Test Case Prioritisation and Ranking using Reinforcement Learning and improvement using Genetic Algorithms

Final Year Project

Thomas Armstrong
Msc Computer Science
Module Credits : 60
Supervisor: Anis Zarrad
Student ID: 2179997
Word Count : 13510

MEng Computer Science with Software Engineering

UNIVERSITY OF BIRMINGHAM

School of Computer Science
University of Birmingham

Contents

1 Abstract **2**

2 Introduction **2**
2.1 Testing . . . 2
2.1.1 Unit Testing . . . 2
2.1.2 Regression Testing . . . 2
2.2 Test Case Selection Metrics . . . 3
2.2.1 Average Percentage Fault Detection (APFD) . . . 3
2.3 Normalised Rank Percentile Average (NRPA) . . . 3

3 Background **3**
3.1 Reinforcement Learning . . . 3
3.1.1 Introduction . . . 3
3.1.2 Base Algorithm . . . 4
3.1.3 Deep Q Network . . . 4
3.2 Evolutionary Algorithms . . . 6

4 Related Work **6**

5 Methodology **8**

6 Understanding the Dataset **8**

7 Mutual Information **9**
7.1 Overview . . . 9
7.2 Definition . . . 9
7.3 Development Process . . . 10

8 Feature Engineering **10**
8.1 Created Features . . . 11
8.2 Iterative Enginerring of Mutual information . . . 12

9 Prioritisation **14**
9.1 Methods . . . 14

10 Reinforcement Learning **18**
10.1 Learning Process . . . 18
10.2 Environments . . . 18
10.2.1 List Wise . . . 18
10.2.2 Pairwise Verdict . . . 19
10.2.3 Pairwise Rank . . . 19
10.3 Pseudocode . . . 20
10.4 Training Setup . . . 22
10.5 Challenges in production . . . 22
10.6 Normalisation . . . 23

11 Genetic Improvement **24**
11.1 How GA will help RL . . . 25
11.2 Maintaining Shared Memory . . . 25
11.3 Algorithm . . . 27
11.4 Fitness Function . . . 27
11.5 Selection . . . 28
11.6 Mutation . . . 28
11.6.1 Simple Random Mutation . . . 28
11.6.2 Availability Mutation . . . 28
11.7 Crossover . . . 30
11.7.1 Naive Crossover . . . 30
11.7.2 Similarity Crossover . . . 30
11.8 Arithmetic Crossover . . . 31

12 Results **31**
12.1 Reinforcement Learning 31
12.1.1 Testing procedure 31
12.2 Results Comparison and Analysis 32
12.3 Evolutionary Improvement 34
12.3.1 Testing procedure 34
12.3.2 Graphs 34

13 Future Work and Evaluation **38**

14 Conclusion **38**

15 Appendix **41**

1 Abstract

This research project focuses on the field of test case prioritisation and ranking using Reinforcement Learning techniques. Random forest classification performs test case selection with an accuracy score of 0.98847 before a deep Q learning algorithm is employed to prioritise the test cases using two pairwise and a listwise reinforcement learning environment. After a successful improvement in APFD compared to results from previous work, prioritisation using genetic algorithms is employed. Although the adapted variation operators did not build on vanilla versions a small increase in performance was observed with the cost of larger running time when applying a genetic algorithm.

2 Introduction

Software is ubiquitous in today's society, with every aspect of life governed by one program or another. As a software company, one must be able to compete with the vast number of competitors in an effective manner, be it with smaller application development or larger, industrial software production. Pioneering ways to boost software quality is a must. In this paper we will discuss the manner of quality assurance of software testing. Development teams have a limited budget, on time and and money, when it comes to testing some software. When a new change is made to the system, then a large number of tests must be carried out in order to ensure that the correct functionality is ensured. Some of these tests are much more likely to fail than others and it is in the tester's interest to find the tests that fail as quickly is possible. With greater ease in planning the test cost and phase length, comes high resource optimisation, high user retention rates, early results feedback and improved test coverage. In fact Whittle (n.d.) states that 90 % of users stop using apps after a week due to poor performance.

The aim of this project is to find an efficient method to select these test cases in order to optimise the testing process. To understand and define the problem at hand we must first understand code testing as a whole.

2.1 Testing

There are two main types of testing of code: one offers a low level insight into the workings of individual functions or modules while another offers a high system wide check.

2.1.1 Unit Testing

The former is that of unit testing. In Tosun et al. (2018) it is written that these tests' objectives are to test a specific functionality on a small area of code, for example individual functions within the program. These are crucial to development because these tests ensure that later code refactoring do not damage previous basic functionality.

2.1.2 Regression Testing

Alternatively, according to Onoma et al. (1998), regression testing is that of testing the entire code structure according to the pre-written tests in order to ensure that the system continues to meet the predefined functional and non functional requirements. This is our system wide testing procedure. After the software modification, valid test cases (test cases that are relevant to the change) are executed. Finally, the result of the testing is the fault identification, in which a programmer can identify the malfunctioning lines by examining the results and mitigate the fault. Put more simply, the code is tested after a change with the relevant tests to make sure that it works as intended. According to Amir et al. (2017), there are two main types of regression testing, *retest*

all and *selective retest*. *Retest all* meaning that the entirety of the testing suite will be retested. Naturally however the paper goes on to mention that this procedure is inefficient for large test suite and is therefore the reason for this paper. The second testing strategy is one in which a subset of the test suite is used for regression testing and Amir et al. (2017) goes on to mention differing test selection techniques in an effort to optimise the regression period. Furthermore, Akira K. Onoma et al. Onoma et al. (1998) discusses how the cost of the testing process is increased, and although some of the problems come down to the human issue, for example writing the tests and manually checking the results of the tests, another large competitor is that of the selection of the correct test cases Onoma et al. (1998). In this paper, we will explore a solution in which test cases can be selected using a novel approach in the field, that of genetic algorithms in conjunction with RL.

2.2 Test Case Selection Metrics

2.2.1 Average Percentage Fault Detection (APFD)

The first case used for comparative evaluation is as follows. Bagherzadeh et al. (2021), Srivastava (2005-2008*b*) among others in the subject used APFD as standard for measuring the effectiveness of a test case prediction. The metric is calculated by taking the weighted average of the number of faults detected during the run of the test suite. More precisely the metric is defined as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} - \frac{1}{2n} \tag{1}$$

Where T is the test suite in evaluation, m is the number of faults in the subset of the test case being tested, n is the total number of test cases and TF_i is the first position in T which exposes fault i. Srivastava (2005-2008*b*). This test metric will be used primarily to ascertain the effectiveness of a ranking algorithm focused on prioritising test case failure, see section 10 for further discussion.

2.3 Normalised Rank Percentile Average (NRPA)

The second metric, in contrast to APFD, does not take test failure into consideration, but rather a measure of accuracy when compared to the optimal ranked list. The NRPA is useful when there are minimal test cases in the selection, in which the APFD score will perform poorly. The metric is defined below:

$$NRPA = \frac{RPA(E)}{RPA(E_o)} \text{ where } E_o \text{ is the optimal order of a sequence } E \tag{2}$$

$$RPA(E) = \frac{\sum_{e \in E} \sum_{i=id(E,e)}^{k} |E| - id(E_o, e) + 1}{j * *2(j+1)/2} \tag{3}$$

Where $id(E, e)$ returns the position of e in E.

Note here that in the field of test case prioritisation and ranking, due to the overall aim of test case prioritisation, it is more important to rank test cases on their likelihood of failure compared to ordering test cases on their perceived rankings. As such, the APFD metric should be weighted with higher importance compared to NRPA.

3 Background

This section provides a background to the project.

3.1 Reinforcement Learning

3.1.1 Introduction

There are three fields of Artificial Intelligence (AI) learning; Supervised Learning, Unsupervised Learning, and Reinforcement Learning (RL). Although we will cover RL, we will give a very brief overview of the other fields of AI. Supervised learning is a field in which given a set of labelled data, models can learn to predict, which label the an unseen datapoint belongs to. For example, image classification algorithms can take an image of a cat, and given training on a dataset of cats and dogs, can distinguish this image with a label of "cat". Unsupervised learning is unlabelled, meaning given this image of a cat, the AI can distinguish that the image can be grouped with the other cat images. While supervised and unsupervised learning are shown to have more accurate results (Bertolino et al. (2020*a*)), RL, much new than the aforementioned AI techniques, can adapt to changes in the dataset in a much more efficient manner.

3.1.2 Base Algorithm

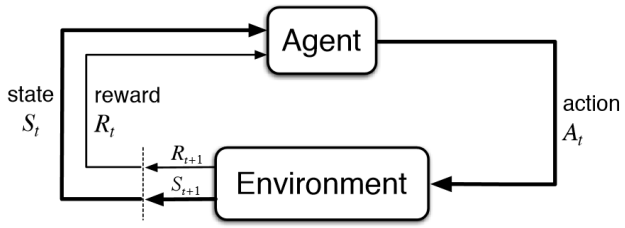

Figure 1: Reinforcement Learning Event Flow

RL consists of a learning agent interacting with an environment through actions, states, observations and rewards. At each time step t the agent will receive an input state $s_t \in S$ where S is the the set of all possible states. At each state the agent can take an action $a_t \in A$, where A is the set of all possible actions and move to the next state. s_{t+1}-the observation. Note this explanatory example. The environment is a grid of squares, the state is the coordinate of the particular square that the agent currently resides ($s_t \in \{(0,0),(0,1),(1,0),(1,1)\}$) in and the action would be to move the agent either North, South, East or West to a neighbouring square: $a_t \in \{"North","South","East","West"\}$. Within the environment, there are rewards for reaching particular states $r_t \in R$ (where R is the set of all possible rewards), or reaching a particular time threshold. To continue the example, some squares may contain an "enemy" and thus that state would contain an negative reward. Meanwhile, the agent may pick up rewards along the way, for example collecting coins. Here the reward given may be a positive reward. Therefore the experience of an agent at each time step will be notated as transition T = $\{s_t, a_t, r_t, s_{t+1}\}$ and the RL algorithm takes this T as input to learn to maximise the estimated future reward for all future actions.

3.1.3 Deep Q Network

This project will be using a deep Q network as the RL algorithm.

Q learning

To understand Deep Q networks, one must first understand Q learning. In order to learn, as mentioned previously, the agent must maximise the future reward of all actions. Therefore, the agent must keep track of the reward that it receives for each action it takes from each state. This is done in what is called a Q table. A Q value is informally defined as the reward achieved in the current state, plus the perfect actions in the future. This was the Q value can be recursively defined be figure 6.

The Bellman Equation is as follows (Lizard (n.d.))

$$Q_*(s_t, a_t) = E[R_{t+1}(s,a) + \gamma \max_{\text{a}} Q_*(S_{t+1}, a)] \tag{4}$$

This states that for an optimal Q function (Q_*), given any state action pair (s_t, a_t), the optimal value will be the expected value achieved after the agent receives it's relevant reward and thereafter behaves optimally. All Q functions obey this rule and therefore our Q value is defined as follows:

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \max_{\text{a}} Q(S_{t+1}, a) \tag{5}$$

where the maximum value of a state action pair is the the value of the current reward $R(s,a)$ plus the discounted value of the next state with the best next action taken.

$$Q(s,a) = R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) + \gamma^2 Q(s_{t+2}, a_{t+2}) + ... + \gamma^n Q(s_{t+n}, a_{t+n}) \tag{6}$$

Expanding out the recursive function we produce equation 6, it becomes clear to see that a Q value of a state is dependent on all future reward, while altering the value of the hyper-parameter γ will reduce the contribution of all future rewards. Note that the expansion takes place due to the fact that all states are considered. Q_{s1} is dependant on Q_{s2} which is dependant on Q_{s3} and so on.

Furthermore the Bellman Update Equation is shown as follows:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_{\text{a}} Q(S_{t+1}, a) - Q(s_t, a_t)] \tag{7}$$

This describes how to improve the current value in the Q table. Where α is the learning rate of the problem, whose value decides the rate at which new information overrules the current information.

Deep Q Learning

As the dimensionality of the problem increases, storing the Q values of all possible states and actions will quickly become infeasible. However, as a neural network is a universal function approximator, here we use a neural network to predict the Q function. A Q value is defined as the Q value of the next state plus the reward achieved in the current stare

A. Temporal Difference Learning

From equation 7 we can define the error to be used in the loss function for use in the gradient descent of the neural network.

$$\delta = \underbrace{R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a)}_{\text{target}} - \underbrace{Q(s_t, a_t)}_{\text{prediction}} \tag{8}$$

This is known as the temporal difference error (Paszke & Towers (n.d.) and fully encapsulates the definition of a Q value.

B. Problems with Deep Q Learning

At each time-step, δ will be used used to calculate the loss and update the weights of the neural network (θ). However both the target and the prediction have a high correlation with θ, and thus both will change simultaneously in training and cause oscillation. The loss cannot be calculated accurately if the primary comparative entity varies.

A solution proposed by Face (n.d.) is to use two neural networks, one for the prediction and one for the target such that the target network has the same architecture as the prediction network but with frozen parameters. Every C time steps the parameters of the target network are updated to match that prediction network so that prediction network is an estimate of the prediction network, and training can remain effective. Figure 3 depicts this process.

Usually, an agent such as this receives a transition, learns from it and is never used again. Not only is this not an efficient learning process, but it also presents the issue of forgetting: some states may be visited in training but then never used again; if when the state appears in testing, new information has taken precedence and the agent does not act correctly. Face (n.d.) discusses the solution of the Experience replay memory. This is a memory storage component that will store the state transitions to be used again in the training process, such that the previous states can be repeatedly revisited and reinforced.

C. Loss Function

Sabah et al. (2023) states that there is no best error function for the general problem. However, Fox et al. (2016) explains that that Q learning approaches are vulnerable to noisy input data. This is because networks spend a long time backtracking, after learning an incorrect policy. With this in mind, Sabah et al. (2023) explains that Hubert loss is a loss function that is very robust to noise, and in order to protect my network I chose to use this loss in my solution.

D. Algorithm used

The deep Q learning algorithm used in this paper will be the method discussed, with an overview highlighted in figure 2.

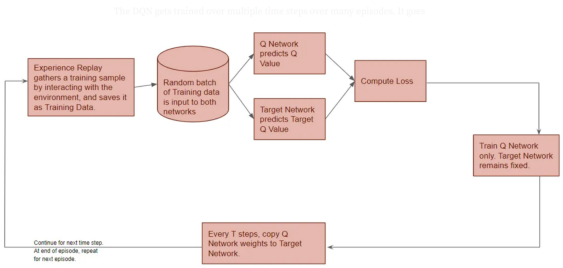

Figure 2: High level deep Q learning workflow
Doshi (n.d.)

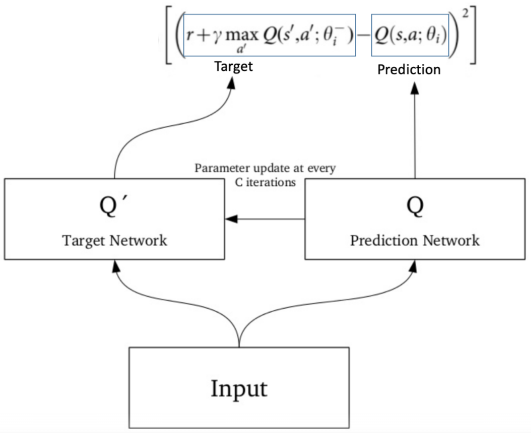

Figure 3: Deep Q learning networks flow
Choudhary (n.d.)

3.2 Evolutionary Algorithms

Evolutionary algorithms (EA) are a population based, stochastic local search optimisation algorithm. Specifically they are a subset of the metaheuristic algorithm class. While heuristics are algorithms are those that follow a simple rule to achieve near optimality, a metahauristic is a "master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality" Glover & Laguna (1997). Initially coined by Holland (1975), EA are designed to replicate Darwin's model of Evolution in which the fittest individuals survive to the next generation. Here a population of solutions are generated and compared using a fitness function. Once a subset of the "best" solutions are chosen through a chosen selection scheme, genetic reproduction takes place, be it through genetic crossover or mutation to produce offspring. Pseudocode of the process used in the program can be found in figure 6 in section 11

4 Related Work

There have been multiple works throughout the years detailing research in the test case prioritisation field, and work here will be discussed canonically, as the improvements to the approaches are made. Work discussed here will begin with studies of testing as a whole, before moving towards basic human driven prioritisation and then its improvement: an automated solution using ML. More recent work on RL based prioritisation is detailed and then finally work on how improvements to the accuracy and efficiency can be made.

We begin with research carried out in 2014 by Mäkinen & Münch (2014), with findings to suggest that testing code before deployment, specifically in test driven development, proves advantageous to the quality of the tested software. Leading to solutions improvements such as increased maintainability of the code, with tests that validate the functionality of the software, developers can, with more ease, make changes to the code-base without introducing new defects to the system. Later Amir et al. (2017) discusses the two main types of regression testing: *retest all* and *selective retest*, before concluding that the retest all convention is highly inefficient when it comes to making use of a large scale test suite.

The most prominent techniques for regression testing optimisations are shown to be test selection, test case minimisation and test case prioritisation. Yoo & Harman (2012) explains that although these techniques are largely similar in terms of motivation and functionality, there is a trend towards the use of test case prioritisation compared to the other techniques in the literature. There have been a number of previous works detailing the use of test case prioritisation. Rothermel et al. (1999) discusses differing techniques used for the prioritisation: risk based prioritisation, which aims to identify test cases based on their impact on critical areas, or functionalities of a program; code coverage based prioritisation, which focuses on selecting test cases which cover the most lines of code, when access to the original code-base was available. Rothermel et al. (1999) concluded that test case prioritisation, no matter the technique used, greatly increased the rate of fault detection compared to an untreated retest all approach. This conclusion is also matched with Srivastava (2008*a*), in which a novel manner of test case prioritisation, ranking test cases based on average faults per minute, is tested compared to the untreated test suite in terms of fault detection rate. Test case prioritisation is proven to improve fault detection rate, budget adherence and time effectiveness and give early feedback for debugging for even the least sophisticated techniques; in these papers as mentioned, but also discussed by: Sharma & Chande (2023),

Kajo Meçe et al. (2020), and Rothermel et al. (2002).

