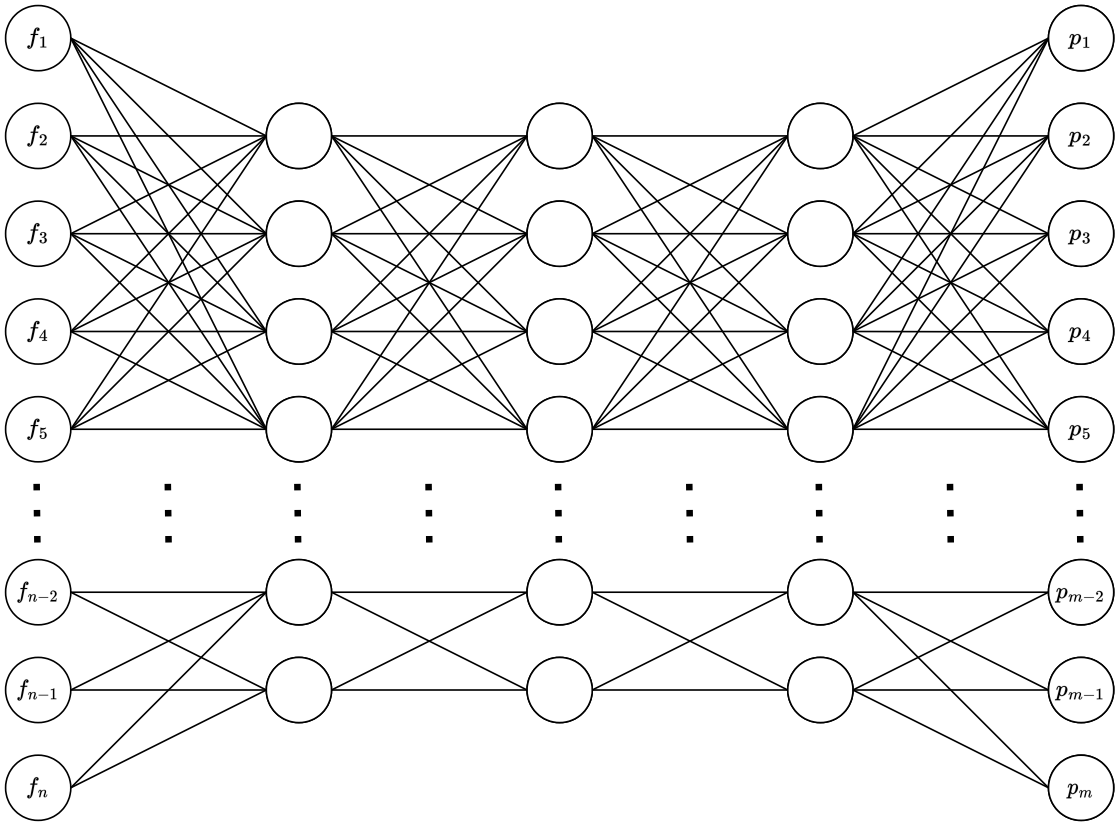
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Optimization of Test Execution

Using Test Case Prioritization and Machine Learning

Master's thesis in Computer science and engineering

Erik Brink
William Risne

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Optimization of Test Execution

Using Test Case Prioritization and Machine Learning

Erik Brink
William Risne

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Optimization of Test Execution
Using Test Case Prioritization and Machine Learning
ERIK BRINK WILLIAM RISNE

Supervisor: Jeremy Pope, Department of Computer Science and Engineering
Advisor: Sebastian Johansson, Aptiv Contracting Services Sweden AB
Co-advisor: Jacob Landelius, Aptiv Contracting Services Sweden AB
Examiner: Thierry Coquand, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An overview of the Prioritized Order Model (POM) structure.

Typeset in LaTeX
Gothenburg, Sweden 2023

Optimization of Test Execution
Using Test Case Prioritization and Machine Learning
Erik Brink
William Risne
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Testing software is among the most fundamental practices of programmers. Though sometimes daunting to carry out, testing still fills an important role of assuring correct software in various forms. The possibly daunting part of testing is the time it takes to execute an entire test suite potentially containing millions of test cases. Such test suites might end up taking days to run, which might leave developers with idle hands.

Various solutions has been proposed to solve the problem of optimizing test suite execution in terms of time efficiency. The time from the start of the execution until receiving an error can be minimized by using *test case prioritization*. This could involve ordering test cases in a test suite, such that the test cases with higher probability to fail (to produce an error) based on modification to a piece of software, are prioritized in the order of execution.

In this thesis, we implement test case prioritization using a Deep Neural Network that produces an order of test cases to be executed. We refer to this model as Prioritized Order Model (POM). We also use *test case selection*, which involves taking a subset of a test suite based on some criteria. In the case of this thesis, the criteria is based on time limitations of the execution of tests. This is done by using an approach that utilizes the Knapsack Problem.

