 18 features and 4 classes.
- Musk “Clean1” dataset (clean1.data, clean1.info and clean1.names), with 476 instances, 168 features and 2 classes.

NOTE: these datasets are not split. As this is an optimization problem and cares only about the training performance, I will be using the entire dataset to do the feature selection without considering the test and feature selection bias.

Solution

To Solve this problem We will employ a Non-dominated Sorting Genetic Algorithm-II (NSGA-II) to minimise the above two objectives in feature selection, as described in {cite} and represented in the figure below:



https://ecs.wgtn.ac.nz/foswiki/pub/Courses/AIML426_2022T2/LectureSchedule/lec09-EMO2.pdf

For this problem I utilised the pymoo library for NSGA-II.

The Genetic Operators used are described below:

Representation/Encoding: The indidividual representation chosen here was a binary encoded array with 1 representing an “active” feature and 0 representing an “inactive feature”

Fitness/Objective Function: For the wrapper-based fitness function I used knn. This was because it has been recognised, in studies such as [1] to be a good choice of ML classification, with high performance regarding accuracy however a slow execution time. For our purposes, this works as we are not trying to optimize for time complexity and high levels of accuracy mean that any deviation can be assumed to be based on our feature selection.

This algorithm then compares the classified knn output with the known labels for the training data to evaluate the error and ratio of selected features. The best individual will then be selected based on which best minimize both of these values.

Crossover and Mutation: For this I used the built in **crossover** and **mutation** operators in the pymoo library - with their recommendations for NSGA-2 which was **Two point crossover** and **BitFlipMutation** respectively.

Two point crossover: Two point crossover essentially operates the same as single point crossover except with more points to divide the parents. point 1 will be taken from the first half of the number of features and 2 will be taken from the remaining features. offspring A will then get all the features from parent A between the indices of 1 and the first point with offspring B receiving the same from parent B. Then between point 1 and point 2 this swaps around and offspring B will get the features from parent A and vice versa. Finally this swaps again back to the original for the remainder of the features. Essentially two points will be inherited from the first parent and one will be inherited from the second. the pseudocode for this is shown below:

```
parent A, B
point1 = rand(2, NumberOfFeatures/2)
point2 = rand(NumberOfFeatures/2, NumberOfFeatures -1 )

for i = 1 to point1 - 1
  offspring1[i] = A[i]
  offspring2[i] = B[i]

for i = point1 to point2
  offspring1[i] = B[i]
  offspring2[i] = A[i]

for i = point2+1 to NumberOfFeatures
```

```
  offspring1[i] = A[i]
  offspring2[i] = B[i]
```

BitFlipMutation: Bit Flip (or bit inversion) mutation is much simpler as it simply involves flipping a random binary bit (representing whether a feature was active) in the individual (if a random value is less than the mutation probability). The pseudo code for this is below:

```
OFFSPRING =
for i = 1 to length(OFF)
  if rand() < mutationProbability
    OFF[i] = mod(OFF[i] + 1, 2)
```

- Set the necessary algorithm parameter values, such as population size, termination criteria, crossover and mutation rates, selection scheme.

Crossover and **mutation rate:** For this I choose 0.9 and 0.1 respectively

population size: For the **population size** I used 15.

selection scheme: tournament selection. Tournament Selection (TOS) selects k random chromosomes from the population, compares their fitness, and selects the best of that group as parent. This process is repeated n times until all parents have been selected. By choosing its competitors randomly, the algorithm puts pressure on the selection, and the pressure can be increased by selecting a bigger number for k. Random selection also improves the diversity of succeeding generations. The time complexity of this operator is $O(n)$, since there are n iterations, and each requires a constant number of selections k.

termination: I have two termination criteria. The first is the max number of iterations the algorithm should run, the other, is that the algorithm stops when the population has fully converged. A population is said to have converged when 95% of the population share the same genes.

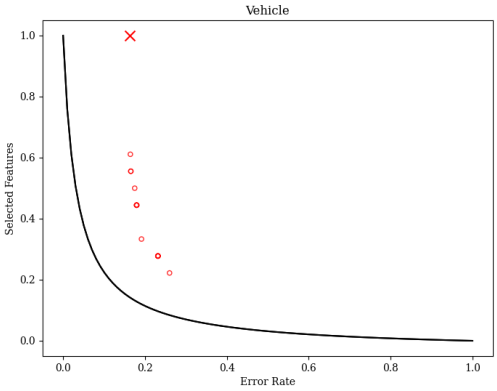
- For each dataset, run the NSGA-II for 3 times with different random seeds. Each run will obtain a set of non-dominated solutions (each solution is a feature subset).
- Compute the hyper-volume of each of the 3 solution sets obtained by NSGA-II.