Sharma & Chande (2023) then noted that traditional techniques (Rank based and code coverage based), although effective in improving code quality, had their own drawbacks. Risk based prioritisation requires domain expertise from the test selector, and the coverage of the code may not be adequate. Code coverage based prioritisation does not consider fault detection and so cannot be used to prioritise on fault likelihood which is an essential component of test case prioritisation. These methods cannot capture complex feature relationships like ML can. When prioritising test cases, Onoma et al. (1998) explain that testing these redundant test cases are a waste of resources. Furthermore, both these features require human input, can then lead to subjective prioritisation and risk assessment and are in need of automation due to poor time efficiency.

In later work, test case prioritisation and selection is then automated using machine learning (ML). Kajo Meçe et al. (2020) makes use of numerous ML techniques in their paper, namely some examples are: genetic algorithms, support vector machines, Bayesian networks and decision forests. These methods are used to prioritise test cases with good results for prioritisation and selection. Despite this, these methods require a large dataset to be able to train effectively and give the desired accurate results. Neural networks are said to be very adaptive to differing input features, which, in our case, inspires a neural network based RL-GA algorithm such as the one proposed in this paper. Furthermore, genetic algorithms also prove an effective ranker in this paper, adding to the previous point. Kajo Meçe et al. (2020) go on to explain that the use of an ensemble learner, which combines multiple machine learning algorithms, proves effective in test case selection and prioritisation while Cao et al. (2022) concludes that ensemble methods used for test case prioritisation can outperform their base algorithms. In another work, Marijan (2022) makes use of the aforementioned individual ML techniques as well as gradient boosted ML methods, such as gradient boosted decision trees, which result in strong comparative performance compared to other ML approaches, non ML approaches and untreated data.

The application of RL algorithms for test case prioritisation is a new and active area of research, with all papers found in this field written no earlier than 2020. In the paper written by Bertolino et al. (2020*b*), the use of RL and other ML techniques are compared and it is concluded that RL algorithms take much longer to train, due to the fact that the training is repeated at every commit in the CI (redhat (2022)) cycle. However, there the time to first prioritisation (TTFP) is 0 because the RL models have the capability to begin to predict immediately, while the non RL methods must complete their training.

RL approaches are proposed to be much more adaptive to change than other non RL techniques. By being adaptive to change in the context of test case prioritisation, it means that for every CI cycle the test case execution history is extended, with the new test execution data retrieved from the current iteration. Therefore the model must be able to quickly adapt to these additions. This is a key characteristic discussed in by Bertolino et al. (2020*b*), as well as Bagherzadeh et al. (2021).

This adaptivity is down to numerous characteristics which are introduced in this paper- François-Lavet et al. (2018). Continuous learning: RL agents are able to learn and adapt their generated policies based on ongoing interactions with their environment. New states can be added to this environment at any moment, namely mid training, and the agent will work as expected. Generalisation: François-Lavet et al. (2018) states that RL techniques can generalise their learnt knowledge across similar states, and into new environments- as well as having the capacity to achieve good performance in an environment where limited data has been gathered. This concept is known as having good sample efficiency. One reason to the latter point is down to how RL algorithms use a replay memory structure to train from previous environment interactions such that the learning experience can be repeatedly revisited and reinforced.

Furthermore, these RL algorithms discussed by Bertolino et al. (2020*b*) have a low accuracy score compared to other non-RL techniques. It it clear to see that RL has a strength in adaptability, and in this paper we attempt to improve the perceived weakness of accuracy and complete training time so that these strengths can be brought more to light.

Bagherzadeh et al. (2021) executes and compares numerous state of the art RL algorithms in the field of test case prioritisation and ranking in the CI/CD environment. This paper discusses Reinforcement Learning environments, such as pairwise and listwise environments (discussed in this paper) combined with different state of the art RL algorithms. The paper contributes the fact that the most effective algorithms is that of the ACER algorithm (Wang et al. (2016)) combined with the pointwise environment. However, the results proposed demonstrated that the deep Q learning algorithm performs poorly in terms of prediction time, training time and accuracy, for all environment selections- placing last in many of the comparisons. In this paper we will continue this work, aiming to improve this deep Q learning result.

Iglesias et al. (2006) discussed a learning problem using the Q learning algorithm, in which a robotics system

was able to navigate around a set perimeter. The authors proposed a method to improve the performance of the TTD algorithm by combining a GA into the RL procedure. Their results proved that with the combination of GA and RL the convergence time for the algorithm (compared to the base RL), as well as the number of negative rewards, decreased significantly. The main drawback from this algorithm used is that the TTD suffers from the same "curse of dimentionality" similar as explained in section 4.5.3. Therefore, the intention of this paper is to continue this work and apply this technique to model more more real world scenarios that may have many input states and actions.

Multiple previous works discuss genetic improvement on neural networks. Crossover methods have widely been discouraged in neural network evolution due to the competing conventions problem or otherwise known as the permutation problem, as discussed by the following authors: Uriot & Izzo (2020), Dragoni et al. (2010), Haflidason & Neville (2009), Pretorius & Pillay (2024). This problem explains that there is a many to one mapping between the phenotype and genotype pairs of a neural network. In other words, "there exists many different chromosomes that represent functionally equivalent neural networks".

Crossover operators can become destructive to the offspring the networks for this reason. Due to this fact, neuroevolution works often tend to omit the crossover operator, and furthermore lose a significant ability to traverse the search space in larger steps.

Crossover operators that negate the issues previously mentioned have been later explored by Dragoni et al. (2010) and Uriot & Izzo (2020) which have been shown to improve performance compared to previous neuroevolution approaches. These operators will adapted and used in this paper and thus will be explained section 11.7.

5 Methodology

To execute this project, the following steps will be followed.
1. Understanding the dataset, and engineering features that will correlate to the target features more strongly such that higher accuracy scores can be obtained.
2. Carry out the classification process to prioritise the test cases into groups. Classification algorithms used in the field will be compared, their parameters will be optimised using grid search and the top performing classifier will be selected.
3. To prioritise the date using RL, a range of environments will be constructed as used as the learning environment for the RL agent.
4. Apply a genetic algorithm to improve the performance of the original RL agent using adapted variation operators.

6 Understanding the Dataset

The initial data I have used for this paper includes results from a testing process that took place throughout 2016. The dataset was provided by my supervisor, named paintcontrol final, and is linked below pai (n.d.). Throught the year, 89 individual test cases were used and 25594 tests were run. The features of the data are shown in Table 1.

Table 1: Initial Features

Feature Name	Content Explanation/ method of creation
ID	Unique Identifier of the test execution
Name	Unique Numeric Identifier for the test case
Duration	Run time of the test case
CalcPrio	Priority of the test case - previously calculated by a prioritisation algorithm priority 3 is the most critical, and 1 is the least
Last Run	Previous Execution of the test case in the date-time format string format (YYYY/MM/DD HH:SS)
Last Results	List of previous test results (Failed = 1, Passed = 0) Ordered by ascending age. Delimited by []
Verdict	Outcome of test case execution (1 for failed test, 0 otherwise)
Cycle	Which CI cycle the this test execution belongs to
Rank	The previously calculated rank of a test execution within a particular priority
Rank Score	Rank score previously calculated by a ranking algorithm
Quarter	Quarter of the year in which the test was run
Month	Month of the year in which the test was run

Before using this data, redundant features were removed: Name and ID

7 Mutual Information

7.1 Overview

It is paramount that the correct features are selected for use in the model. Features that have no correlation with the target feature will not return accurate results.

Mutual Information is a numeric value that measures how much information is communicated between two variables that are sampled simultaneously, and so features with high mutual information with a target feature will be used more effectively to predict its outcome.

7.2 Definition

The equation for mutual information is defined as follows:

$$I(X;Y) = H(X) + H(Y) - H(X,Y) \tag{9}$$

Where $H(X)$, $H(Y)$ and $H(X,Y)$ are defined below.

$$H(C) = -\sum_{x} p(c)\log_2 p(c) \tag{10}$$

$$H(X,Y) = -\sum_{x}\sum_{y} p(x,y)\log p(x,y) \tag{11}$$

$p(x)$ and $p(x,y)$ is defined as the probability mass function and join probability mass function in the above equations. Figure 10 shows the marginal entropy of an arbitrary variable C, while figure 11 shows the joint entropy between random variables X and Y. Entropy is used to measure the level of unpredictability in a random variables outcome, for example when flipping a coin, there is an equivalent amount of "surprise" when landing on either heads or tails because the outcomes are equiprobable. In this example there would be one bit of entropy. However if the coin is biased, with a 1/3 and 2/3 chance for heads and tails respectively, there would be greater "surprise" landing on heads and therefore an outcome of heads would yield a higher entropy than tails. Joint distribution is a measure of entropy in the joint distributions of the two random variables. In our context, entrpoy can be used to calculate the mutual information of two variables.

7.3 Development Process

Initially, an algorithm of personal design which made use of the above formulas was employed. While it was successful, the algorithm used needed to discretize the data into bins so that the continuous distribution could be made effective for use in a probability function. However, upon application of this algorithm, it was found that the more bins that were added, the slower the execution time was of the function. Meanwhile the results of the calculation were varied constantly.

After trialling this method and observing poor results, a sklearn library was utilised in its place which solved the aforementioned issues.

Results from this are shown below:

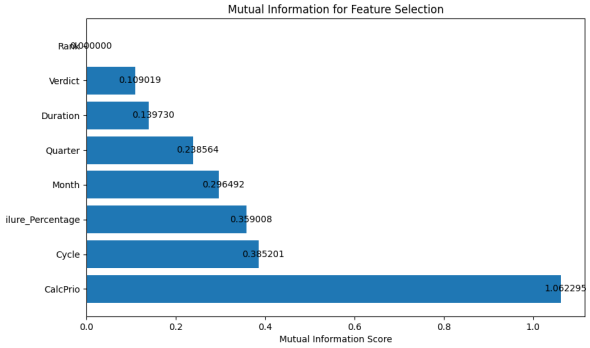

Figure 4: Initial Mutual Information for CalcPrio

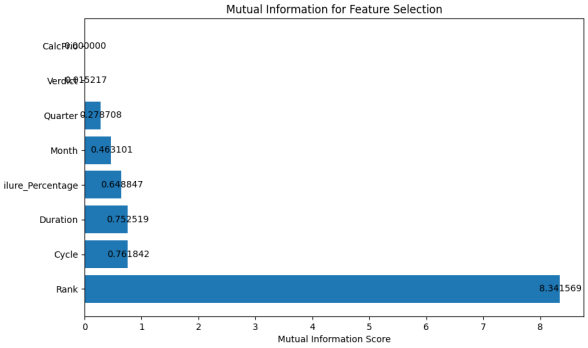

Figure 5: Initial Mutual Information for rank

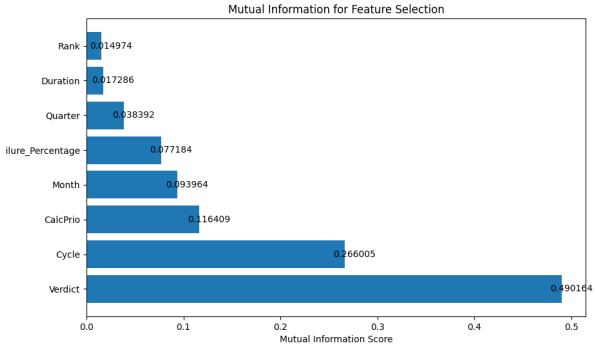

Figure 6: Initial Mutual Information for verdict

Explained in section 10 verdict is essential our predictions, but it can be seen that verdict only has a mutual information score of more than than 0.2 with only one feature which is not acceptable.

For prediction of priority, it is shown that there are only 4 viable features which can be used for the classification process and this is insufficient. In terms of predicting ranking however, all features except verdict and priority have strong correlation with ranking and are thus already sufficient for use.

In the next section, the process of creating more, improved features for use is modelled.

8 Feature Engineering

In order to predict correctly, only features with a high correlation with the target feature can be included. To allow for this, new features were created using feature engineering, a practice of extracting new information and new variables from the original data. A metric for whether a set of features is acceptable for use in the model, is decided as follows:

1. A feature will be used in the model if the mutual information score is at least 0.05.
2. There must be at least three significant features in the model where a significant feature has a mutual information score of at least 0.2.
3. A feature will be used in the model if the information they share are not trivially similar. Instead features created must be capturing different behaviours in the data than those already. To provide a concrete

example although data can be linked, cycle and cycle run, there is not a simple arithmetic operation that can directly map one data point to another- unlike month and quarter.

The features produced and an explanation as to their origin and the intuition behind their creation is given below.

8.1 Created Features

Last_Run

The last time that the test case was run, but converted into the python datetime integer such as this: 1480098840. This captures the date as the format of input features can only be numeric value and not a string which was previously not usable in the dataset. An important inclusion as if a test case has been run very recently, it follows that it is more likely to be run again.

Failure_Percentage

A missed, but crucial feature in the original dataset. The last results array cannot be directly used by the algorithm as only numeric inputs are accepted. On deciding whether a test case will fail or not, the historical percentage of this will become an invaluable marker. For example, if a test case has failed in every previous test, then it is paramount that the test case is ranked very highly.

Times_Ran

The total number of times a test case had been historically run. This feature has not be directly captured in the dataset previously, but can be easily calculated by calculating the length of the last results array and is important for ranking as follows: If a test case has been run many times, it is likely that this test case is crucial to the software and will have failed more often so should be ranked highly for prioritisation.

last_result

The immediate last results of the previous test case. Another feature not previously captured by the dataset, but a necessary addition. If a test failed their immediate last test case, then it may be likely that the test case will fail again.

gap_in_run

The difference in time between the last two runs of the test case. Calculated using the last_run feature. This feature was created with the intuition that if there is a large gap between test case execution, the test case is being run less frequently. This is to provide additional depth to times_ran, because the times_ran will not capture disparity in execution time, and may place tests with a high intensity at the start, but not towards the end, in the same category as those with a constant execution throughout.

gap_cats

The difference in days, with a maximum value of 3, between the previous and penultimate test case execution. This has a similar intuition to gap_in_run, but with the intent to discretize the data. The reasoning is that tests that have been run more than three days ago are less likely to be tested again, compared to a test case that has been tested the very same day.

in_same_cycle

The value will be one if the previous execution of the test case was in the same cycle as the current test case record, 0 otherwise. This feature has a very similar intuition in gap_in_run, but captures different information to cycle and gap_in_run and so was a necessary inclusion.

cycle_run

Contains the running total of runs of a test case in the same cycle. The intuition is that if a test case is being run more often in the same cycle, it is more likely to return in failure.

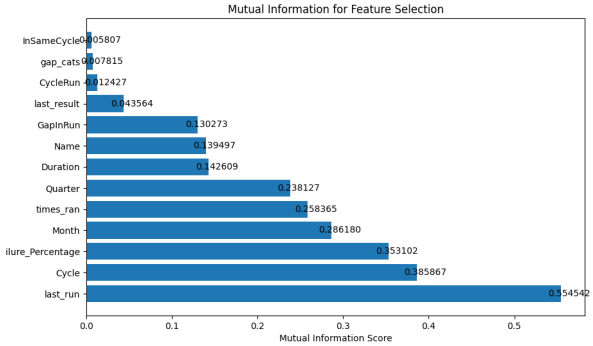

Figure 7: Mutual Information results results after feature creation : calcPrio

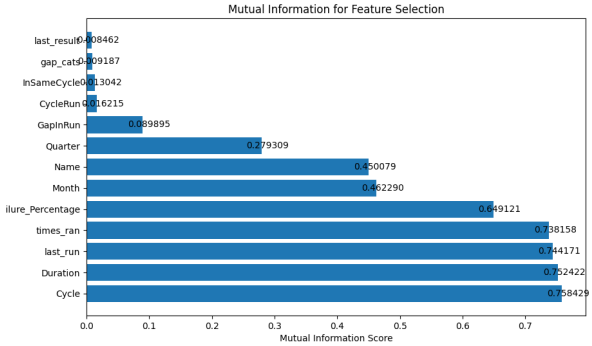

Figure 8: Mutual Information results results after feature creation : rank

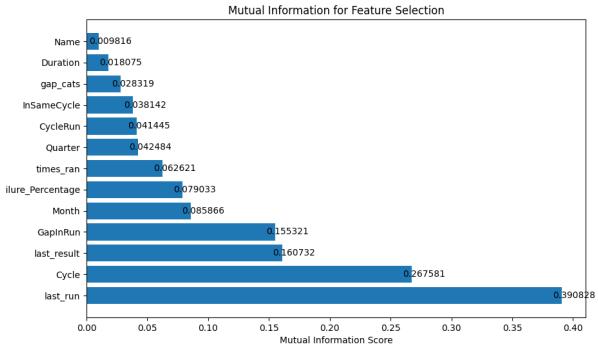

Figure 9: Mutual Information results results after feature creation : verdict

After creation of these features it it clear to see that the extraction of **last_run** was very successful, sharing a very high mutual information with every target feature, becoming the highest for **calcPrio** and **verdict**. Similarly for **times_ran**, improving on more than half the values in all target features.

Gap_in_run also performed to a good standard, which could be attributed to sharing information with an already good performing **last_run**. Unlike **month** and **quarter** which are trivially similar, **gap_in_run** captures unique information to **last_run**.

Meanwhile **gap_cats** and **in_same_cycle** returned poor results across all target features. This could be because there is such a large range of potential gaps in test cases that discretizing creates information loss. Bins may contain datapoints that belong to a separate class.

The previous features involved extracting new features from the data but the next feature is comprised of an arithmetic combination of existing features.

8.2 Iterative Enginerring of Mutual information

The desired feature is called maturity level, with the name inspired from the original pre-engineered feature. The goal is to produce a feature with a higher mutual information in relation to verdict than cycle, while also sharing high levels of mutual information with the other target variables.