We found that POM performs well given a sufficient amount of data on test suite error reports and modified files in a software repository. POM is compared to different orderings and their time efficiency, which indicated superior performance by POM.

Keywords: Test case prioritization, test case selection, deep neural network, data augmentation, the knapsack problem.

Acknowledgements

First and foremost, we would like to thank our supervisor Jeremy Pope for excellent guidance, lovely chats and insisting on having meetings in the afternoon. Secondly, we would like to thank to our examiner, Thierry Coquand, for taking on this project and providing helpful feedback. Thirdly, we would like to thank the team at Aptiv. In particular we would like to thank Hampus Carlsson for helping us with technical hassle and guiding us through the daunting jungle of version control. Finally, we would like to extend a thank you to our company advisors, Jacob Landelius and Sebastian Johansson for the insightful discussions and the opportunity to work with you on this project.

Erik Brink & William Risne, Gothenburg, 2023-06-14

Contents

List of Figures

List of Tables

1

Introduction

This chapter introduces the problem statement for this thesis project, along with some background, previous work and limitations.

1.1 Background

Testing software is among the most fundamental practices of programmers. The tests themselves are usually collected in test suites that belong to a certain category of tests, like unit and integration tests. However, those test suites might contain millions of tests and might therefore end up taking days to run. As such, developers might experience long waiting times between running the test suite and potentially receiving error feedback. To counter this, *test case prioritization* can be used in order to reduce this waiting time. By prioritizing tests that have a higher probability to fail, errors might be revealed earlier in a test execution.

Different implementations of this problem have been done in previous research. For example, Rothermel et al. [18] used different techniques which include ordering test cases based on executed statements and code branching. The problem has also been attempted with various machine learning methods [5, 21, 20].

From test case prioritization, a subproblem called *test case selection* can be derived [18], in which a subset of the prioritized test suite is selected. Since the time it takes to execute the entire test suite will almost always be the same for each execution, one might only want to execute the "most relevant test cases". This has the effect of constraining the time for the test execution.

1.2 Problem Statement

The purpose of the project can be formulated in the following problem statement:

> Does a machine learning model, which is trained on a collection of test cases and commit history, that produces an ordered test suite have any significant performance improvements based on the following:
>
> (i) That test case failures should occur as early as possible.
>
> (ii) The ordered list of tests should not exceed a given time limit.

This problem statement is addressed through the implementations mentioned in Chapter 3, where the theory for those implementations are explained in Chapter 2.

1.3 Previous Work

The following subsections lists previous research that are of relevance, and inspired the solution used in this thesis project.

1.3.1 The Test Case Prioritization Problem, TCPP

The formal problem definition of test case prioritization is introduced in Rothermel et al. [18] as:

Definition 1. We are given the following: a test suite denoted $\mathcal{T}$, the set of permutations of $\mathcal{T}$ denoted $\mathcal{PT}$, and a function $f : \mathcal{PT} \to \mathbb{R}$. We state the problem as the following: Find $\mathcal{T}' \in \mathcal{PT}$ such that for all $\mathcal{T}'' \in \mathcal{PT}$ we have that if $\mathcal{T}'' \neq \mathcal{T}'$ then $f(\mathcal{T}') \geq f(\mathcal{T}'')$.

In the definition above, f is a scoring function that yields an "award value" for an ordering $\mathcal{T}$. Notice that the definition defines $\mathcal{T}$ as a *test suite* and not an individual test case. Definition 1 is referred to as *The Test Case Prioritization Problem* (TCPP).

Along with the definition of the problem, the authors also introduced nine different so called *prioritization techniques*. These techniques are used when ordering the different test cases. The first three referred to as M_1, M_2 and M_3 are experimental controls which could not be used in practice. The authors highlight M_3, "Optimal prioritization". M_3 is based on the rate of fault detection and is employing a greedy algorithm that always selects test cases that cover the most known faults of a program. In doing so, the algorithm prioritizes the test cases that cover more faults of the program.

The rest of the prioritization techniques, $M_4, \ldots, M_9$, serve as heuristics when prioritizing test cases. Some of these techniques are based on code statements and code branching, which makes them applicable in a practical context. More specifically, a technique like M_4 counts the number of statements that are executed by a test case in some program. The test cases are then ordered by the number of statements that were executed in descending order. For example, if test cases τ_1, τ_2 and τ_3 have 5, 4, and 8 statement that are executed respectively, the order for the test cases would be (τ_3, τ_1, τ_2). Branch coverage is similar to statement coverage, where the number of branches are counted instead of statements.

A third kind of prioritization technique is the total fault-exposing-potential (FEP). As indicated by its name, FEP refers to the ability for some test cases to expose certain faults. For the interested reader, the technique is thoroughly explained under the name M_8 in Rothermel et al. [18].