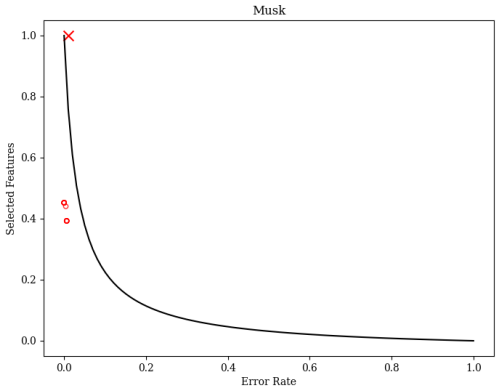
Results:

The results of running this algorithm on each dataset 3 times with different random seeds are as follows:

```
Vehicle
===========
Vehicle error rate with all features: 16.312056737588655%

1. Vehicle Error rate with 22.22222222222222% of features: 26.00472813238771%
Calculated Hyper-Volume: 0.6363934856842659

2. Vehicle Error rate with 16.666666666666664% of features: 27.30496453900709%
Calculated Hyper-Volume: 0.6922117152613607

3. Vehicle Error rate with 38.88888888888889% of features: 16.193853427895977%
Calculated Hyper-Volume: 0.6899789860782769

Musk
===========
Musk error rate with all features: 1.0504201680672232%

1. Musk Error rate with 39.285714285714285% of features: 0.6302521008403339%
Calculated Hyper-Volume: 0.6067927170868348

2. Musk Error rate with 39.285714285714285% of features: 1.0504201680672232%
Calculated Hyper-Volume: 0.6020158063225292

3. Musk Error rate with 42.26190476190476% of features: 3.361344537815125%
Calculated Hyper-Volume: 0.5746048419367746
```

And this is represented visually below with a plot of error rate against selection rate for both datasets shown below:





Each circle represents an output and the cross represents the plot of the original features for comparison.

NB: All three iterations for each dataset are represented on the same graph

Based on this we can see that, given,= the error rates obtained, cutting up to 60% of features, had, conservatively, a negligible difference to that seen when using the entire feature set. In some rare cases we even see marginal improvement which is interesting. This is only based on the training data set, however, and would be interesting to see how it performs when given a test set.

4 Genetic Programming for Symbolic Regression

Objectives:

- Main idea of genetic programming (GP) techniques
- GP program representation and generation;
- Determination of a GP terminal set and a function set for a given problem;
- Determination of a appropriate fitness function for solving a problem with GP;
- Choice of a set of parameters for GP learning;

Problem Domain:

In this question, your task is to build a GP system to automatically evolve a number of genetic programs for the following regression problem:

$$f(x) = \begin{cases} \frac{1}{x} + sin(x), x > 0 \\ 2x + x^2 + 3.0, x \leq 0 \end{cases}$$

Solution:

To solve this I used Symbolic Regression, implemented using the DEAP GP library and follows closely to their exemplars. My genetic operators were as follows:

Terminal and **Function** set: The terminal and function set was mapped as follows:

```
"add" <-- x + y
"sub" <-- x - y
"mul" <-- x * y
"neg" <-- x * -1
"div" <-- x / y
"cos" <-- cos(x)
"sin" <-- sin(x)
"inv" <-- x ** -1
"sqrt" <-- x ** 0.5
"RAN" <-- random(-1, 1)
```

These were chosen as virtually any equation can be created from some combination of these.

Fitness Function: My fitness function for this problem is the mean-squared error of the individual generated. That is to say, in the objective space (-10 < x < 10) every x value is tested for each individual as well as the regression problem function itself. The difference is then squared and summed. This program aims to minimise this error and so the lowest error will be selected for the next generation.

Population Size: My population size was set at 300 as guided in the DEAP tutorial

Max Tree Depth: The max depth of a given solution was capped at 17, as suggested in *John R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992, pages 162-169*.

Termination Criteria: My termination criteria was set at max number of generations - this was predimantly due to the nature of the DEAP eaSimple algorithm not providing a termination parameter.