An intuition laid out as follows:

For something to be mature, it must

1. Have been run many times when compared to the minimum and maximum execution times in all previous tests
2. Be run towards the end of the testing process
3. Have a high failure percentage rate

Summarised results from more than 50 combinations trials were as follows:

Combination	Results	Discovery
$month * fp * times\_ran$	0.1149996351707161	None
$\frac{month*fp*times\_ran}{(MOST\_TIMES\_RAN)}$	0.11534824013197031	Comparing against the most times is an improvement
$\frac{fp*times\_ran}{(MOST\_TIMES\_RAN)}$	0.06267210464346462	Month is essential for maturity level
$\frac{month*fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.2149909238654341	ensuring the minimum times ran is taken into account
$month + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.24650071849576527	Adding features works well also
$month + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.24650071849576527	Adding features works well also
$\frac{month}{in\_same\_cycle+1} + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.25007521405153454	The creation of in same cycle was a success
$\frac{month}{in\_same\_cycle+gapcats+1} + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.2558423750783039	The creation of gapcats was also a success
$\frac{month*(in\_same\_cycle+1)}{gapcats+1} + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.2619157726714434	Multiplying in_same_cycle gives a better score
$month * cycle + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.3410313651754393	The inclusion of cycle works better then other gapcat and cycle_run
$\frac{month*cycle*(in\_same\_cycle+1)}{gap\_cats+1} + \frac{fp*(times\_ran-LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN}$	0.357122583825527	The previous results are still effective

Table 2: Maturity Level Trials

Note that MOST_TIMES_RAN and LEAST_TIMES_RAN represent the minimum and maximum count of how often an individual test case is executed.

We select the maturity level calculation with the highest mutual information and use that as the produced feature.

$$my\_maturity = \frac{month * cycle * (in\_same\_cycle + 1)}{gap\_cats + 1} + \frac{fp * (times\_ran - LEAST\_TIMES\_RAN)}{MOST\_TIMES\_RAN} \quad (12)$$

With the maturity level calculated, and the metric enforced, the features used for the prediction of each target feature are graphed below.

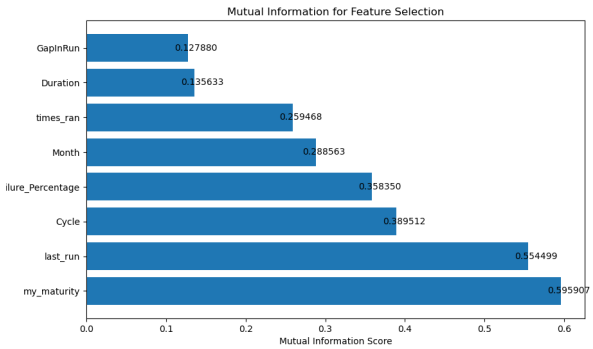

Figure 10: Final feature selection with labelled mutual information : calcPrio

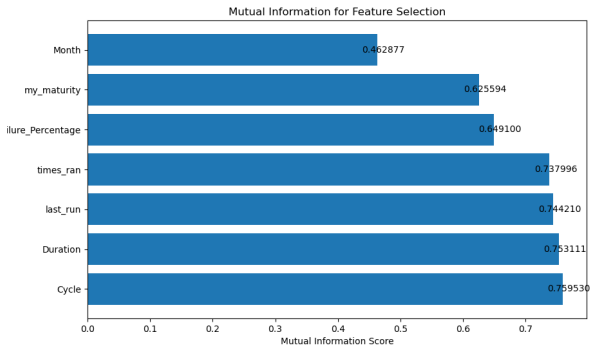

Figure 11: Final feature selection with labelled mutual information : ranking

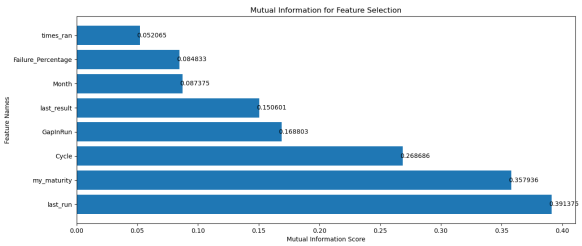

Figure 12: Final feature selection with labelled mutual information : verdict

9 Prioritisation

In order to rank the test cases, we must first organise the test cases into priority groups. The data used in this paper has three priorities and thus we will use 3 explanatory examples.

Priority 3 Tests that **must** be executed before release of the software. These tests check for application features which are integral to functionality. If these tests are not executed then there is a very high risk that the program will not run as expected or an error may emerge after the software is released.

Priority 2 Test cases that **could** be executed if there is enough time in the testing schedule. These tests are not critical to the functional requirements of the product, but should be double checked before launch as a best practice.

Priority 1 Test cases that are **not necessary** to be tested before the release of the product. Best practices would state that they should be tested after the release of the current version, but the product is not dependant on it.

Within the systems software development processes mentioned previously, the software development process can take an extended amount of time to complete. There is a strict quota on the time that can be allocated to individual areas due to release deadlines internal to the company and pressures from clients. Perhaps a development team is only given three months to test a project, and therefore only have time to test the most crucial test cases, in this example, they would only test cases in priority 3.

9.1 Methods

To prioritise the test cases, we will be using numerous supervised learning techniques to classify the data into three categories, guided by findings in previous work.

After engineering the features discussed in section 8, the top 6 features were selected as features to be utilised in the classification algorithm: **my_maturity, last_run, Cycle, failure_percentage, Month, times_ran**

At first, following good results from Sharma & Chande (2023), we made use of a random forest classifier. This method handles high dimension data effectively, its ensemble learning characteristic can be utilised improve prediction accuracy and is robust to noise and outliers. These characteristics fit the problem at hand and is the reason for its initial selection.

Models attempted are as follows:

Model selected	Reason for selection
Random Forest Classifier	This method handles high dimension data effectively, its ensemble learning characteristic can be utilised improve prediction accuracy and is robust to noise and outliers.
Bayesian Classifer	A very fast algorithm, will not add high overhead computation to the latter ranking feature. Can also perform much better than other models when the size of the data is very small, because of the lack of size constraint, the model can hence be applied in more scenarios than other models.
SVM	Is reported to be efficient and effective in industrial settings which matches the source of the used dataset. Marijan (2022) reports SVM has the joint best time efficiency in its fault detection.
GBRM	Marijan (2022) reports that this model has the join best fault detection rate accuracy in their study.
Voting Classifier	Cao et al. (2022), concludes that ensemble based methods, in the application of test case prioritisation, can outperform their base algorithms.

Table 3: Models selected for test case prioritisation
Sharma & Chande (2023) Vadapalli (2022)

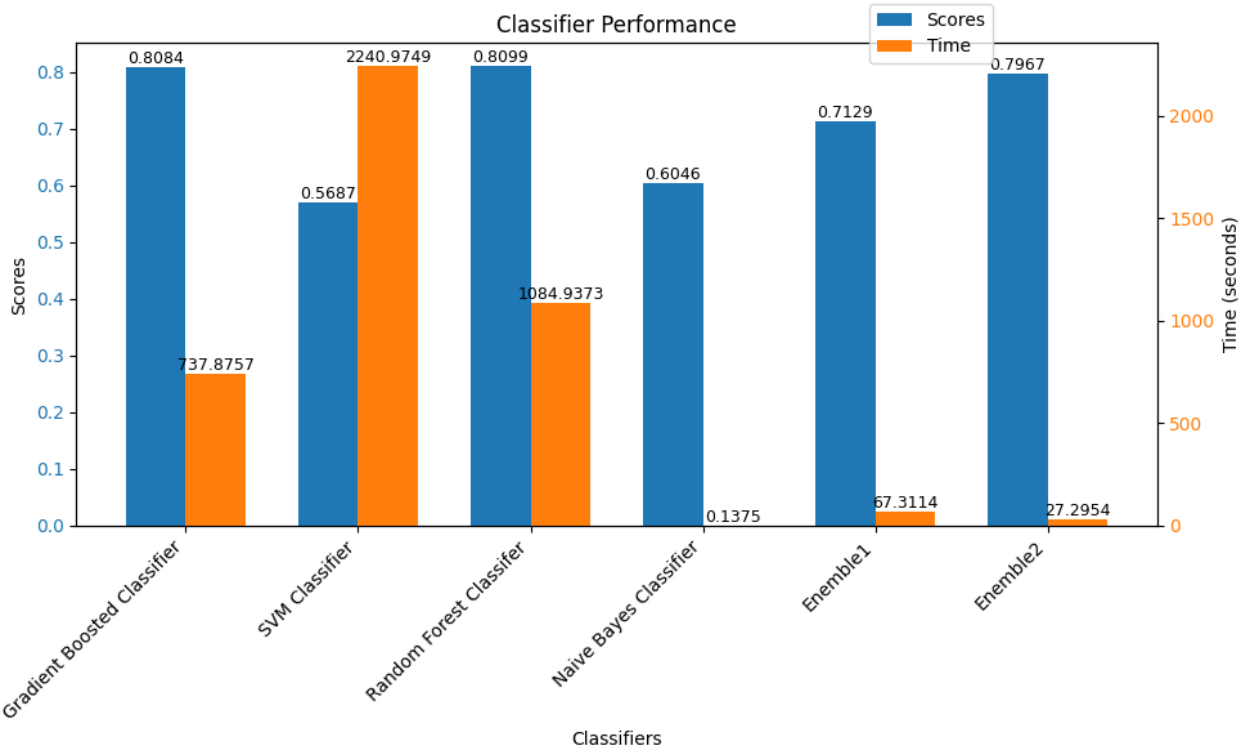

Figure 13: Initial Classifer Results

The performance metrics used on these models are running time and prediction accuracy. In order for theses models to be selected, they must pair well with the goals of this project. The prioritisation cannot add an excessive amount of running time to the prioritisation algorithm such that its use does not outweigh the cost of simply running the test cases in an untreated order. Similarly, the necessity of accuracy of the prioritisation is trial, the correct test cases must be chosen for ranking otherwise risking extended testing time, or the testing of redundant cases.

Models are trained using an 80-20 training testing split to avoid the problem of data over fitting.

Figure 13 shows the results achieved after the first round of classification attempts. The random forest classifier and the Gradient boosted classifier both perform similarly with an accuracy score of 0.799 rounded to 3 decimal places, followed by Naive Bayes with 0.467 and the SVM classifier with 0.420. With low accuracy scores, these models cannot be successful in a prioritisation context, and cannot be selected. Note here that there are 2 voting classifiers. During the testing, it became clear that the voting classifier making use of all the classifiers performed weaker than some of the "children" classifiers. To negate this issue, a voting classifier with only the top 2 performing models was also included into the results collection, providing better results.

A potential reason to the low performance of the voting classifier here is that 2 of the children models have a considerable lack in performance, which may introduce a bias towards a poor selection in the voting procedure.

Ensemble 1:

$$GradientBoostedClassifier, SVMClassifier, RandomForestClassifier, NaiveBayesClassifier$$

Ensemble 2:

$$GradientBoostedClassifier, RandomForestClassifier$$

In order to improve these results, a hyperparameter grid search used for every model in the selection. To decide which parameters of the models have the greatest performance, cross validation with 5 folds is used. The parameters that return the highest model performance are then fitted to the testing data and an accuracy score is outputted. This highest performing model is then used as a child in the voting classifier alongside the others. Here, the 20% split of testing data is then used to evaluate the model performance.

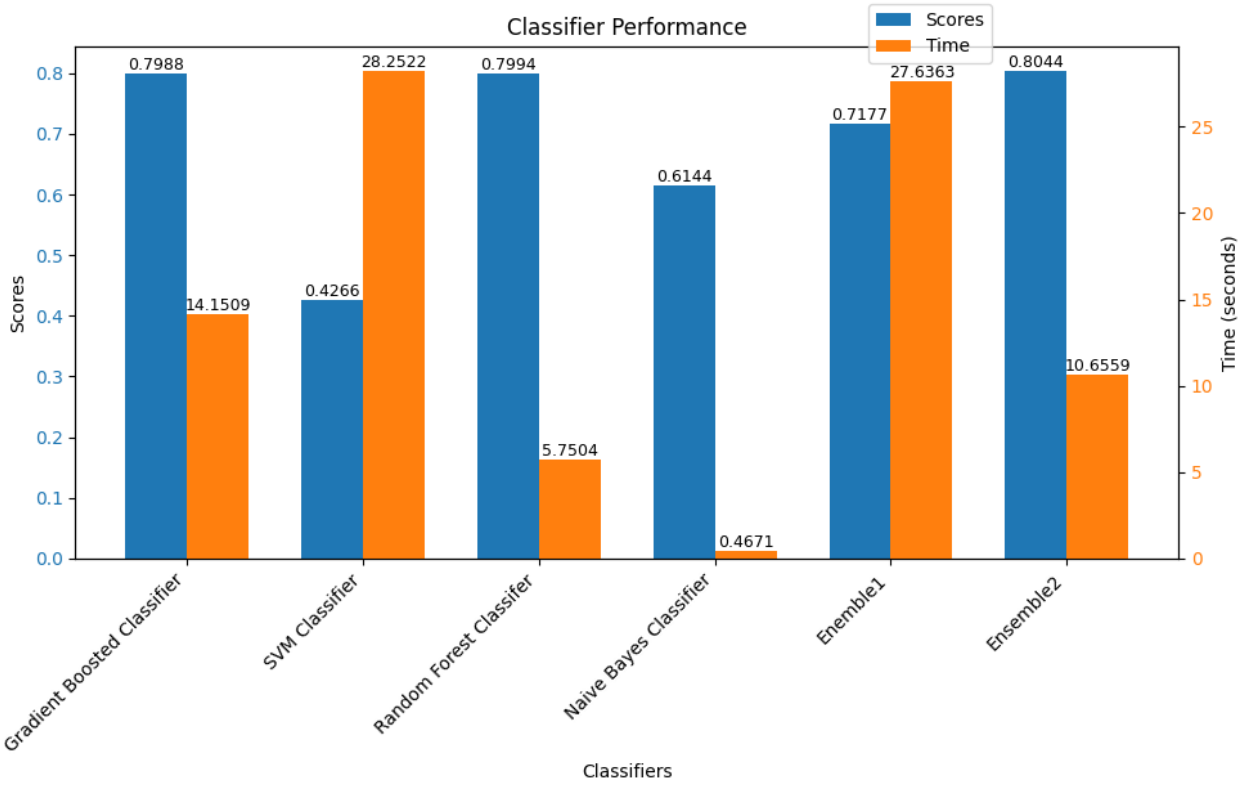

Figure 14: Classification models after grid search

Model	Parameters Chosen
Gradient Boosted Classifier	learning_rate: 0.15, loss: log_loss, n_estimators: 170
Random Forest Classifier	max_depth: 10, min_samples_split: 5, n_estimators: 30
SVM Classifier	C: 0.4, decision_function_shape: ovo, gamma: scale, kernel: poly
Naive Bayes Classifier	-

Table 4: Parameters Chosen

The grid search results lead to an interesting discussion. It is clear to see that the performance of the gradient boosted regression model, as well as the random decision forest have improved, but only by a marginal value. Meanwhile the SVM, has improved a considerable 56.87%, while still maintaining a very low performance score. Naive bayes, had no parameters to tune and therefore an comparison cannot be made in this instance. However, Stenvatten (2020) reports that both gradient boosted decision trees and random forest classifiers are both capable of accuracy score of above 90%, while Mohd Radzi et al. (2022) produce comparable results for random forest, and add that naive bayes can achieve scores of more than 90% on some datasets. In fact, results

similar to this can also be concluded from Xhemali et al. (2009) and not specified (not specified). Although the datasets are not the same, it is indicative of the potential of the classifiers at hand.

Therefore, it was clear that the performance is not to do with the models selected, but rather the data inputted. Although the mutual information of the inputted features and the priority level are high, perhaps the issue lies within the fact that there is still a large amount of combined information missing from the selection. The combination of GapInRun and Duration both add up to more than Month, times_ran and is just less than the score for Month(shown in figure 11).

To improve these results, the input features for the classifiers were revised. Previously, the features selected are **my_maturity, last_run, Cycle, failure_percentage, Month, times_ran**.