1.3.2 Previous Machine Learning Implementations

Previous research have shown the use of machine learning as an implementation for the test case prioritization problem. Examples of machine learning models used in this research includes reinforcement learning [21], Support Vector Machines (SVMs) [5] and Deep Neural Networks (DNNs) [20].

1.3.2.1 Using Reinforcement Learning

One of the main ideas described in Spieker et al. [21] is the use of a history-based approach with reinforcement learning. The authors call this method the RETECS (Reinforcement Test Case Selection) method. It considers the steps of prioritization and selection, followed by test case scheduling for one continuous integration (CI) cycle. It follows the philosophy of minimizing the time between executing the tests and receiving feedback in the shape of errors and such, meaning that early execution of tests that fail are rewarded by their model.

The paper utilizes the problem formulation from Rothermel et al. [18] shown in Definition 1 above. Spieker et al. [21] goes on to state a different variation of the problem statement, *Time-limited Test Case Prioritization Problem* (TTCP), which takes the time limit of test executions into account as well. This formulation of TTCP served as a promising source of inspiration for the objective item (ii) (see Section 1.2). However, the authors' objective for the paper include taking CI into account. Therefore, the *Adaptive Test Case Selection Problem* (ATCS) is proposed which is derived from TTCP and takes historic test suite executions into account. The authors conclude that RETECS can be applied on ATCS, which means that the time aspect was solved within RETECS.

1.3.2.2 Using Support Vector Machines

The idea of test case prioritization has also been used with SVMs. This was done in Busjaeger and Xie [5]. The objective was to order a set of tests using training data with binary labels in order to maximize the ordering of tests from unseen data. The training was based on code changes and tests with an associated label, indicating whether the test passed or not. The model contain five features, which include a file modification metric, text path similarity, text content similarity, failure history and "age" of the test.

1.3.2.3 Using Artificial Neural Networks

The authors of Jahan et al. [13] proposed an Artificial Neural Network (ANN) approach to TCPP. Their approach consisted of utilizing feature extraction of multiple features and test case selection as means of preprocessing test case data. One such feature is the one referred to as the *number of modified modules* (MM). The term "module" refers to either a method or function. Based on the extracted features, tests are being selected to be used in the ANN classifier.

However, the creation of the dataset had a manual component. In order for the ANN to be trained, the authors resorted to manually label a dataset of test cases. Test

cases were labeled with a priority value of either 0, 1 or 2, where 0 signifies a low priority and 2 signifies a high priority. The label is decided based on two features, one of which being MM. The second feature is the *number of modified requirements* (MR) which is simply the number of changed requirements. The sum of MM and MR is calculated for each test case and then each test case is ordered in descending order based on the sum. After doing so, the priority class boundaries were decided by performing "some experiments".

1.3.2.4 Using Deep Neural Networks

Deep Neural Networks (DNNs), which is a kind of ANN, has also been put to use in the context of TCPP. Sharif et al. [20] implemented DeepOrder, a deep neural network for prioritizing test cases. The network consisted of one input layer of features, three hidden layers and one output layer. The input layer include features that described individual test cases. Duration of the test case is one such feature. The output layer consist of only one node, which is the assigned priority value of the inputed test case. The three hidden layers consists of 10, 20 and 15 neurons respectively, which was chosen based on experimentation.

The authors of DeepOrder faced issues of imbalanced data. As such, data augmentation was introduced (for more detail, see Section 2.4 and Section 3.2.1).

1.4 Limitations

The project was carried out in an industrial setting. However, this thesis did not aim to provide a perfect solution that can be used by employees at Aptiv (Aptiv Contract Services Sweden AB). The process of integrating the solution to Aptiv's system was deemed to not be feasible for the project's time span. As a potential future improvement, the solution could be further developed to be integrated into an industrial setting.

2

Theory

In this chapter, the theoretic parts of the project are laid out. The application of this theory is later realized in Chapter 3.

2.1 The Implementation of TCPP in The Project Setting

The problem statement in Section 1.2 is closely related to Definition 1, which defined the test case prioritization problem (explained in Section 1.3.1). In the context of this thesis project, Definition 1 can be instantiated, with some modifications, as the following:

Definition 2. We are given the following: a test suite denoted $\mathcal{T}$, the set of permutations of $\mathcal{T}$ denoted $\mathcal{PT}$ and a function $f : \mathcal{PT} \to \mathbb{R}$. We state the problem as the following: Find $\mathcal{M} : \Delta \to \mathcal{PT}$ such that for every revision with alteration $\delta \in \Delta$, given $f : \mathcal{PT} \to \mathbb{R}$, for all $\mathcal{T}' \in \mathcal{PT}$ we have that

$$f(\mathcal{M}(\delta)) \geq f(\mathcal{T}').$$

In this definition, Δ is the set of possible alterations to the current repository. Informally, Definition 2 states that some function $\mathcal{M}$ needs to be found. More precisely, in order to find $\mathcal{M}$, *every revision* of some repository needs to be taken into account. This is due to the fact that the behavior of f, the scoring function for a test suite, changes with each revision. The scoring of orders varies for each revision, which results in differences in what orders it considers to be good or not. With this caveat of revisions in mind, the aim is to find a test suite $\mathcal{M}(\delta) \in \mathcal{PT}$, which is based on an alteration δ, such that $\mathcal{M}(\delta)$ scores better than any other possible ordering $\mathcal{T}'$. It is important to note that Definition 2 is the ideal case of attempting to solve item (i) in the problem statement.

In the case concerning this thesis, $\mathcal{M}$ is a deep learning model. As such, this is not an ideal case. Therefore, it is not guaranteed that the machine learning model, $\mathcal{M}$, produces a perfectly prioritized ordered list of test cases, since $\mathcal{M}$ will merely be a prediction.

Another aspect to consider is *Test Case Selection* (TCS) in which a subset of tests in a test suite are selected which are to be executed, in order to reduce the execution time.

Spieker et al. [21] used this approach in their RETECS model (see Section 1.3.2.1). As expected by the name, TCS involves ignoring some test cases during the test execution. This is not without drawbacks, as some empirical evidence shows that fault detection capabilities are highly reduced [17]. However, Rothermel et al. [18] suggest that combining test case prioritization and test case selection might be reasonable in the cases where leaving out certain test cases are acceptable.

2.1.1 Types of Alterations, Δ_f and Δ_c

The different alterations, δ, that are explored in this project are of two kinds: file alterations, δ_f, and code alterations, δ_c. Each of these have a corresponding types of all possible alterations, Δ_f and Δ_c respectively. The shape of the vectors $\delta_f \in \Delta_f$ and $\delta_c \in \Delta_c$ are the following:

$$\delta_f = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{bmatrix}, \delta_c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n'} \end{bmatrix}$$

where f_i for $i = 1, \ldots, n$ are file names and c_i for $i = 1, \ldots, n'$ are code alterations. The first kind of alteration that can be considered for $\mathcal{M}$ is file alterations, denoted δ_f. A file is considered to be altered if any modification of the code within the file has occurred. The second kind of alteration that could be considered for $\mathcal{M}$ is code alterations, denoted δ_c. This kind of alteration considers changes in line-by-line. As one can imagine, the order of magnitude in the number of possible code alterations is significantly larger than file alterations, since the code alterations are on a much finer level than file alterations. However, in this thesis, only file alterations will be explored, due to time frame limitations discussed further in Section 5.3.

2.2 Artificial Neural Networks

Hassoun et al. [12] mentions that artificial neural networks are motivated by distributed, and very large parallel computations accomplished by the human brain that allows it to be so successful at recognition and classification tasks. While the development of neural networks can be found between 1950s and 1960s, it is not until years later, in the 1980s it has had an immense development [12], allowing a neural network to train on input data.

Neural networks is usually described with three kinds of layers: an input layer, hidden layers and an output layer [22]. Each layer consists of interconnected nodes, often referred to as neurons, and each connection is associated with a weight. These components together form the foundation to a trainable neural network.

When training a neural network, the input is passed through the hidden layers, and for each pass, an activation function is applied to the weighted sum of the neurons to

determine which neurons are to be considered in the next layer of the network. When an input has passed through the entire neural network, a loss value is computed, comparing the output with the actual ground truth. The loss value combined with the input is then used with a back propagation algorithm, whose purpose is to update the weights of the network, improving the neural network to achieve better predictions [22]. Note that a prerequisite for efficiently training a neural network is to have a sufficient amount of data [1].

While there exists multiple heuristics to solve TCPP [18], neural networks in the context of this thesis seems natural and could be helpful in finding how δ_f relates to test case failures. This is due to the fact that large projects have repositories that often contain lots of files, and it could be difficult to relate these files δ_f with their test case failures in a purely algorithmic way such that TCPP is solved efficiently.

2.3 The Knapsack Problem

The knapsack problem is possibly one of the more well known problems in computer science. The problem formulation goes as follows [14]: We have a set of n items $i = 1, 2, \ldots, n$, where each item has a value, v, and an integer weight, w, associated with it. We wish to find a subset of items, S, such that we maximize the total value that we can get, however the total weight must not be larger than some maximum integer capacity W. In formal terms we have the following:

$$\text{maximize} \sum_{i \in S} v_i$$

$$\text{subject to} \sum_{i \in S} w_i \leq W$$

Where i is an item, and the value and weight of that item are v_i and w_i respectively. The knapsack problem can be solved using dynamic programming with pseudo-polynomial time complexity. The weight w_i and W is sometimes referred to as a quantity of *time*, which is fitting for its use in TCS in the sense that time is a limiting factor to the problem.