Crossover: My Crossover function was OnePoint Crossover - which is essentially the same as the Two point crossover described previously, but with one less step. I set its' probability to 0.7 as I found this yielded the best results

Mutation: The mutation operation used here was Uniform Mutation., which, according to the DEAP gp documentation:

```
Randomly select a point in the tree *individual*, then replace[s] the
subtree at that point as a root by the expression generated using method expr
```

where expr here was genFull which generates a full expression where each leaf has the same depth between a predefined min and max (here 1 and 3). The mutation probability was 0.1 as I found the lower this value was the better the results.

Selection: For Selection I used a tournament of 20 values which selects the individual with the best fitness from the 20 randomly selected individuals in the population.

Results

I ran this GP system 3 times with a different random seed each time and returned both the expression and its respective fitness value of the best individual in the output hof. This can be seen below:

```
seed: 1
expr: add(inv(div(-1, add(x, add(add(-1, sub(x, sub(x, 1))), add(add(add(add(-1, sqrt(add(add(sqrt(div(add(x, x), x)), sin(x)), -1))),
fitness: 38.650737632105354

seed: 2
expr: sub(sqrt(add(add(add(add(div(-1, 0), mul(x, 0)), 1), mul(x, 0)), x)), sub(add(add(x, x), x), sqrt(add(mul(add(mul(x, x), x), x),
fitness: 35.73900162022097

seed: 3
expr: add(sub(neg(x), sub(x, -1)), sqrt(mul(mul(x, sub(sub(neg(1), sub(x, neg(x))), div(neg(x), div(x, x)))), sub(sub(sqrt(mul(sub(neg
fitness: 23.764279284789474
```

In the interests of keeping this on the page I perhaps should have set my max depth at a marginally lower value. That being said the algorithm seems to perform well in terms of fitness with the mean squared error being fairly low consistently, although the actual output expression looks vastly different from that expected.

5 Particle Swarm Optimisation

Objectives:

- Design proper topology, velocity and position update mechanisms for PSO to solve an optimisation problem;
- Proper parameter setting for PSO;

Problem Domain:

The problem here is to find the minimum of the following two functions where D is the number of variables, i.e. $x1, x2, ..., xD$. Rosenbrock's function:

$$f_1(x) = \sum_{i=1}^{D-1}(100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2), x_i \in [-30, 30]$$

Griewanks's function:

$$f_2(x) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}}) + 1, x_i \in [-30, 30]$$

Dataset Evaluation

For this problem there was no provided dataset. Despite this we know that Rosenbrock and Griewanks functions are solved problems and we know that both have a global minimum of 0, so we will use this for the evaluation of our algorithm.

Solution

To solve this problem we will use Particle Swarm Optimisation (PSO) with the following genetic operators:

For **c1** and **c2** I choose to make c1 > c2 (`1.49618` and 0.49618 respectively) as guided in the lectures given that this is a multimodal optimization problem with many local minima.

for **w** I originally choose 0.7298, also as guided by lectures, as this prioritised exploitation. However, I later implemented a step function to linearly decrease w until it reached 0.4 from 0.9. This step was proportional to the number of max iterations so will only hit 0.4 on the final iteration.

For my **population size** I choose 50 for no particular reason that I can remember

fitness function: for the fitness function I used the algorithms themselves and compared it with the known optimal values (ie where we know the minimum points are). I took the absolute value of the difference between them as the error and optimised to minimise this.

particle encoding: for the encoding of the particles I used a dictionary with the keys being the current position, current velocity, current fitness, personal best fitness, and associated personal best position. While each particle is, in theory, also supposed to remember the global best fitness, this was unnecessary as it made more sense as a global variable.

Topology Type:

Stopping Criterion: My stopping criteria were a) max iterations (50) and b) minimum acceptable error threshold reached (defined by the known optimal value for each algorithm)

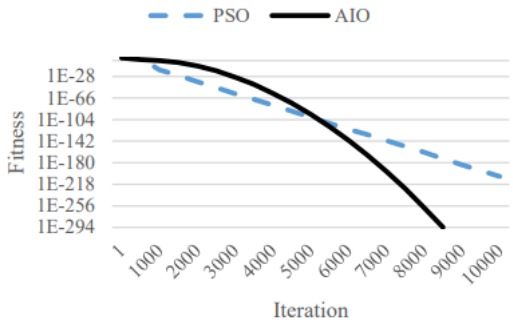
Results:

For each of these functions I repeated the experiment 30 times, and the output at **D=20** (Number of x values) is as below:

```
Rosenbrock target fitness:  0
Griewanks target fitness:  0.0

Rosenbrock's function for D=20:
mean:  71757.25276407467
std:  106755.08805674239

Griewanks's function for D=20:
mean:  0.901499013998535
std:  0.11652973068931972
```

My first point of discussion revolves around the Rosenbrock dataset - where the fitness seems remarkably off the expected values, despite converging very quickly.