The next two features with the highest mutual information with CalcPrio are included in the selection: Duration and GapInRun. As before, grid search with cross validation is applied to these models and the results are shown in figure 15

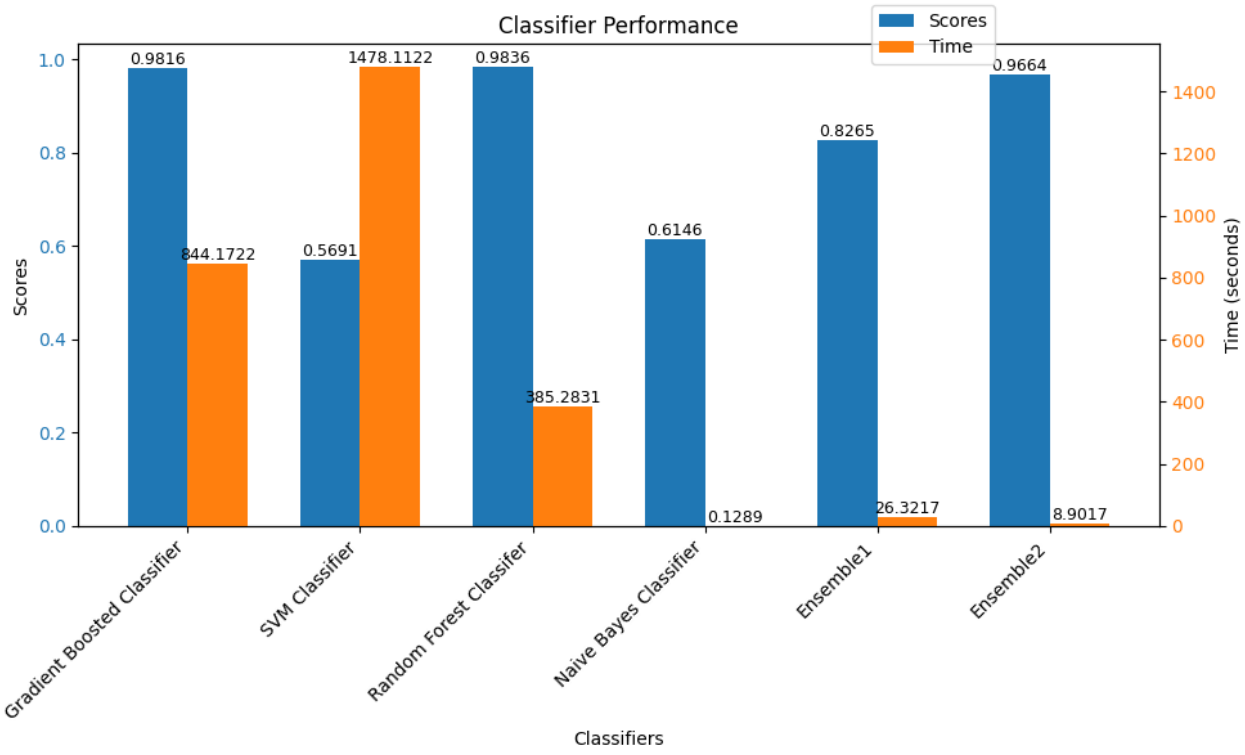

Figure 15: Classification with GapInRun and Duration

The parameters tuned to be included are:

Model	Parameters Chosen
Gradient Boosted Classifier	learning_rate: 0.2, loss : log_loss, n_estimators: 270
Random Forest Classifier	max_depth: 20, min_samples_split: 2, n_estimators: 60
SVM Classifier	C: 1.6, decision_function_shape: ovo, gamma: scale, kernel: poly
Naive Bayes Classifier	-

Table 5: Parameters Chosen after Grid Search

Although cross validation2 has the best performance to execution time ratio, the parameters of both child models need to be executed before this. Therefore, the running times of both voting classifiers in the graph are not accurate. The classification model to be selected is therefore that of the Random Forest Classifier, for its superior cross validation time and accuracy.

10 Reinforcement Learning

10.1 Learning Process

During University studies, the syllabus covers Supervised and Unsupervised ML techniques but does not teach RL. Therefore in this paper, the concepts of the model must be learnt from scratch. The sources of information used are previously discussed in 3.1. To implement the RL model I adapted code from Paszke & Towers (n.d.) and studied the documentation of the Gym environment to understand how to implement an environment myself. Other learning environments included Bagherzadeh et al. (2021) and their git repository linked on their paper. From here, I then produced my own environments (discussed below). The steps taken to implement a working solution will be detailed, as well as discussing the iterative attempts at improving the quality of the solutions.

10.2 Environments

An RL environment E dictates how an agent learns as it contains these three concepts: action space A, reward space R and state space S. Two learning environments are applied in this paper which will be detailed below. These environments are selected due to to the capability of the Deep Q network model utilised in the research, the model historically does not perform well when predicting continuous action spaces and therefore only environments with a discrete action space are used.

However, in the aforementioned paper, Bagherzadeh et al. (2021) utilises the below methods and it is this which will be continued in this work.

With each iteration, the agent takes a "step" which involves the environment E supplying an input state s_t, the optimal action for that timestep a_t, an observation s_{t+1} and the reward for reaching that state r_t. After which the current state is updated to the observation and the process continues until the episode is complete.

10.2.1 List Wise

The listwise environment can see the entire test case execution history at once. The algorithm will predict the highest rank element from that list and place it in a final ranked array L_{out}. Then the agent will predict the next highest ranked element and the process is continued. If the agent picks an element that has already been selected, then the episode will be terminated.

10.2.1.1 State Space

As the listwise ranking is predicting rank, the input features will be features engineered and showcased in section 8, as well as a dummy feature.
Let F_d = Dummy feature of record F

$$F_d \begin{cases} F_d = 1 & \text{if the record has been previously selected and placed into the final list} \\ F_d = 0 & \text{otherwise} \end{cases}$$

f_d is an essential part of the design, such that the agent will learn to not pick the test case more than once.
With a sequence L of test case execution records, of length n -
$L = [F_1, F_2, ..., F_n]$ where F = one test execution record. The feature names are thus as follows
$\forall f \in F, f \in \mathbb{R}, f_{name} = [Cycle, Duration, last\_run, times\_ran, Failure\_Percentage, my\_maturity, Month, Dummy]$

10.2.1.2 Action Space

The agent selects one record F from L each step to place in the output list : L_{out}. Therefore the action is the index of the selected record F from L - $a \in [0, n-1]$.

10.2.1.3 Reward Space

The optimal ranking of L is known: L_{opt}. The agent will be rewarded the closer the L_{out} is to L_{opt}.

For this metric, and following Bagherzadeh et al. (2021) previous work, the agent is rewarded based on the mean squared error of the predicted rank, to the optimal rank . Furthermore, in order to be able to compare with differing datasets with differing sized lists, like ones used by Bagherzadeh et al. (2021), the ranks must first be normalised.

$r_t = 1 - (a_t - L_{opt}[SelectedFeature]_{rank})^2$.

10.2.2 Pairwise Verdict

The pairwise environment's aim is to directly compare two test case execution records and perform a sorting procedure on which test case is more likely to fail, with the verdict estimated by the deep Q network utilising the input features discussed below. If both test case records are predicted to fail, then a tiebreaker of which test case record has the shortest duration is applied. This sorting process consists of an iterative and step wise quick sort algorithm which will be explained below. Although Quicksort is a recursive algorithm, recursion does not match the iterative nature of Reinforcement Learning training process and is adapted to provide a usable interface. The algorithm in direct comparison for selection that of the merge sort, a sorting algorithm with the same time complexity $O(NLogN)$. Throughout multiple sources, Quicksort is said to perform well in different smaller datasets, but has an improved space complexity to merge sort, and is discussed to perform better on well when data does not need to be read from an external memory. In terms of the general case, the universal verdict is that it greatly depends on the exact problem at hand. The aim of this paper is to implement a RL approach and improve upon it using genetic methods, and so the analysis and choice of an optimal sorting algorithm for this environment is not necessary but is potentially a focus for future work (Al-Kharabsheh et al. (2013), Sabah et al. (2023), Oladipupo et al. (2020), Mishra & Garg (2008)).

10.2.2.1 State Space

An overview of the state space is as follows: **Features of record1 , Features of record2**. Let $F = [f_0, f_1, ..., f_8]$ where F is the set of features for one record.
Let $F_{name} = \{Cycle, my\_maturity, Month, Failure\_Percentage, GapInRun, last\_run, last\_result, times\_ran, Duration\}$

$\forall f \in F,\ f \in \mathbb{R}$
The state space is then $S \in \{\text{record1} \in F, \text{record2} \in F\}$

10.2.2.2 Action Space

Action a is defined like so:

$$a \begin{cases} a = 1 & \text{if the two test execution records should be swapped when sorting} \\ a = 0 & \text{otherwise} \end{cases}$$

10.2.2.3 Reward Space

The reward space is designed with the intention to reward the agent if the decision is to swap the test case with the predicted failing verdict; or if the verdicts are the same, then predict the test case with the shortest duration. Hence:
let $f_0 =$ the record placed higher after the agent completes one step of the sort
let $f_1 =$ the lower ranked record
let $g_v =$ predicted verdict of g where $g \in \{f_0, f_1\}$

$$r \begin{cases} r = 1 & \text{if } f_{0_v} > f_{1_v} \text{ and } f_{0_v} \neq f_{1_v} \\ r = 1 & \text{if } f_{0_v} = f_{1_v} \text{ and } f_{0_{Duration}} < f_{1_{Duration}} \\ r = 0 & \text{otherwise} \end{cases}$$

Note here that the reward when the verdicts are the same and the algorithm is then comparing the duration is the same- both at one. This design choice was made due to the fact that within the dataset, it is very common to have verdicts that are the same and thus the model should put train effectively to know how to deal with such cases.

10.2.3 Pairwise Rank

Very similar to the pairwise verdict approach as mentioned above, but this environment predicts and directly compares the ranks of two test case execution records. The records are sorted in order of lowest rank. This Environment was included as a potential supplement to the listwise ranking approach, due to results discussed below.

10.2.3.1 State Space

As the Pairwise Ranking approach is also predicting the rank of test case execution record, the state space is the very similar to listwise. The feature names are thus as follows
$\forall f \in F, f \in \mathbb{R}, f_{name} = [Cycle, Duration, last\_run, times\_ran, Failure\_Percentage, my\_maturity, Month]$

10.2.3.2 Action Space

Action a is defined like so:

$$a \begin{cases} a = 1 & \text{if the two test execution records should be swapped when sorting} \\ a = 0 & \text{otherwise} \end{cases}$$

10.2.3.3 Reward Space

In the pairwise environment, the reward space is designed to reward the agent agent if, after the swapping has taken place, the test case execution records are ordered correctly in ascending order of ranks Hence:
let f_0 = the record placed higher after the agent completes one step of the sort
let f_1 = the lower ranked record
let g_r = predicted rank of g where $g \in \{f_0, f_1\}$

$$r \begin{cases} r = 1 & \text{if } f_{0_v} > f_{1_v} \text{ and } f_{0_v} \neq f_{1_v} \\ r = 1 & \text{if } f_{0_v} = f_{1_v} \text{ and } f_{0_{Duration}} < f_{1_{Duration}} \\ r = 0 & \text{otherwise} \end{cases}$$

10.3 Pseudocode

Pseudocode detailing the differing environments is detailed below.

Algorithm 1 Calculate Reward

```
function CalculateReward(action)
Require: the current pair of test case execution records: state1, state2
    Verdict1, Verdict2, Duration1, Duration2 ← orderCasesFromAction(action, state1, state2)
    if Verdict1 = Verdict2 then
        if Duration1 < Duration2 then
            return 1
        else
            return 0
    else if Verdict1 > Verdict2 then
        return 1
    else
        return 0
```

Algorithm 2 Step Function for the Pairwise Environment

```
Require: the current pair of test case execution records: state1, state2
function Step(action)
    done ← False
    reward ← CalculateReward(action)
    if action = True then
        Perform 1 step of the quicksort algorithm
    if quicksort has finished then
        done ← True
    state1, state2 = update current pair of test case execution records
    return observation, reward, done, {}
```

Algorithm 3 Step function for the Listwise Environment

Require: current rank that the environment is working at: *currentRank*
Require: number of records being ranked: *numRecords*
Require: actions that have already been taken: *preSelected*
Require: current list of ranked records: *observation*

function STEP(*action*)
    done ← False
    truncated ← False
    reward ← CALCULATEREWARD(*action*)
    if *currentRank* > *numRecords* **and** *action* **not in** *preSelected* **then**
        add *currentRecord* ← 1
        currentRank ← *currentRank* + 1
    else if *currentRank* = *numRecords* **then**
        done ← True
    if reward = −100 **then**
        observation = *None*
        truncated = *True*
    return *observation*, *reward*, *done*, *truncated*

Algorithm 4 Calculate Reward Function for listwise environment

Require: current rank that the environment is working at: *currentRank*
Require: number of records being ranked: *numRecords*
Require: actions that have already been taken: *preSelected*
Require: current list of ranked records: *observation*
Require: function which returns the optimal rank for given action: *getOptimalRank*

function CALCULATEREWARD(*action*)
    if *action* **in** *preSelected* **then**
        return 0
    else
        optimalRank ← GETOPTIMALRANK(*action*)
        normalisedOptimalRank ← *optimalRank*/*numRecords*
        normalisedAction ← *action*/*numRecords*
        return 1 − (norm_action − *normalisedOptimalRank*)^2

Algorithm 5 Deep Q learning Training Loop

procedure DEEPQLEARNINGTRAINING
  Initialize replay memory M with capacity N
  Initialize policy network Q with random weights θ
  Initialize target network $\hat{Q}$ with weights from Q
  Initialize environment state s_0
  for *episode* $\leftarrow 1$ **to** M **do** ▷ Loop over episodes
    Reset environment to initial state s_0
    Set total episode reward $R \leftarrow 0$
    for $t \leftarrow 1$ **to** T **do** ▷ Loop over time steps
      Select action a_t using ϵ-greedy policy based on Q
      Execute action a_t, observe next state s_{t+1} and reward r_t : executed by step function
      Store transition (s_t, a_t, r_t, s_{t+1}) in M
      Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute the state action values of the transition batch from the policy network
      Compute the expected state action values for each batch transition using the target network:
      $$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$
      Update the policy network Q by minimising the Huber loss:
      $L = \frac{1}{|\text{batch}|} \sum_j \text{Huber}(y_j - Q(s_j, a_j; \theta))$
      Update target network parameters:
      $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$
      Accumulate reward: $R \leftarrow R + r_t$
      Move to next state: $s_t \leftarrow s_{t+1}$
      if episode terminates **then**
        break
    Decay exploration rate ϵ

10.4 Training Setup

To ensure that the models received enough training time, each appearance of a training instance must have at least 50 training episodes for pairwise, corresponding to $50nlog_2n$ training steps for pairwise. Listwise was decided to have at least 1000n steps - 1000 per training instance in order to accommodate for the more complex listwise mode. The training stops once the model has not improved for 10 testing procedures. For pairwise and listwise the testing procedure occurred very 1000 and 2000 steps respectively.

10.5 Challenges in production

A deep Q network has a certain number of inputs. Trivially, the larger the number of inputs the network will have, the complexity of training the model is much larger. Listwise has $8n$ inputs, where n is the size of the test execution record list. From the results concluded in Bagherzadeh et al. (2021), the training time for listwise was on average 265% longer than that of pairwise for each dataset that was trialled. Because of this, to maintain within the strengths of deep Q network, it was decided that the ranking should also be completed using a pairwise ranking method

The initial results when running these environments are as follows.

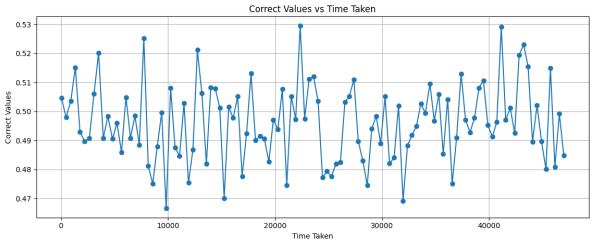

Figure 16: Original Pairwise Procedure

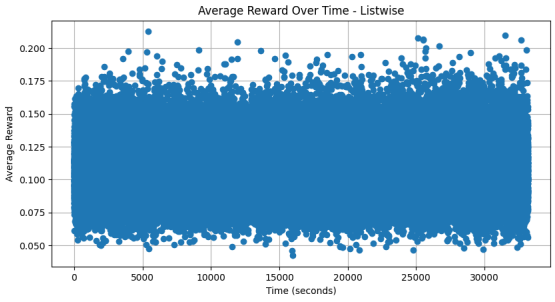

Figure 17: Original Listwise Procedure

It is clear to see, that over an extended period of time, the pairwise and listwise average reward was not increasing. The pairwise agent is seen to be "guessing" right or wrong and nothing more than that. A similar

action is happening with the listwise agent.

One reason this could be is because the algorithm was using solely the default parameters. Therefore, for the next step it was decided that these parameters needed to be tuned.

Parameters to tune

1. **Eps decay level**- this parameter dictates how quickly learning temperature decreases - a larger value meaning the agent is in the exploration phase for much longer- while lower means the agent will exploit the already learnt, highest value actions. **Range** from 1000 to 1000000

2. **Learning Rate**: This is the learning rate of the optimiser function used (AdamW - *adamW documentation* (n.d.)). This variable controls the step size of the optimiser function used in place of gradient descent to update the weights of the RL model. One reason the model was not learning is could potentially be down to how large of a step size the optimiser had. A step size too large or too small, and the agent will never converge to a local optimum. **Range**= 0.1 - 1e-8

3. **Memory Size**. If the replay memory size is too small, the agent risks sample inefficiency, and catastrophic forgetting, where old experiences are overwritten too quickly. **Range** - 100 - 1000000

4. **Number of Hidden Layers** - With one layer of each neural network, the capacity for learning complex relationships between input features is potentially not strong enough. The agent may underfit to the data at hand and may have been the weakness in the agents performance.**Range** - 1 - 5

5. **Changing the reward structure of the pairwise comparison** - It was initially set that the reward that the agent receives was 1 for a correct, 0.5 for if the entry with the smallest duration was selected in the event that the verdict was the same and 0 for an incorrect choice. The hypothesis was that the agent would prioritise the learning of the duration prediction characteristic and thus slow down the learning process

6. **Running Time** - Bagherzadeh et al. (2021) states that it took 2021 minutes to train their DQN model in the pairwise model. In an attempt to match that performance, it was theorised that in comparison, my model did not have enough training time and such this was experimented with.

However, after trialling these methods, it was found that the performance of the model did not improve, and continued to act in an appeared random state. After further research, the issue lay partly with an incorrect driver issue, when using the PyTorch library, and as well as an implementation error stemming from an unfamiliarity with the pytorch framework, causing the Q network to silently never update. After fixing these issues, the timeline of the project was shifted by some weeks, and the focus then rested on implementing the evolutionary improvement. Therefore, parameter and systematic tuning are to be left as a focus for future work.

10.6 Normalisation

Once the Q values were improving, it became clear that they were improving at a very large rate. At the completion of one pairwise training episode of length **80000** steps, the Q values were averaging to be **2.7e+08**. In conjunction with this, the performance of both pairwise and listwise environments using the original data was poor. Figures 18, 19, 20 The reason for this can be summarised down to two reasons. Lan et al. (2020) states that estimation bias, also known as bias in estimation, refers to the "systematic error or deviation of an estimator from the true value it is trying to estimate". Deep Q learning is a maximum function estimator, and thus in the the context of the problem at hand, estimators will make errors of prediction which are down to the stochasticity of the action. When an input value is large, the mistaken estimation is amplified and can lead to a divergence from the optimal policy- because it is unclear whether or not the best Q value is deserved of its position. Due to large input values these errors are compounded to larger and larger errors as the learning continues. Secondly, if there is a big disparity between the sizes of features, the deep Q learning will become unstable as it will have a tendency to associate a higher level of importance to larger features - promoting a suboptimal policy that is learnt.