2.4 Data Augmentation

Data augmentation is the practice of generating synthetic data, based on existing data. One way to achieve this is by using Generative Adversarial Networks (GANs) [8, 7]. GANs are neural networks that takes existing data and attempts at generating new data, while comparing the two. More specifically, GANs consist of two neural networks: a generator and a discriminator.

The generator generates synthetic data. The discriminator, on the other hand, is a classifier that attempts at distinguishing between real data and synthetic data from the generator. The goal for the generator is to produce synthetic data that the discriminator classifies as real. During the training phase of the GAN, two losses

are calculated: discriminator loss and generator loss. When the discriminator is being trained, only the discriminator loss is accounted for, since the purpose of the discriminator is only to distinguish between real data and synthetic data, as a classifier.

Apart from the the generator loss, the training of the generator also takes the following into account [9]:

- The discriminator network
- The discriminator output
- Random noise

The random noise is fed into the generator in order to produce a data instance, g. The produced data is then fed into the discriminator which classifies the data as either real or synthetic. The generator loss calculated based on the classification of g. Then backpropagation through the discriminator and generator is used to obtain gradients, which are then used to update the generator weights. After training a GAN model, synthetic data can be generated.

3

Methods

In this chapter, the methods and implementations of the project are described. The implementations are partly built by the application of the theory laid out in Chapter 2. The project consists mainly of three parts: building a dataset, building a neural network and applying Test Case Selection as a post-processing step. All of these parts are explained in detail.

3.1 The Pipeline

From the problem statement in Section 1.2, the main goal of the project is to create a Deep Neural Network, that orders test cases based on alterations in a repository. However, apart from this neural network, the dataset (described in further detail in Section 3.2) needed to train and test the model has to be created. As such, a pipeline that collects the relevant data and compiles it into one dataset was implemented. Figure 3.1 depicts a diagram of the pipeline.

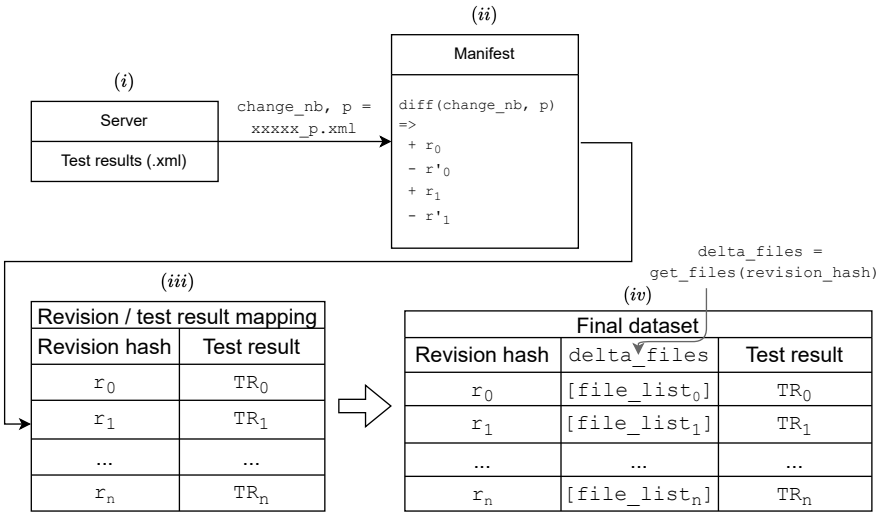


Figure 3.1: An abstract representation of the pipeline

First, test result reports are retrieved from a server, (i) in Figure 3.1, as XML files. These reports contain a list of test cases and whether or not a test case passed or failed based on a build. Each build is associated with a "change number" and a

patchset. The file names of the XML files are formatted in the shape of a change number (denoted `change_nb` and `xxxxx` in the diagram) and patchset (denoted `p` in the diagram) such that the file name is `xxxxx_p.xml`. Second, the filename is parsed and then used to find changes in a manifest repository, (*ii*) in Figure 3.1. The manifest contains a list of other repositories with their corresponding latest revision hashes. The latest revision hash is found for each repository by looking at additions only of a `git diff`. In doing so for every change number and patchset pair, a mapped table ((*iii*) in Figure 3.1) of revision hashes, r_i, and test results, TR_i is acquired. Finally, the altered files in the repository given a revision hash, are listed and stored in the final dataset, (*iv*) in Figure 3.1, along with revision hash itself and the corresponding test results.