This observation has been supported by my additional reading which shows, in figures such as the one below, that this is simply an attribute of the Rosenbrock function:



https://arxiv.org/ftp/arxiv/papers/1804/1804.00768.pdf

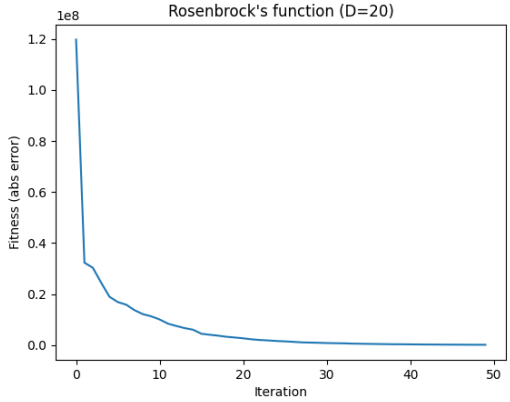
This did improve dramatically as the population size and number of iterations increased, where little change was seen for the griewanks function which seemed to manage to fairly linearly collect close to the global maximum.

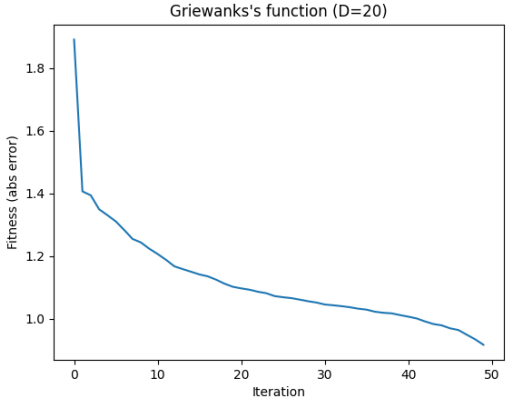
To test this further I Then ran the Griewanks function another 30 times using these same paramters, except with **D = 50**, to attempt to solve the fuction. The Results of This can be seen below:

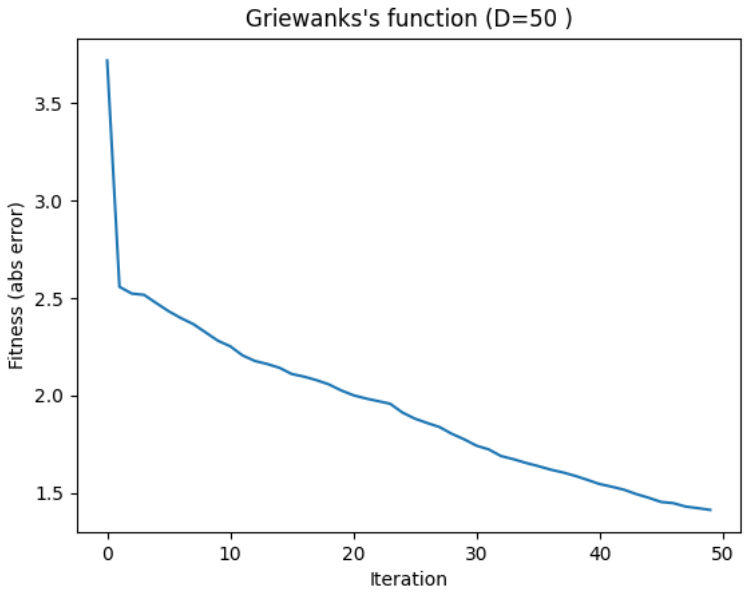
Output over 30 repeats for D=50 as below:

```
Rosenbrock target fitness:  0
Griewanks target fitness:  0.0

Griewanks's function for D=50:
mean:  1.4552127484829802
std:  0.1373815944844559
```

The convergence curves across the three operations (averaged for 30 repeats) as below:







As Can be seen increasing D made very little difference to the Griewanks Function, with both the error and standard deviation remaining low and close to the known global minimum.