From this the features input into the system are analysed:
Last run is identified to be much larger than other input values with an example feature of 1480098840, a minimum of 4 powers of 10 difference. To fix this issue a solution was proposed- normalisation of the data.

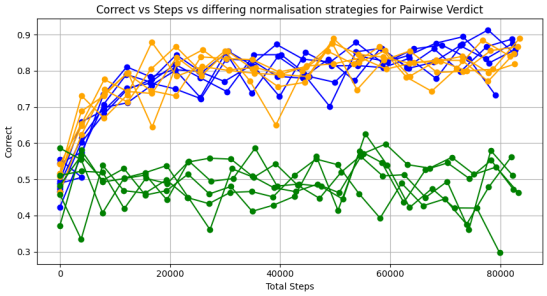

Figure 18

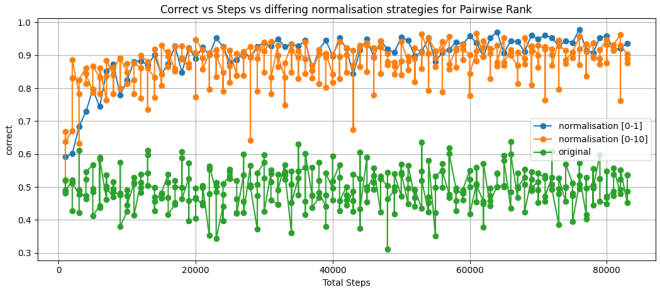

Figure 19

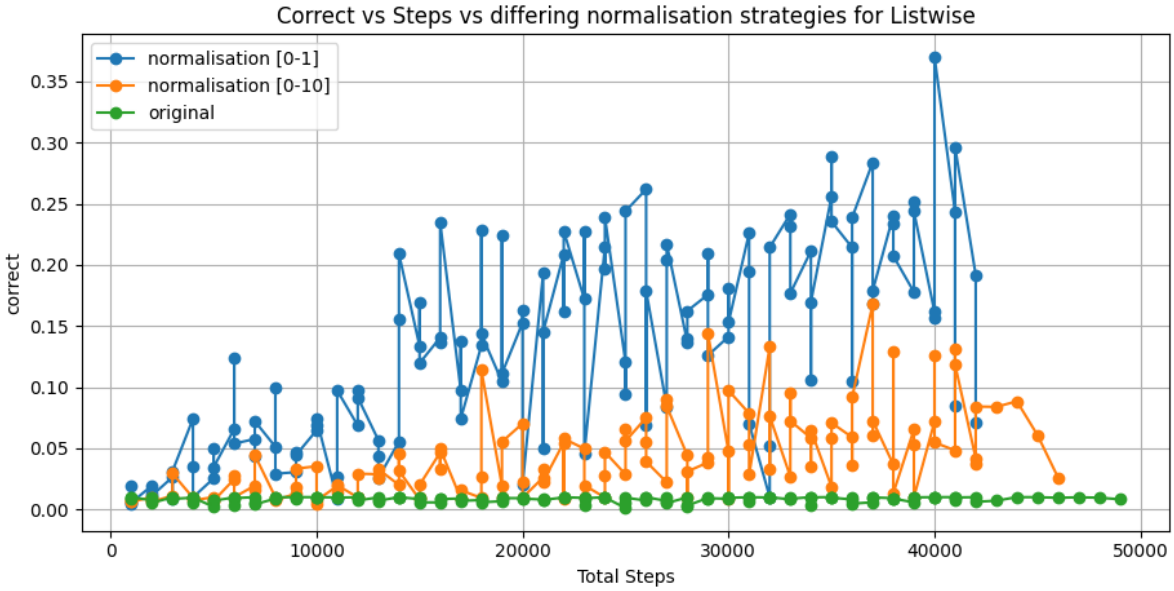

Figure 20

After applying the normalisation, *Gap_in_Run* as well as *Last_Ran* assumed decimal values in the range of 0.0005 and below. In order to avoid an underestimation bias, similar to overestimation bias as mentioned above, these decimal values were then logged.

Above show the findings of these results gathered over 5 training episodes for each environment normalisation combination. The following normalisation techniques are detailed, normalising all values between 0-1 (blue), normalising all value between 0 and 10 (orange) and keeping the data untreated (green). Overall, normalisation between 0 and 1 proved the most effective technique when comparing the metric of average reward values received per agent as is therefore the one used in the rest of the paper. More precisely, the average reward after 1000 epochs was , Pairwise Verdict - 0-1 normalisation - 0.7821994736842105, 0-10 normalisation - 0.762, original 0.491, Pairwise Rank - 0.900 0 - 1 normalisation, 0.872 0 - 10 normalisation 0.497 original, and Listwise 0.132 0-1 normalisation , 0.0405 0 - 10 normalisation, original 0.00846. 0-1 Normalisation although being very similar to 0-10, is more likely due to the above mentioned over estimation bias. Even with slightly larger values, the difference in accuracy is stark, and the compounding of error is clear to see. The larger difference in error for the listwise approach could be attributed to the fact that, unlike a pairwise models 50% chance that the estimated prediction is correct, the listwise agent needs to be very precise with its selection and thus the gulf in accuracy is enlarged.

11 Genetic Improvement

In this section, a genetic algorithm will be employed with the aim to improve the performance of the original RL algorithm. The metrics of improvement considered are: reward percentage, running time (note here that because of differing machines used to train and test these models, the running time will be measured in number of steps executed), average percentage fault detection , Normalised Rank Percentage Accuracy. The final three metrics are calculated in comparison with the "optimal sequence" L_{opt}.

The proposed solution, described in this section, details use of a GA to evolve a population of policy networks used in the RL algorithm. The networks will be evaluated on their average reward return on a shared input, and then based on their return, will be selected by the GA to produce offspring and be included in the next generation of solutions. We will explain the motivation behind genetic algorithms for improvement, and then explain how the algorithm works below making use of pseudo code. As multiple RL agents will be run simultaneously, multithreading is employed to boost program running time.

11.1 How GA will help RL

GAs promote only positive outcomes. During the evolution process, solutions with a low fitness are less likely to be selected in the selection process to continue to the next generation. When a particular action enables the model perform much better than others, the action space similar to the high performing model is much more likely to be explored through the use of its children. In contrast, RL on its own cannot achieve this feat. For this to take place, the model would first require a very fast converging epsilon temperature. This feature promotes a greedy selection policy, and minimising the effectiveness of exploration partition of the agent's learning process. Without this exploration, the model will become stuck in a local optimum with a high probability of poor performance.

Therefore their combination would offset this identified problem in an RL learning agent, allowing for a faster exploration of solution space, and faster convergence to a more powerful model.

11.2 Maintaining Shared Memory

The first problem encountered when designing a solution was that of the shared memory. As explained in section 3.1, an RL agent learns by continuously sampling from a memory bank of past "transition" to eradicate critical forgetting of previous events, and to retrain on these past events for a more stable learning process. When a network is evolved, the memory of parent must pass onto the children.

The problem with this is that a direct copy of the memory cannot be made. Here is the process:

1. By simply using the parent's memory as the child's memory in the child object, due to python's 'passing by reference' and that the memories will share the same memory address, if one of them pushed a transition to memory, both agents would experience that transition. As the agents have diverged, this memory sharing is not representative of the learning process of each agent.

2. The next conceived solution would be that of using python's copy module. This cannot be used due to the additional complexity overhead the solution comes with. In the implementation, the ReplayMemory object can store up to 100000 transitions in memory for learning. Note that these transitions are objects themselves, and as such also have a notable individual storage size. When producing offspring, the entire 100000 transitions must be copied directly, resulting in a large amount memory in use for ReplayMemories from all citizens in the population. With a large test execution record suite, this large space complexity and copying overhead is not feasible.

3. To solve this, the below solution is proposed. A "memory node" (MN) here has 2 main parts, its working memory and its old memory. When the MN experiences transitions, they are stored in the working memory. However, when that node must produce offspring, the working memory must be shared between the parent and the child. To avoid either MN from pushing memory to the shared memory, the parent's previous working memory is pushed into a read only memory stack for both MNs. Figure 21 shows the path between the memories, and the high level structure of each node. When the memory needs to be sampled from, 21 also depicts the memory segments in which the memory nodes will sample from. The "Child" memory in the figure shares memory with its parent and its grandparent and must sample from combined shared memory. Figure 22 shows this in further detail. The top level in a memory node. The MN must produce a child and therefore its working memory is pushed to the bottom of the old memory stack, such that that working memory can no longer be updated. Next the MN experiences some transitions, which are pushed into the new working memory. Finally, figures 23 and 24 indicate how the sampling is carried out. The replay memory has a maximum storage size. A pointer will point to the oldest transition in memory while maintaining a maximum storage size. Once the transition pointer has "left" an old memory segment, that memory is then popped from the memory stack.

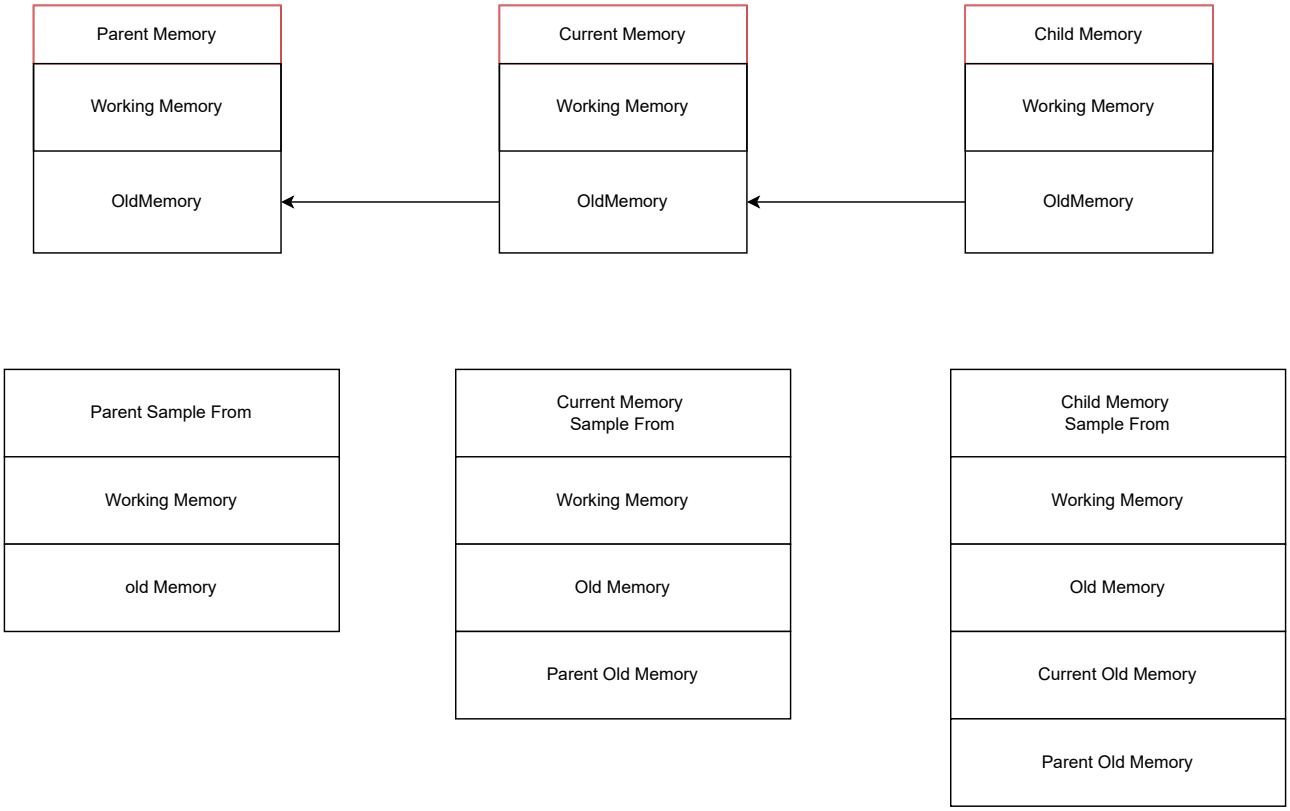

Figure 21: The path between between replay memories

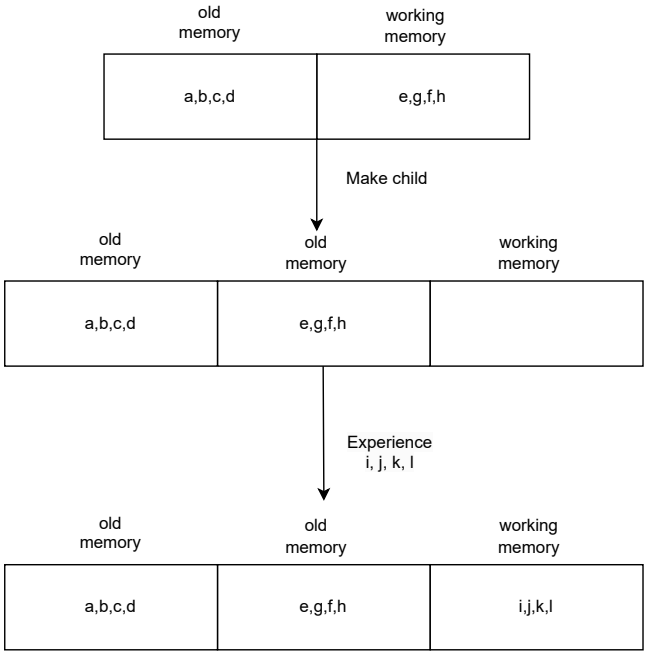

Figure 22: Working memory relocation when making a child

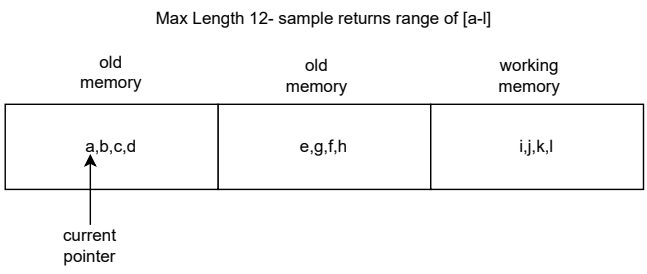

Figure 23: Sampling when under maximum

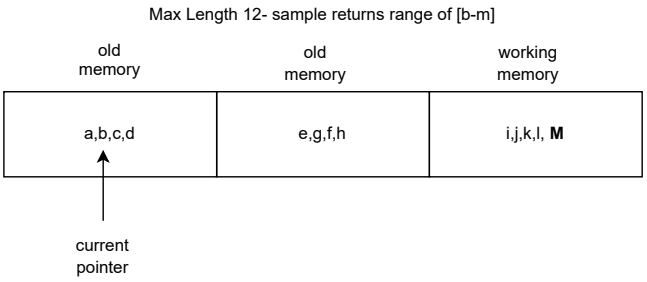

Figure 24: Sampling when over maximum

11.3 Algorithm

The genetic algorithm pseudocode, based on the algorithm introduced in section 3.2.

Algorithm 6 Genetic Algorithm

```
procedure RUN
    initialise population of RL as Population
    for i ← 1 to max_generations do
        run the each RL agent for s training steps
        parents ← SELECTION                          ▷ select the best q parents to be used for selection
        children ← VARIATION(parents)   ▷ perfom mutation and crossover operations to produce offspring
        population ← REPRODUCTION(children)   ▷ Select best offspring to be included in next generation
    Return best RL agent in population
```

Algorithm 6 highlights the overview of the genetic technique applied. First the population of RL agents is initialised which consists of randomly initialising the networks weights and biases. Next the population is to be sorted in terms of how "good" each solution agent is. Hence comes the first key aspect of the genetic algorithm.

11.4 Fitness Function

Potential solutions must be compared based off a metric known as fitness. In this scenario we will use the average test reward after a set input of shared training states in order to gauge the fitness. In previous iterations, the inputs to each model were set to be completely random. However after witnessing very irregular fitness values from every model, it became clear that this was not an accurate manner in which to comparatively test the models. Consider this scenario: if a "good" model (one with a higher average percentage reward per episode), received an input state which was very difficult to deal with- for example a very rare state occurrence, the state that would induce a poorer performance on every model inputted into, including the "better" model. If, in the same scenario, a "bad" model, one with a much lower average reward per episode received a much more common state, the worse model may perform better. In an evolutionary setting, this characteristic is not desired. In order to achieve elitism, the best solutions must always be selected for the next generation.

An alternative approach to standardised inputs across all models could be that of averaging the rewards of k test episodes. Although this approach will partially eliminate the above mentioned input bias, with a larger population size P_{size}, the computational intensity of this task will introduce a large overhead complexity-subtracting from the performance from the program at large. The proposed fitness calculation is a costly one $O(nlogn)$ for pairwise and $O(n)$ for listwise. These are polynomial complexities, but compounded with k, P_{size} and n, it is in the best interest to the program to remove any unnecessary or avoidable complexity.