3.2 The Dataset

In order to train the neural network (detailed in Section 3.3), a dataset with suitable features and labels are needed. Those features are alterations, δ, and the labels are failed test cases corresponding to δ, denoted $\mathcal{T}_{\text{failed}} \subseteq \mathcal{T}$ for some test suite $\mathcal{T}$. The two types of features are discussed in further detail in Section 2.1.1. However, in this project, only file alterations are considered as the input to the neural network. The dataset concerning file alterations is compiled in the matrix $\mathbf{X}_f$, where each row, $i = 1, 2, 3, \ldots, r$, is one datapoint containing n files[1]. Each row is a bit vector, meaning that each entry $f_{i,j}$ is either 1 or 0, indicating whether file j is included or not. $\mathbf{Y}_{\mathcal{T}_{\text{failed}}}$ is the label matrix. Each row corresponds to a row (datapoint) in $\mathbf{X}_f$, where each entry $\tau_{i,j}$ is also a bit vector; if $\tau_{i,j}$ is set to 1, it indicates that test case j failed, otherwise it is set to 0. Each row in $\mathbf{Y}_{\mathcal{T}_{\text{failed}}}$ has m test cases. $\mathbf{X}_f$ and $\mathbf{Y}_{\mathcal{T}_{\text{failed}}}$ is therefore defined as following:

$$\mathbf{X}_f = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & f_{2,3} & \cdots & f_{2,n} \\ f_{3,1} & f_{3,2} & f_{3,3} & \cdots & f_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_{r,1} & f_{r,2} & f_{r,3} & \cdots & f_{r,n} \end{bmatrix}, \mathbf{Y}_{\mathcal{T}_{\text{failed}}} = \begin{bmatrix} \tau_{1,1} & \tau_{1,2} & \tau_{1,3} & \cdots & \tau_{1,m} \\ \tau_{2,1} & \tau_{2,2} & \tau_{2,3} & \cdots & \tau_{2,m} \\ \tau_{3,1} & \tau_{3,2} & \tau_{3,3} & \cdots & \tau_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_{r,1} & \tau_{r,2} & \tau_{r,3} & \cdots & \tau_{r,m} \end{bmatrix}$$

In order to create bit vectors of the file names, which are strings, label encoders from the scikit-learn library [4] are used. A label encoder maps a string to a unique integer, much like an index. With these indices, bit vectors can be created by taking a vector of zeros and substituting the zeros with the corresponding index with a one. The same technique is used when creating a bit vector for test cases.

For example, let `main.py`, `main.hs` and `amazingHelperFunctions.hs` be altered files and map these to the integer indices 4, 6 and 9 respectively. Let us assume there are a total $n = 10$ files that are taken into account for a particular instance of

[1]For the sake of explanation, mathematical conventions such as 1-based indexing have been used.

the neural network. In order to create a bit vector, we begin with a zero vector of size $n = 10$:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

In order to yield a bit vector, the indices 4, 6 and 9 in the zero vector are set to 1:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

.

3.2.1 Expanding The Dataset with Data Augmentation

Due to the lack of data that was collected at Aptiv for the dataset, data augmentation was introduced (explained in detail in Section 2.4). Apart from GANs, models such as SMOGN were also considered [3] which was used in Sharif et al. [20] due to the lack of relevant test cases. In the end, Wasserstein Generative Adversarial Networks with Gradient Penalty [11] (WGAN-GP for short) were used for the purposes of this thesis. The details of how WGAN-GP works and how it generates the synthetic datapoints is outside the scope of this thesis. As such, the interested reader can read up on the details of WGAN-GP and its predecessor WGAN in Gulrajani et al. [11] and Arjovsky et al. [2] respectively.

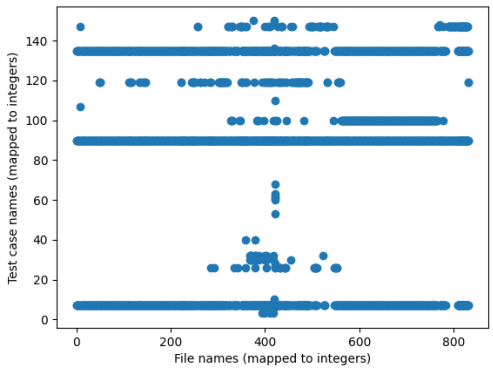
In order to generate synthetic data, a helpful library that puts the theory of GANs into practice is needed. Thus, a Python library called YData Synthetic [23] was used. As discussed in Section 2.4, some real data is needed in order to create synthetic data, that attempts to mimic the original data. Figure 3.2 depicts six scatter plots with "exploded data" for three different projects that have been investigated during the run of this thesis project. The term "exploding data" refers to the transformation of data within one datapoint to individual datapoints. For example, if we have a vector of file names, δ_f, and failed test cases, $\mathcal{T}_{\text{failed}}$, within a *single* datapoint, $\mathbf{x}$, we can take every file-test case pair into its own datapoint by "exploding" it:

$$\delta_f = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}, \mathcal{T}_{\text{failed}} = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} \delta_f & \mathcal{T}_{\text{failed}} \end{bmatrix}$$

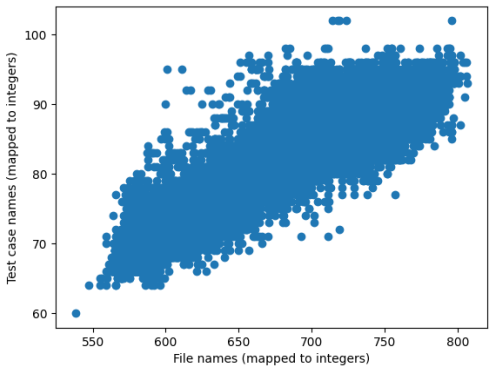
$$\mathbf{x} \xrightarrow{\text{Explode}} \mathbf{X}', \mathbf{X}' = \begin{bmatrix} f_1 & \tau_1 \\ f_2 & \tau_1 \\ f_3 & \tau_1 \\ f_1 & \tau_2 \\ f_2 & \tau_2 \\ f_3 & \tau_2 \end{bmatrix}$$

The reason for using exploded data is because the Python library used to generate the synthetic data requires the data to be in this form. This posed the problem of preserving the *relationships* between altered files and failed test cases. After all, the datapoints (f_1, τ_1), (f_1, τ_2), (f_2, τ_1) and (f_2, τ_2) says a lot less on their own than

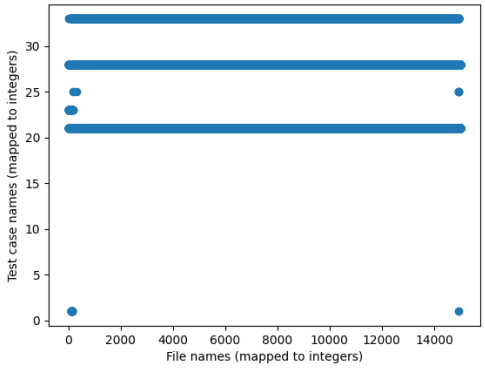
the single datapoint containing $\delta_f^\intercal = \begin{bmatrix} f_1 & f_2 \end{bmatrix}$ and $\mathcal{T}_{\text{failed}}^\intercal = \begin{bmatrix} \tau_1 & \tau_2 \end{bmatrix}$. This is because in the non-exploded datapoint gives meaning and a relation between files and test cases.



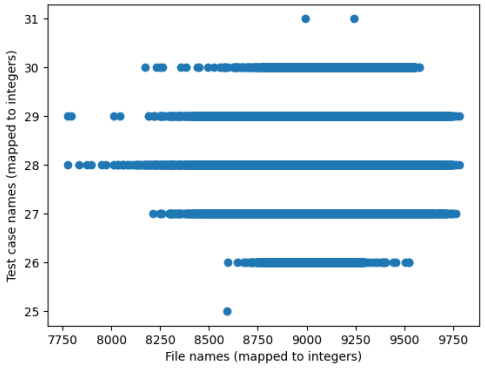
(a) Real data from Project 0



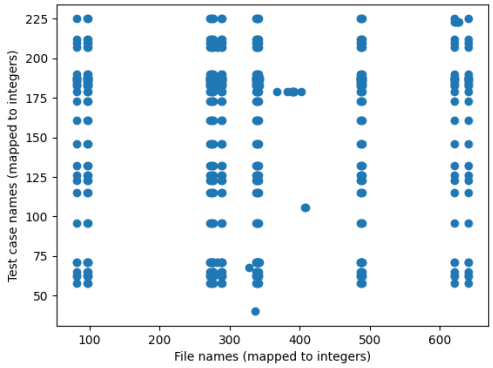
(b) Synthetic data for Project 0



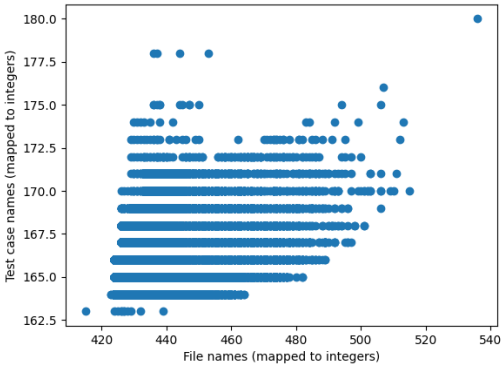
(c) Real data from Project 1



(d) Synthetic data for Project 1



(e) Real data from Project 2



(f) Synthetic data for Project 2

Figure 3.2: Scatter plots of the real data accumulated from the different projects that has been investigated and synthetic data generated using WGAN-GP from real data from the different projects.