Algorithm 7 Fitness Function

Require: $S_{test} \leftarrow$ input states for testing
procedure RUN
    Initialise environment with S_{test}
    initialise state s_1 as S_{state_1}
    initialise episode rewards r_{ep} to 0
    while True **do**
        select action a_t from $\arg\max_{a_t \in A} Q(s_t, a_t)$ ▷ Select the best policy network predicted action for state s_t
        $s_{t+1}, r_t, done \leftarrow$ STEP(a_t) ▷ generate reward, next state and episode status after one step
        $r_{ep} = r_{ep} + r_t$
        if $done = True$ **then**
            break
    return $r_{ep}/episodelength$

11.5 Selection

The selection scheme chosen to select models for genetic variation is that of binary tournament selection. An important aspect of genetic algorithms is that of selection pressure, a measure of difficulty for a worse individual is to be selected for the next generation. With a higher selection pressure, the algorithm will exploit the search space and converge quickly to a local optimum. With a lower selection pressure, the algorithm will be exploitative but will converge slowly. Therefore it is important to balance this exploration, exploitation trade off by selecting the correct selection scheme for the problem at hand. In the literature, strategies such as **Fitness Proportion Selection, Linear Ranking, Truncation Selection, $\mu + \lambda$ selection, genitor selection, and tournament selection** are introduced. (He & of Birmingham (n.d.), Goldberg & Deb (1991)). From work by Goldberg & Deb (1991), Oladele & Sadiku (2013) tournament binary tournament selection has a similar performance to the other methods, but with a lesser execution time - $O(n)$.

Algorithm 8 Tournament Selection

Require: number of parents required as c
Require: population as G
procedure TOURNAMENT_SELECTION(self)
    Initialise empty array of selected parents as P
    Initialise selection pool as copy of G as D
    for $i \leftarrow 1$ to c **do**
        choose two individuals randomly without replacement as F from D
        Select the best individual $\in F$ as F_{best}
        Add F_{best} to P
        Add not selected individual back to D
    return P

Note here in the above algorithm, the better individuals are not replaced back into the selection pool. This is to encourage variation in the possible solutions, such that the population does not become a pool of copies the same parent and that exploration of possible solutions is encouraged.

11.6 Mutation

The first kind of variation consists of a non deterministic change of parent such that neighbouring solutions can be explored.

11.6.1 Simple Random Mutation

A naive method to mutate an RL agent is that of randomly mutating weights and biases in the agent's policy network.

11.6.2 Availability Mutation

This mutation directly builds upon previous work by Iglesias et al. (2006).

11.6.2.1 Probability of Selection of State

Firstly, the probability that a particular model is to be mutated must be calculated. The intuition is as follows: a model is more likely to be mutated if the selected action given state s_t is poor compared to the action selected by the current best solution given state s_t. Note that an action is "good" if the predicted state action value of that combination is high. A probability function that composed this requirement is proposed below:

$$\pi(s) = \text{The policy of the current model, returns the action chosen by policy network} \tag{13}$$

$$Q(s,a) = \text{The state action value given state } s \text{ and action } a \text{ calculated by the best individual in the population} \tag{14}$$

$$V1 = cardinality\{a_x | Q(s_t, a_x) \geq Q(s_t, \pi(s_t)\} \tag{15}$$

$$V2 = cardinality\{a_y | Q_(s_t, a_y) \leq Q(s_t, \pi(s_t)\} \tag{16}$$

$$P_{mutation}(\pi(s)) = \frac{V1}{V1 + V2} \tag{17}$$

Therefore the model is more likely to be chosen to be mutated if the action selected by the model has a lower state action value than many of the other actions when computed by best model in the population.

11.6.2.2 Probability Calculation

Now that a model has been chosen to be mutated, there is an uncertainty as to what action the model should choose instead of the action currently returned by $\pi(s)$. An action should be much more likely to be chosen if that action is better then $\pi(s)$. A relationship that captures this is as follows: $\exp \frac{Q(s_t,\pi(s))}{Q(s_t,a)}$, along with normalisation. First Q values are multiplied by -1, and then normalised to be between 1 and 2. This is such that the better Q value has a value of 1, while the worse Q value has a value of 2. The maximum value of this function is $e^{\frac{2}{1}} = 7.389$ and the minimum is $e^{\frac{1}{2}} = 1.649$. Adapted from Iglesias et al. (2006), this distribution is designed such that there is a bias towards selection of actions better then the current solution, while avoiding the risk of exponential increase in probability if there is a large divergence between values. For example, $\exp \frac{1}{0.01} = 2.7 \times 10^{43}$, removing all possibility of selection for any other action.

$$P(s, a_i) = \frac{e^{Q(s_t,\pi(s))/Q(s_t,a_i)}}{\sum_j e^{Q(s_t,\pi(s))/Q(s_t,a_j)}} \tag{18}$$

11.6.2.3 Implementing the change

Unlike the work done by Iglesias et al. (2006), there is no state action value table that can be easily updated. Instead, the state action values are predicted using the policy network. Therefore, to change the action selected by the model, it must be done through training the model.

Algorithm 9 Mutating the action selected by the model

Require: model to be mutated : M

procedure MUTATE
  T ← sample x transitions from memory
  for $t \in T$ **do**
    if t is selected using equation 17 **then**
      $a_{improve}$ ← select action using equation 18
      $av_{current}$ ← Best state action value computed by M across all input actions given state S
      $a_{current}$ ← Action corresponding to $av_{current}$
      $av_{imporve}$ ← State action value computed my M given S and $a_{improve}$
      while $av_{current} > av_{improve}$ **AND** number of loops $< z$ **do**
        train the agent using 100 batches E
          where $E = \{\{state = t_{state}, action = a_{improve}, reward = 1, nextstate = t_{next\_state}\},$
            $\{state = t_{state}, action = a_{current}, reward = -1, nextstate = t_{nextstate}\}\}$

The above algorithm explains that the process of mutating which action should be selected involves continually training the agent, with transitions providing reward to actions we want to promote and to give a penalty to the action we no longer want to be selected.

11.7 Crossover

Crossover is reproduction breeding in a real world scenario. Here, the traits of two successful parents are combined to produce offspring with a higher fitness value. In the algorithm proposed, an individual in the population is an RL agent, containing two neural networks. In this work, crossover is depicted as follows:

11.7.1 Naive Crossover

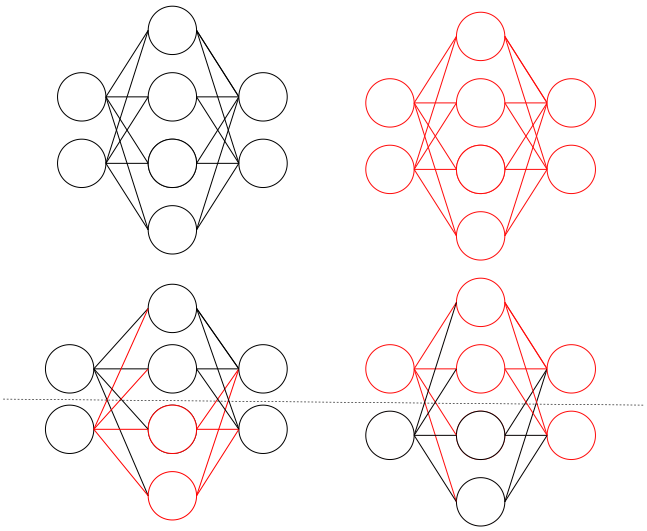

Figure 25: Neural Network Crossover

In this work, the neural networks will have a hidden layer of length 128. The lines dictate the network weights, while the circles dictate the nodes, and importantly contain the biases for each network node. In order to crossover between two networks, a random cutting point is selected along these nodes. Then, the nodes along with their input weights and the output weights are swapped over as shown in the diagram. In previous work however this is shown to be very destructive to network performance. Multiple previous papers discuss the concept of the competing conventions problem, in which there exists a many to many relationship between phenotype and genotype pairs of neural networks. To summarise, neural networks with very different weight architectures can produce the same output performance. And so, although networks may output something similar, crossing over the nodes means potentially crossing over two very different and incomparable networks. Without treating the networks, this crossover process has been coined **Naive Crossover** in this paper.

11.7.2 Similarity Crossover

Genetic crossover is a key component in every genetic algorithm as it is a powerful method used to search the solution space. Due to above mentioned problem, crossover in neuroevolution is historically omitted. Without this genetic operator however, the algorithm loses a very powerful method in which to traverse the solution space, and such the speed of convergence, or perhaps ability to converge to any meaningful solution at all is put at risk. Dragoni et al. (2010) and Uriot & Izzo (2020) introduce a method designed to reintroduce crossover neuroevolution.

	Network a Output	**Network b Output**
node 1	0.32, 0.84	0.6, 0.12
node 2	0.64, 0.14	0.08, 0.91
node 3	0.93, 0.01	0.31, 0.79
node 4	0.11, 0.94	0.90, 0.04

Table 6: Activation values for each node of hidden layer, given the same 2 observations input

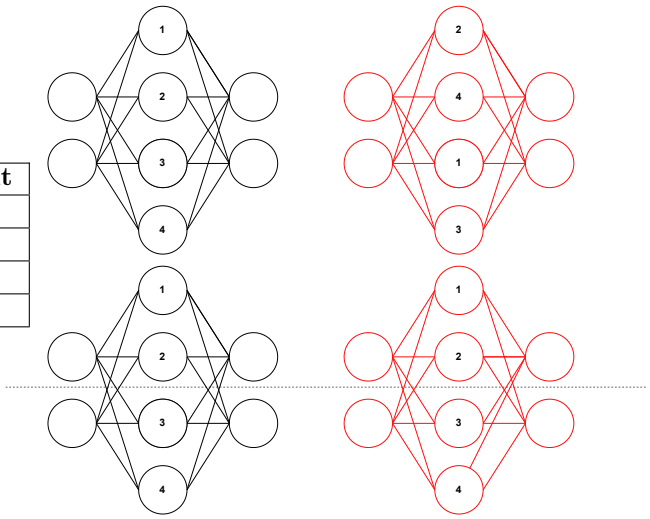

Figure 26: Similarity Based Crossover

Before performing neural network crossover, Dragoni et al. (2010) introduces a manner in which to compare nodes for similarity in performance. The metric for comparison is as follows- using a standard set of input data across all nodes, the activation values of the nodes in network are collected, given a set of observation. Table 6 depicts the activation values of each node in the hidden layers of each network. With this exaggerated example,

it is clear to see that each node in the network can be matched one to one with another node in the other network in terms of activation values. Figure 26 depicts this correlation by labelling nodes that match with the same number. Once the nodes in the network b are permuted and functionally aligned with network a, a crossover as shown in figure 25 can now be performed. With this alignment in functionality, the destruction of child fitness is removed as the network architecture between both networks has a one to one correspondence.

With this trivial example it is clear to see which nodes perform similarly. However, in a more complex example, a correlation metric must be established to achieve a similar result.

In order to achieve this Dragoni et al. (2010) details a method of pairwise cross correlation which will be adapted for this problem into this paper. Pairwise cross correlation is used to determine the relationship between two datasets. The equation below depicts how a matrix PCC is computed, with $element_{i,j}$ containing the correlation between activations ha_i and hb_j.

$$ha = [actiavtions_1, actiavtions_2, ..., actiavtions_n].T \text{ of the hidden layer of network a} \tag{19}$$

$$hb = [actiavtions_1, actiavtions_2, ..., actiavtions_n].T \text{ of the hidden layer of network b} \tag{20}$$

$$activations = \text{distribution of activation values given a standard set of inputs} \tag{21}$$

$$PCC_{i,j} = \frac{(ha_i).T * hb_j}{\sqrt{Var(ha_i) \times Var(hb_j)}} \tag{22}$$

Where n is the number of input observations to the neural network.

To continue with the explanatory example:
Finally, PCC can be interpreted as an adjacency matrix of a graph where the edge weights dictate the strength of correlation between nodes. Using this information, we can apply Maximum Bipartite Matching, to maximise the total correlation between these nodes and hence produce a pairing in which each node is maximally correlated.

	hb1	hb2	hb3	hb4
ha1	4.69230769	7.32159407	12.22435897	2.8765653
ha2	6.68	1.72144578	5.15	5.41023256
ha3	5.06521739	0.43740178	2.68297101	4.23356926
ha4	1.79518072	5.01785455	7.79819277	0.76548053

Table 7: PCC matrix using the working example

$$\text{Pairing produced: } ('ha1','hb3'), ('ha2','hb1'), ('ha3','hb4'), ('ha4','hb2') \tag{23}$$

Table 7 and the above statement show the pairing produced, matching the trivial conclusion.

11.8 Arithmetic Crossover

Another crossover method attempted known as arithmetic crossover, previously shown in Dragoni et al. (2010). Instead of "cutting" the nodes at a certain point and swapping the nodes, instead a weighted average of the neural networks is applied. By functionally aligning the networks initially, this crossover method also removes the competing conventions problem previously discussed. The crossover is done as follows:

$$\theta_a = (1-t)\theta_a + t\theta_b \tag{24}$$

$$\theta_b = (1-t)\theta_b + t\theta_a \tag{25}$$

Where t is randomly generated such that $t \in [-0.25; 1.25]$

12 Results

12.1 Reinforcement Learning

12.1.1 Testing procedure

Note here that NRPA is a score used to predict how close the predicted rank of L_{out} is to that of L_{opt}. Because of this, it can only be used to compare features which predict the rank while not applicable to pairwise verdict. However in order to compare correctness verdict and rank based methods, an adapted NRPA can be used. For pairwise verdict, L_{opt} is set to be a sorted input list based on first verdict and then duration. When calculating NRPA from this, the ability to predict the features in a desired order can be directly compared, be it predicting rank or verdict.

12.2 Results Comparison and Analysis

The models performances will now be documented. Upon running the listwise environment, due to the large number of input features the performance of the model is poor. The training time for improvement, the NRPA, APFD and average reward return in comparison between all models and comparisons is poor.

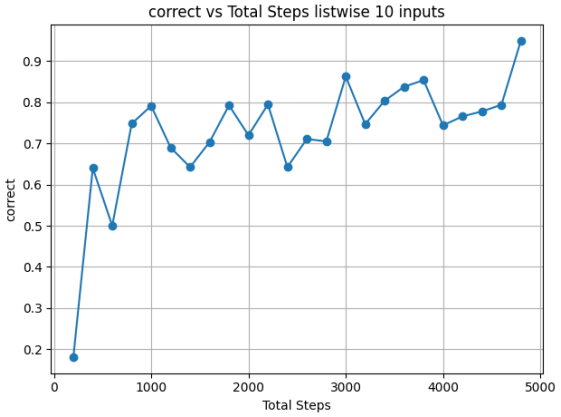

Figure 27

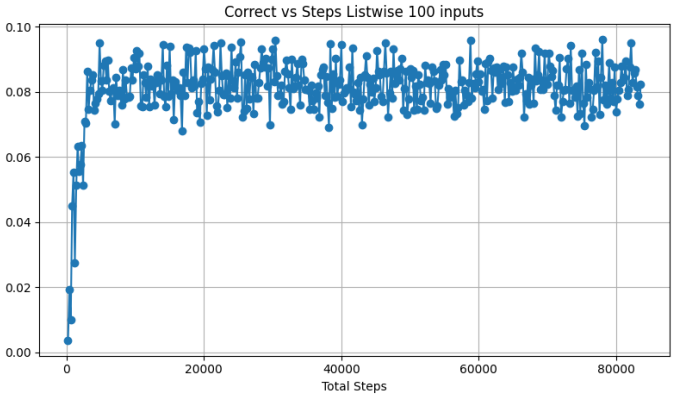

Figure 28

From this figure 27, listwise performs well on an input size of 10 - achieving results consisting of a high episode reward average of 0.95 after only 500 steps or 40 seconds. The model is equally matched in performance by NRPA and APFD, however, with a trivially low input size such as this, there is no benefit to this performance.

Figure 28 then shows the average performance when the listwise environment is used for an input size of 100:

Once more the difficulty remains that the environment performs poorly in relation to all metrics. And when the model is applied to larger learning spaces, for example inputs, the problem is compounded - achieving an average reward of 0.091 in 83600 steps.

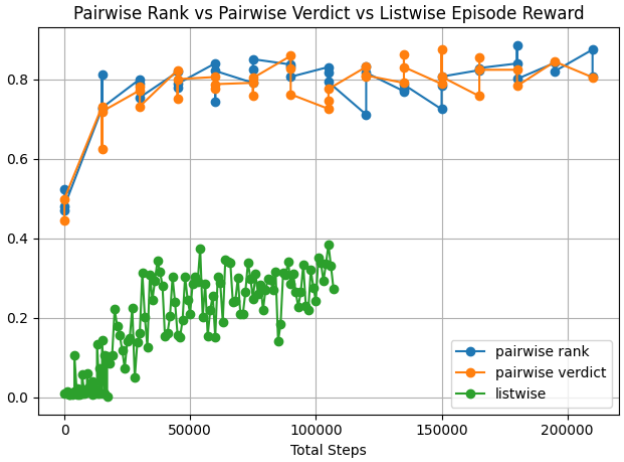

Figure 29

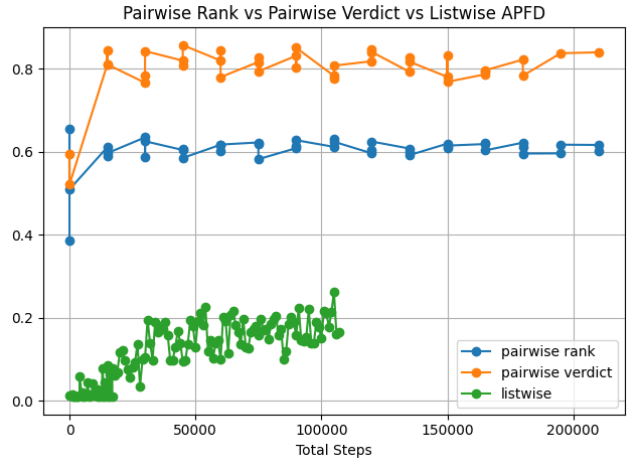

Figure 30

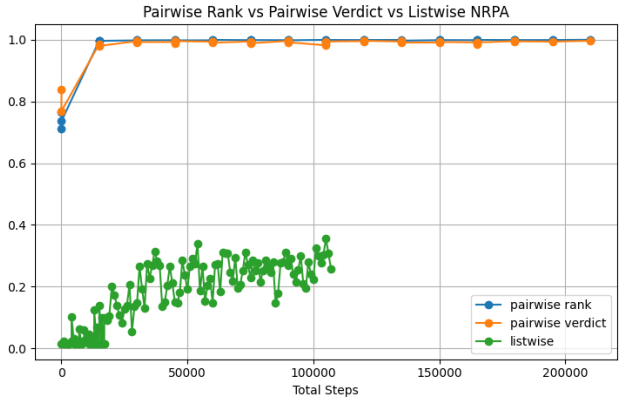

Figure 31

The table below directly compares all the environments with an input size of 100, with performance visualised in the graphs above.

Environment	Average Episiode Reward	NRPA	APFD	Total Environment Steps
Listwise	**0.18802480000000002**	**0.09834726201722319**	**0.0.17622597810133955**	**35310.0**
Pairwise Rank	**0.781756976744186**	**0.98006714494251**	**0.6027431144797216**	**210000**
Pairwise Verdict	**0.7732197499999999**	**0.982015516031241**	**0.7986743296107621**	**210000**

Table 8: Models selected for test case prioritisation

The difference between the models is a stark one. Firstly there is a gulf in performance accuracy between listwise and pairwise for the use case of 100 inputs. Taking the average episode reward each test as a metric for the speed of learning of the model, the listwise agent begins with a random initialisation, and a correctness of 0.12, and progresses only by 0.08 points. The listwise agent has the worst metric scores in all categories. As of the original RL algorithm, a listwise approach is not recommended, however, the genetic improvement may boost the performance and will be analysed in section 12.3.

Below is the performance of the both pairwise approaches for an input dataset of 1000.

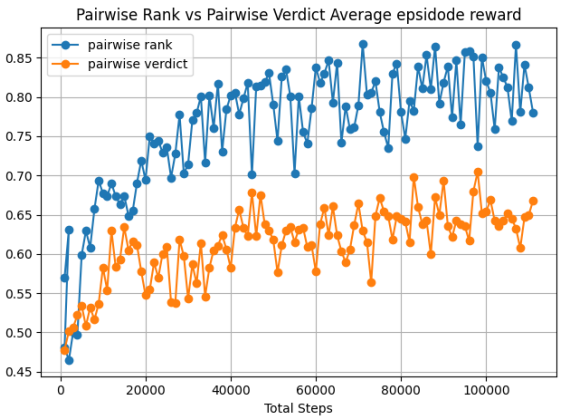

Figure 32

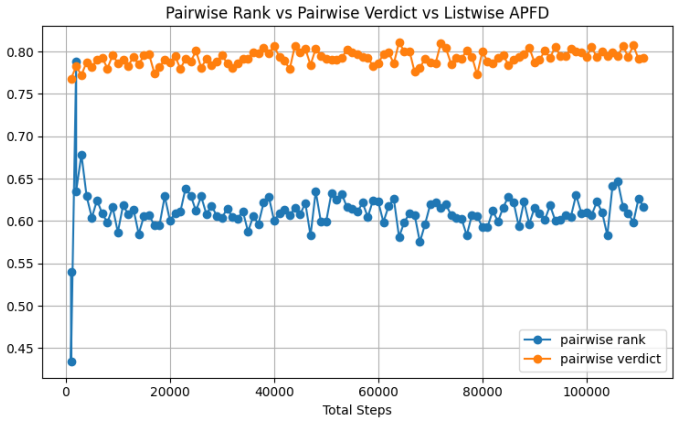

Figure 33

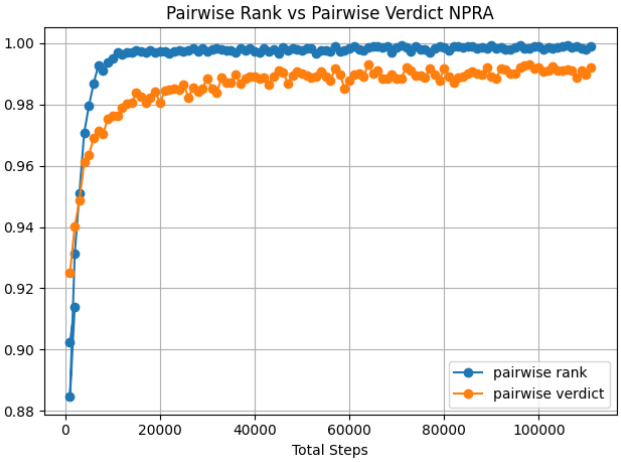

Figure 34

Over 125000 environment steps, pairwise rank performs considerably better in all metrics that the models can be directly compared in. The average reward can be a metric for how fast a the agent can train in comparison to its opponent, 7.723 average reward with a 0.780 with a final value. In comparison, The pairwise verdicts reward is converges at an average of 0.614 and a final value of 0.618. It is unsurprising that the verdict agent received a higher score for APFD (0.627, compared to 0.7910) due to the failure focused learning characteristic. However, this is a an impressive result, Bagherzadeh et al. (2021) achieved an average of .53±.13 and 0.67±.2 across two listwise DQN datasets. Be that as is may, both models score highly in the NRPA metric, converging at a final value of 0.989858599484348, 0.9978615247028421 for rank and verdict respectively. The boost in performance in the pairwise rank runner is likely attributed to the greater mutual information scores the input features had with the target rank, and would be interesting topic for future work to continue feature engineering with the intent to boost the Pairwise verdict reward score.

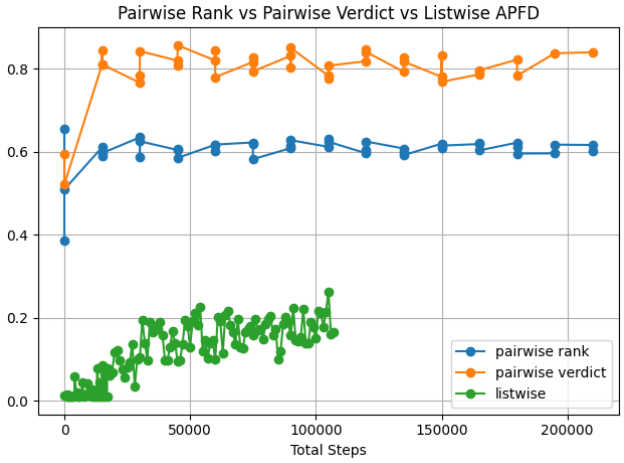

Figure 35: Listwise environment with an input size of 10

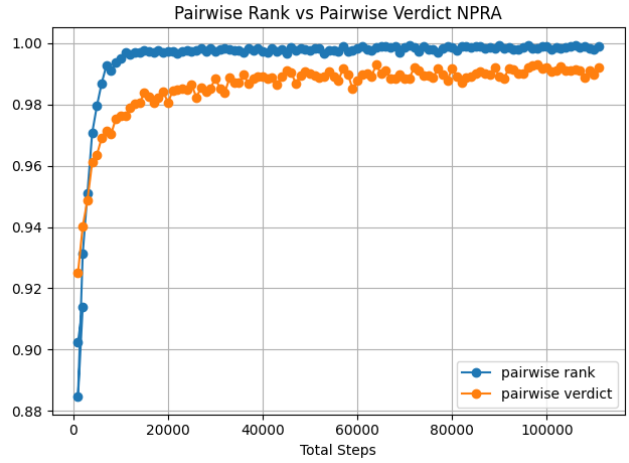

Figure 36: Listwise environment performance with an input size of 100

12.3 Evolutionary Improvement

12.3.1 Testing procedure

The variation methods discussed in this section will be, “cut crossover”, “arithmetic crossover”, and “availability mutation”. These methods will be tested individually for each environment.

Each test will consist of 100 generations, where the running fitness, APFD, and NRPA of every population individual are stored.

12.3.2 Graphs

A variation operator is considered effective if the children produced are viable for the next generation, and whether or not this is the case is analysed below.

If this method was included in the evolutionary algorithm, the algorithm will perform similarly to multiple RL agents being run simultaneously, as none of the new children will be accepted into the next generation, negating the population based improvement design of the genetic algorithm.

As discussed in section 11, the problem of the competing conventions needed to be addressed. In the referenced section, naive crossover is discussed, and will be analysed here for comparison.

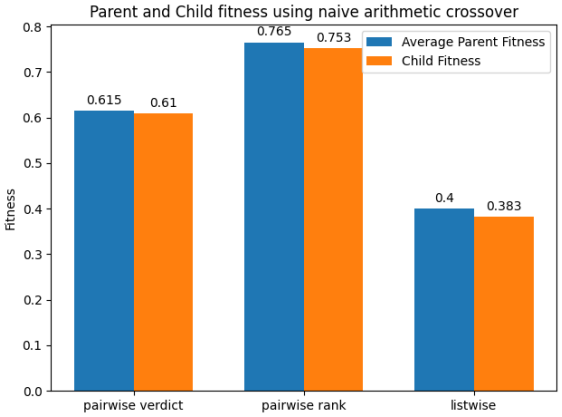

Figure 37

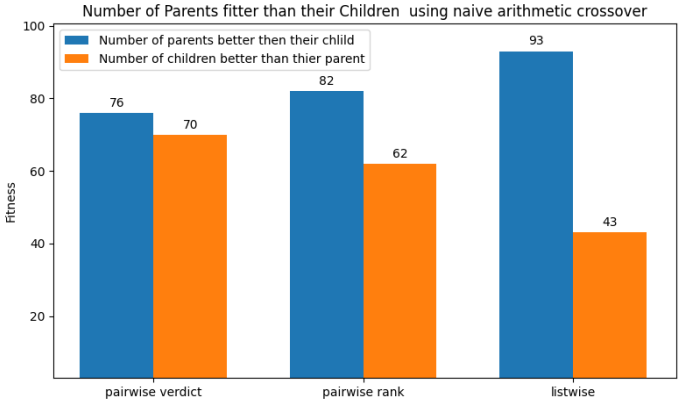

Figure 38

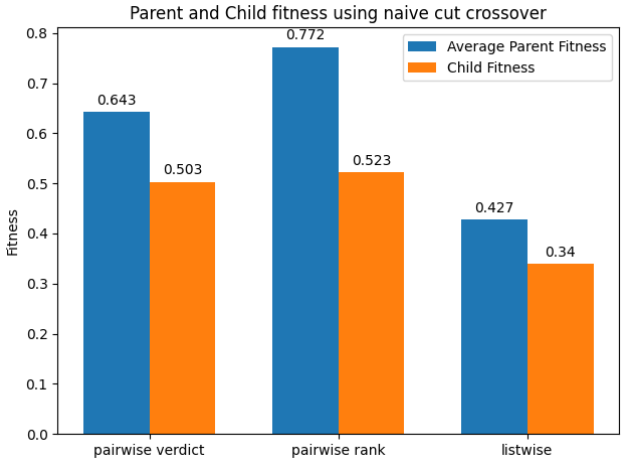

Figure 39

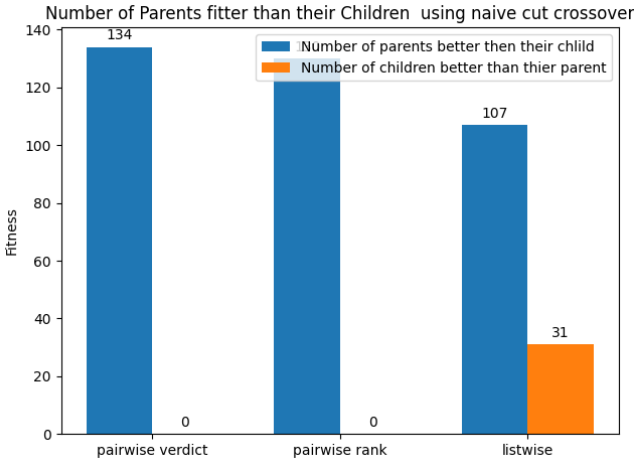

Figure 40

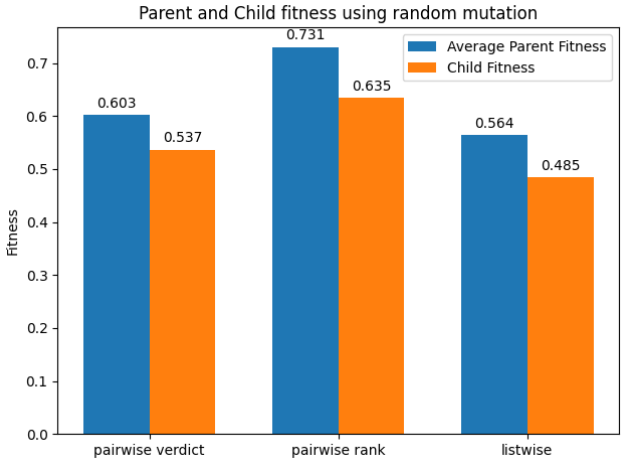

Figure 41

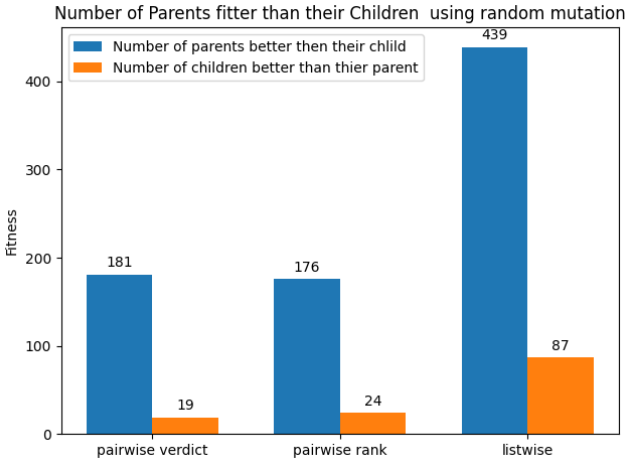

Figure 42

As expected, the average child fitness for all naive methods is worse than the parents, not providing a significant amount of viable children to the population. Especially that of the naive cut crossover, achieving 100 generations of children entirely worse than their parents. However, although the fitness is lower in all

environments, the average children number is much closer to that of the parents.

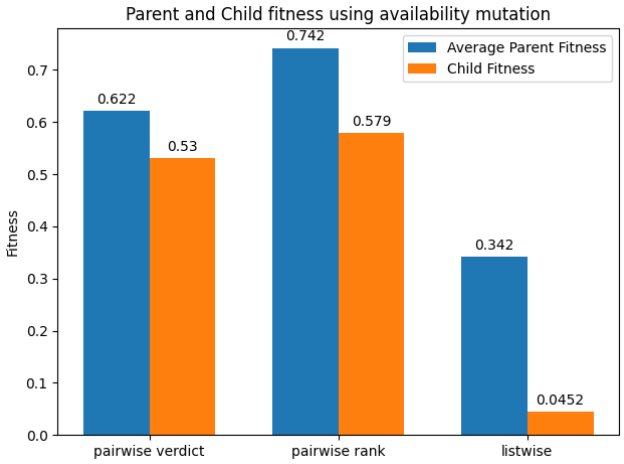

Figure 43

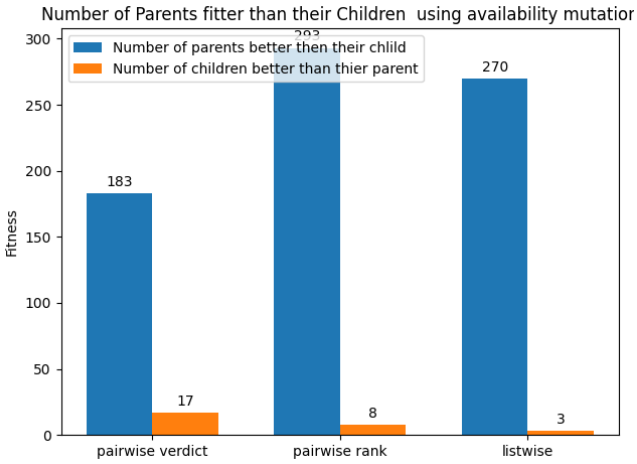

Figure 44

The next model here is the *Availability Mutation*. The children produced are on average much worse than their parents and are rarely better, as shown in figures 43 and 44. This is speculated to do with the unstable "best" individual in the population. Due to training time being minimal at the start of the learning process, the "best" individual is put there based on pure stochasticity, forming a flawed architecture target that the other agents are aiming towards. However, the variability of the availability mutation is to be noted. For example, for there to be 3 better children with an average of just under a 0.2 fitness units, the children that are improving on the solution must have a very large fitness value. This variation suggests that the operator can achieve these performances more often with the correct tuning. For future work, an approach with differing probability functions or varying hyperparameters can be trialled to attempt to offset this problem to yield stronger results.

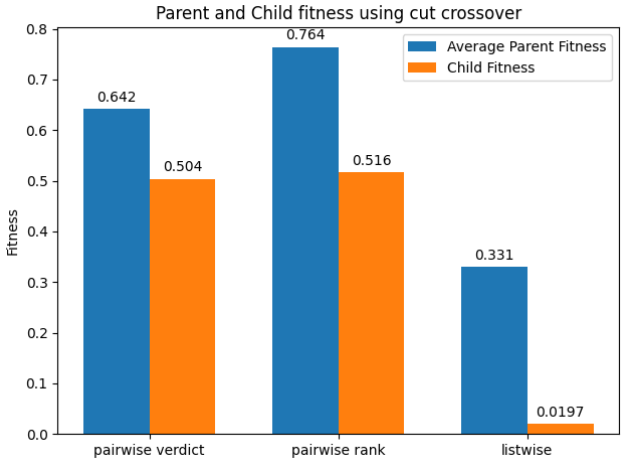

Figure 45

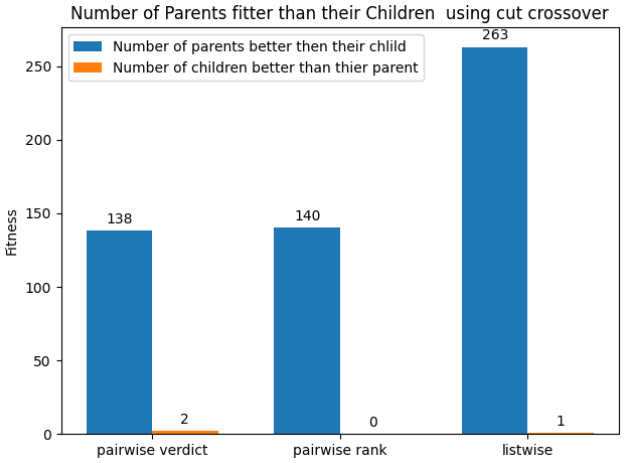

Figure 46

Figures 45 and 46 display the results from the use of the cut crossover. It is interesting to note that the cut crossover performs much worse than its arithmetical counterpart as discussed below. As above, the produced children are much weaker individuals than their parents. Arithmetic crossover performing better at this stage indicates that the operator is less destructive. The PCC correlation is likely going to correlate some nodes that are less similar to each other as a result of the stochastic nature of the training process, and competing conventions process and the operator can be concluded to be much less compliant with error.

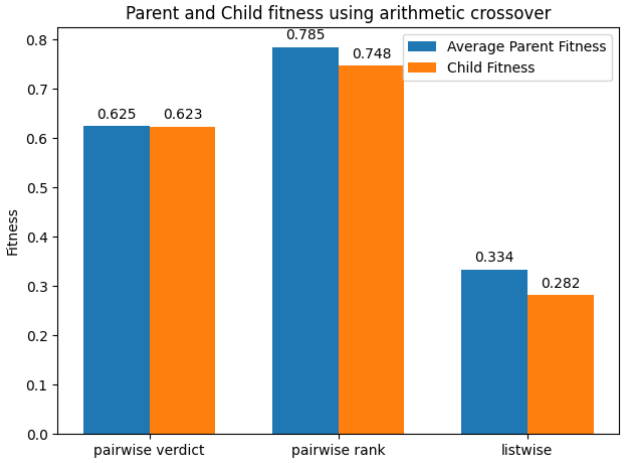

Figure 47

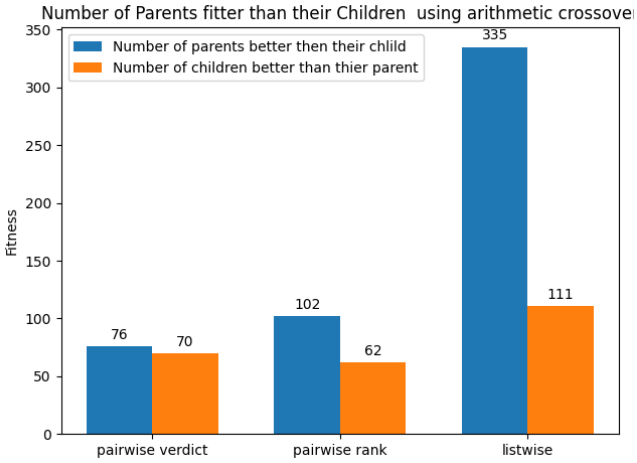

Figure 48

Other results however, for example the similar performance of the naive cut to the cut operator, show that the current attempts at improving the crossover using PCC were unsuccessful. Dragoni et al. (2010) discusses another form of correlation metric, CCA, which performs well with noisy data, opening another avenue for future work. PCC does not report such an ability.

Nonetheless, the variation operators did produce a number of fitter chidren, and thus the performance of the algorithms are shown below.

The models that produced viable children are then analysed in terms of their performance in the prioritisation and ranking context.

The tables below contain the evaluation of each of the evolutionary models for each of the RL environments. Each table contains the original RL agent performance as shown by the metrics, as well as that environment when a genetic algorithm is applied. The values in the table are collected from when each agent terminates, be it from reaching a maximum number of steps or once the agent reaches a plateau, when the fittest individual does not receive an improvement on its fitness values for 10 generations. For the genetic agents, the minimum execution time is 100 generations.

Listwise

	NRPA	APFD	Plateau Time
Original	**0.7696418171228887**	**0.40349999999999997**	**1563.4554607868195**
Arithmetic Crossover	**0.7372859638905067**	**0.4875531914893617**	**6952.407241821289**
Naive Arithmetic Crossover	**0.7390302853814793**	**0.491**	**9123.587044715881**
Cut Crossover	**0.7492312172393709**	**0.5190476190476191**	**5596.867602109909**
Naive Cut Crossover	**0.7502329644729179**	**0.49913043478260866**	**9294.866579055786**
Random Mutation	**0.7747815958066394**	**0.51225**	**4750.12050485611**

Table 9: Listwise genetic improvement results

Pairwise Rank

	NRPA	APFD	Plateau Time
Original	**0.9978615247028421**	**0.6266095100864553**	**940.7619524002075**
Arithmetic Crossover	**0.9980735940706**	**0.6369119496855346**	**7127.7047238349915**
Naive Arithmetic Crossover	**0.9972104990817564**	**0.606466360856269**	**9550.755471467972**
Cut Crossover	**0.9882493614170261**	**0.6003493975903613**	**10353.793496608734**
Naive Cut Crossover	**0.9961822399157728**	**0.6074674556213018**	**13793.953528642654**
Random Mutation	**0.9901641544084657**	**0.618668168168168**	**5552.218576431274**
Availability Mutation	**0.9967399038087661**	**0.611414373088685**	**7148.6632125377655**

Table 10: Pairwise Verdict genetic improvement results

Pairwise Verdict

	NRPA	APFD	Plateau Time
Original	**0.989858599484348**	**0.7910449101796406**	**1563.4554607868195**
Arithmetic Crossover	**0.9882261690405402**	**0.7846957831325301**	**5240.971687555313**
Naive Arithmetic Crossover	**0.986741836008303**	**0.7803967551622418**	**10381.851953268051**
Cut Crossover	**0.97305168421039**	**0.7947424242424241**	**10688.156397819519**
Naive Cut Crossover	**0.9853732794151955**	**0.7851089552238806**	**14889.111254692078**
Random Mutation	**0.9887370802849844**	**0.7859104046242774**	**5671.776670694351**
Availability Mutation	**0.998021777623574**	**0.7883724637681159**	**7768.417682170868**

Table 11: Pairwise Rank genetic improvement results

The listwise agent can be seen to benefit from the genetic algorithm and witnesses a boost in APFD. NRPA is improved with the application of random mutation of all genetic algorithms.

Meanwhile, the pairwise agent showed some improvement in the NRPA and APFD metric using the arithmetic crossover.

Finally, pairwise verdict observed an improvement in NRPA when using cut crossover and availability mutation. Cut crossover was the only source of improvement in the APFD metric also.

Analysing the data however, there is no clear superior genetic operator, but for both metrics an improvement is made in availability mutation and cut crossover. This is due to the fact that even with low performing variation operators, children of better fitness were being produced which can be seen to boost the agent performance. The plateau time alongside this fact, although indicating the superior complexity of the genetic improvement algorithm, maintains that the agent continues to learn for longer than the base RL agent, and is a topic of discussion for future work.

However despite some improvements, it is noted that the running time of all the genetic improvement agents is 339% , 850%, and 680% longer in seconds compared to the base RL agent. Therefore, in order to retrieve a higher performance, a large cost must be surmounted. When applying test case prioritisation to a set of test suites, the intent is to optimise the time and budget of the testing process. Naturally, if the ranking process consumes a large chunk of the budget, the solution is not a viable one.

13 Future Work and Evaluation

The undertaking of this project was not without its challenges. Firstly, the reinforcement learning paradigm is an untaught concept and had to be learnt with no previous experience other than related knowledge in previous machine learning disciplines. Secondly, the timeline of the project had to be adjusted along the process due to unfamiliarity with the PyTorch library and the access to advanced GPU hardware. The personal laptop used does not have graphics hardware meaning extended agent training. When using university provided GPU machines, due to incompatible drivers with PyTorch it became a challenge to train models effectively until that problem was resolved. When that time came, a silent error caused by a combination of driver dysfunctions and syntactical PyTorch errors meant that the agent was silently not learning, pushing the project timeline further back. In future work, the exploration of hyper parameters, reward space adaptation and an exploration of differing probability functions with the availability mutation can be addressed as discussed in sections 10.5. With stronger hardware available, more individuals can be included in the genetic population. The results dictate that the evolutionary adaptation can provide slight improvements in APFD and NRPA at the cost of a long execution time. Therefore, with greater hardware these negatives can be mitigated.

14 Conclusion

In this paper, we investigated the concept of RL in the case of prioritisation and ranking with a genetic improvement. With software testing being a paramount feature in software development, the drive to reduce the time and resource budget of software regression testing is an active area of research. To effectively select test cases for regression testing, we apply 4 different methods to run. Firstly, the dataset was analysed for mutual information such that the selected features are the ones most correlated to the target values - being the prioritisation group, the rank and the verdict features in the dataset. Classification was then carried out using ML to distribute the dataset into three priority groups. Upon selecting the features with the highest mutual information, the random forest classifier was selected as the highest performing algorithm for classification among other tested algorithms commonly used in test case prioritisation, with an accuracy score of 0.98847. To rank the test cases, a reinforcement learning approach was proposed, specifically using a deep Q network. The unfamiliar paradigm was first researched, and then applied using three well known test environments: listwise, pairwise verdict and pairwise applied for ranking. When applied with the most effective algorithm in terms of accuracy and time, the pairwise ranking had an NRPA score of 0.9989902373076025 in a test size of 1000 test cases. To improve these results, RL was combined with a genetic algorithm in a novel combination of GA and

RL in this context. Variation operators from previous work are adapted to a deep Q learning environment. Although the genetic operators did provide a slight improvement to the agent in terms of NRPA and APFD, it was not without a considerable increase in running time of up to 850%. To conclude, the combination of the random forest classifier and the pairwise ranking approach is the most effective for the use of test case prioritisation and ranking when compared to the other agents' trend. However when applying the genetic improvement, the potential for improvement can be unlocked with greater processing power. As it stands, the genetic operator cannot be recommended in an industrial environment.

References

adamW documentation (n.d.).
URL: *'https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html'*

Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M. I. & Zanoon, N. I. (2013), 'Review on sorting algorithms: A comparative study', *International Journal of Computer Science and Security (IJCSS)* **7**(3), 120–125.

Amir, N., Abdallah, M. M. & Mohammad (2017), 'An overview of regression testing', *Journal of Telecommunication, Electronic and Computer Engineering* **9**, 45–49.

Bagherzadeh, M., Kahani, N. & Briand, L. (2021), 'Reinforcement learning for test case prioritization'.

Bertolino, A., Guerriero, A., Miranda, B., Pietrantuono, R. & Russo, S. (2020*a*), 'Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration'.

Bertolino, A., Guerriero, M., Miranda, H., Pietrantuono, R. & Russo, S. (2020*b*), 'Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration'.

Cao, T., Vu, T., Le, H. & Nguyen, V. (2022), 'Ensemble approaches for test case prioritization in ui testing', *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering* .
URL: *http://dx.doi.org/10.18293/SEKE2022-148*

Choudhary, A. (n.d.).
URL: *https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/*

Doshi, K. (n.d.).
URL: *https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b*

Dragoni, M., Azzini, A. & Tettamanzi, A. G. B. (2010), 'A novel similarity-based crossover for artificial neural network evolution', *LNCS* **6238**.

Face, H. (n.d.).
URL: *https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-algorithm*

Fox, R., Pakman, A. & Tishby, N. (2016), 'Taming the noise in reinforcement learning via soft updates'.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G. & Pineau, J. (2018), 'An introduction to deep reinforcement learning', *Foundations and Trends® in Machine Learning* **11**(3–4), 219–354.
URL: *http://dx.doi.org/10.1561/2200000071*

Glover, F. & Laguna, M. (1997), 'Tabu search'.

Goldberg, D. E. & Deb, K. (1991), 'A comparative analysis of selection schemes used in genetic algorithms', *Department of General Engineering, University of Illinois at Urbana-Champaign* .

Haflidason, S. & Neville, R. (2009), On the significance of the permutation problem in neuroevolution, *in* 'Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation', GECCO '09, Association for Computing Machinery, New York, NY, USA, p. 787–794.
URL: *https://doi.org/10.1145/1569901.1570010*

He, S. & of Birmingham, U. (n.d.).

Holland, J. H. (1975), 'Adaption in natural and artificial systems'.

Iglesias, R., Rodríguez, M., Regueiro, C. V., Correa, J. & Barro, S. (2006), 'Combining reinforcement learning and genetic algorithms to learn behaviours in mobile robotics', *Proceedings of the Third International Conference on Informatics in Control, Automation and Robotics* pp. 188–195.

Kajo Meçe, E., Binjaku, K. & Paci, H. (2020), 'The application of machine learning in test case prioritization - a review', *European Journal of Electrical Engineering and Computer Science* **4**.

Lan, Q., Pan, Y., Fyshe, A. & White, M. (2020), Maxmin q-learning: Controlling the estimation bias of q-learning, *in* 'International Conference on Learning Representations'.

Lizard, D. (n.d.).
URL: *https://deeplizard.com/learn/video/rP4oEpQbDm4*

Mäkinen, S. & Münch, J. (2014), Effects of test-driven development: A comparative analysis of empirical studies, *in* 'Lecture Notes in Business Information Processing'.

Marijan, D. (2022), 'Comparative study of machine learning test case prioritization for continuous integration testing', *Preprint submitted to arXiv* **2204.10899**. arXiv:2204.10899v1 [cs.SE] 22 Apr 2022.

Mishra, A. D. & Garg, D. (2008), 'Selection of best sorting algorithm', *International Journal of Intelligent Information Processing* **2**(2), 363–368.

Mohd Radzi, S. F., Hassan, M. S. & Mohd Radzi, M. A. H. (2022), 'Comparison of classification algorithms for predicting autistic spectrum disorder using weka modeler', *BMC Medical Informatics and Decision Making* **22**(1), 306.
URL: *https://doi.org/10.1186/s12911-022-02050-x*

not specified, A. (not specified), 'Random forest vs. gradient boosted trees: Pros and cons', Medium.
URL: *https://medium.com/@wilbossoftwarejourney/random-forest-vs-gradient-boosted-trees-pros-and-cons-8c1feec0ea0d*

Oladele, R. O. & Sadiku, J. S. (2013), 'Genetic algorithm performance with different selection methods in solving multi-objective network design problem', *International Journal of Computer Applications* **70**(12), 5.

Oladipupo, E. T., Abikoye, O. C., Akande, N. O., Kayode, A. A. & Adeniyi, J. K. (2020), 'Comparative study of two divide and conquer sorting algorithms: Quicksort and mergesort', *Procedia Computer Science* **171**, 2532–2540.

Onoma, A. K., Tsai, W.-T., Poonawala, M. H. & Suganuma, H. (1998), 'Regression testing in an industrial environment'.

pai (n.d.).
URL: *https://bham-my.sharepoint.com/personal/tla097_student_bham_ac_uk/_layouts/15/guestaccess.aspx?share=Eduq63oTpRAjpA9K57a1JMBXDOTdPpfqws1UITAnORp0ge=YysMmy*

Paszke, A. & Towers, M. (n.d.).
URL: *https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/9da0471a9eeb2351a488cd4b44xoghqFBEpF38*

Pretorius, K. & Pillay, N. (2024), 'Neural network crossover in genetic algorithms using genetic programming', *Genetic Programming and Evolvable Machines* .
URL: *https://doi.org/10.1007/s10710-024-09481-7*

redhat (2022).
URL: *https://www.redhat.com/en/topics/devops/what-cicd-pipeline*

Rothermel, G., Untch, R., Chu, C. & Harrold, M. (1999), 'Test case prioritization: an empirical study', pp. 179–188.

Rothermel, G., Untch, R. H., Chu, C. & Harrold, M. J. (2002), 'Test case prioritization: A family of empirical studies', *IEEE Transactions on Software Engineering* **28**(2), 159–182.

Sabah, A. S., Abu-Naser, S. S., Helles, Y. E., Abdallatif, R. F., Samra, F. Y. A., Taha, A. H. A., Massa, N. M. & Hamouda, A. A. (2023), 'Comparative analysis of the performance of popular sorting algorithms on datasets of different sizes and characteristics', *International Journal of Academic Engineering Research (IJAER)* **7**, 76–84.
URL: *www.ijeais.org/ijaer*

Sharma, S. & Chande, S. V. (2023), 'Optimizing test case prioritization using machine learning algorithms', *Journal of Autonomous Intelligence* **6**(2).

Srivastava, D. P. (2008*a*), 'Test case prioritization', *Journal of Theoretical and Applied Information Technology* **4**, 178–181.

Srivastava, P. R. (2005-2008*b*), 'Test case prioritization', *Journal of Theoretical and Applied Information Technology* **178-181**.

Stenvatten, D. (2020), A comparative study for classification algorithms on imbalanced datasets: An investigation into the performance of RF, GBDT and MLP, PhD thesis, University of Applied Sciences.
URL: *https://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-18660*

Tosun, A., Turhan, B., Ahmed, M. & Juristo, N. (2018), 'On the effectiveness of unit tests in test-driven development'.

Uriot, T. & Izzo, D. (2020), 'Safe crossover of neural networks through neuron alignment'.

Vadapalli, P. (2022).
URL: *"https://www.upgrad.com/blog/naive-bayes-classifier/"*

Wang, Ziyu, Bapst, Heess, V., Nicolas, Mnih, Volodymyr, Munos, Remi, Kavukcuoglu, Koray, de Freitas & Nando (2016), 'Sample efficient actor-critic with experience replay', *arXiv preprint arXiv:1611.01224* .

Whittle, D. (n.d.).
URL: *https://www.apmdigest.com/nearly-90-percent-surveyed-stop-using-apps-due-to-poor-performance*

Xhemali, D., Hinde, C. J. & Stone, R. G. (2009), 'Naïve bayes vs. decision trees vs. neural networks in the classification of training web pages', *IJCSI International Journal of Computer Science Issues* **4**(1).

Yoo, S. & Harman, M. (2012), 'Regression testing minimization, selection and prioritization: a survey', *Softw. Test. Verif. Reliab.* **22**(2), 67–120.
URL: *https://doi.org/10.1002/stv.430*

15 Appendix

Git repository: https://git.cs.bham.ac.uk/projects-2023-24/tla097
Instructions on how to run the code are in the read.me file.