This posed a problem at first since the generated data was also exploded. This was

solved by entering a third column for unique identifiers, that needed to be generated. Specifically, the unique identifier was chosen to be a revision hash that has been mapped to a unique integer, since this is the format that was used for files and test cases. The distribution of identifiers attempts to mimic the original data. This means that each datapoint in the generated data was on the form (u, f, τ), where u is the unique identifiers, f is a file and τ is a test case. In order to "unexplode" the data, every datapoint that shared the same identifier was merged. This is effectively the inverse of the explode operation explained above. As such, data was created that more accurately mimics the real data.

As an extra benefit, the exploded data can be used to plot the data on a plane. Each row in $\mathbf{X}'$ can be plotted where file "names" are on the x-axis and test case "names" are on the y-axis, as in Figure 3.2. In this thesis project, three different company projects are considered. Those projects are anonymized as *Project 0*, *Project 1* and *Project 2* in order to distinguish them.

It is evident from the scatter plots above that some tests are more predominant than others across multiple source code files. This is what gives the plots "lines" of dense datapoints as tuples, (f_i, τ_j). The plot in Figure 3.2e can be differentiated from the others with its vertical lines, as opposed to the horizontal lines in Figure 3.2a and Figure 3.2c. One possible explanation for this might be the fact that some of the files in the plot for Project 2 didn't associate with any test cases when the dataset was built. In formal terms, it could be expressed that $|\delta_f| \geq 1$ and $|\mathcal{T}_{\text{failed}}| = 0$ for a single datapoint before it was exploded. As such, no point on the scatter plot would be possible.

As mentioned previously, in order to generate new synthetic data, real data is needed. By using the real data depicted in Figure 3.2a, Figure 3.2c and Figure 3.2e, data could be generated, which can be seen in Figure 3.2b, Figure 3.2d and Figure 3.2f.

Some remarks can be made regarding the plots of the synthetic data. The generated data from Project 1 (Figure 3.2d) and Project 2 (Figure 3.2f) appear to generally follow the same pattern as their corresponding real data. The synthetic data for Project 1 has five relatively defined "lines" of densely packed datapoints like three "lines" found in the real data. Both the synthetic dataset and real dataset contain deviating points for some less failure prone test cases; in the synthetic dataset those test cases would be number 31 and 25, while in the real dataset those being number 25, 23 and 1.

The synthetic data for Project 0 does not appear to mimic its real data correspondence, however. The possible reasons for this are numerous. The real data contains three predominant test cases that each form a "line" of points, however the spread of test cases in Figure 3.2a might have had an effect in the creating the very diverse set of test cases and files, that does not conform to "lines". Table 3.1 contain the hyperparameter settings that were used in producing the synthetic data. These settings were yielded through experimentation in a trial-and-error approach. The experimentation mostly involved tuning the hyperparameters based on the results yielded in the plot format shown in Figure 3.2b, Figure 3.2d and Figure 3.2f. Attempts were made to mimic the real data correspondence as closely as possible.

Hyperparameter	Value
Learning rate	0.01
Batch size	32
First beta coefficient	0.6
Second beta coefficient	0.8
Number of epochs	20
Noise dimension	128
Layer dimension	128

(a) Settings for Project 0 and Project 2.

Hyperparameter	Value
Learning rate	0.01
Batch size	256
First beta coefficient	0.6
Second beta coefficient	0.8
Number of epochs	20
Noise dimension	200
Layer dimension	128

(b) Settings for Project 1.

Table 3.1: Hyperparameter settings for the three projects.

3.3 Implementing The Prioritized Order Model

The process of writing and designing a neural network that fits the project setting included some experimentation. Previous research served as guidance for the direction of what the final model might look like and the main source of inspiration for this stem from Sharif et al. [20]. In order to test the format of the dataset, a simple neural network was established. The structure of this network was inspired by DeepOrder [20] which is a model that uses a Deep Neural Network (DNN) in order to assign priority values, given a test case that consists of a number of different features.

The proposed model for this thesis project is called Prioritized Order Model (POM). The idea of the network differs substantially from DeepOrder especially in terms of the input features and the network output. POM takes a vector of file alterations, δ_f, as input and produces a vector of probabilities for how each test case is to fail, $\mathbf{p}$. As such, the greatest inspiration from DeepOrder was the network structure in terms of layers and number of neurons. The structure served as a base on which later could be altered to suit the needs of the project.

In contrast to the ANN introduced in Jahan et al. [13], POM does not take TCS into account as a step in preprocessing the data. Instead, it is dealt with after the prediction is made. This also different to how RETECS (see Section 1.3.2.1) integrated TCS. The approach for TCS in Jahan et al. [13] is based on just one of their features called MM (see Section 1.3.2.3). The selected test cases to be prioritized using the ANN, are chosen if MM ≥ 1 for some test case τ. The implementation of TCS in POM is explained further in Section 3.4.

Additionally, as mentioned in Section 1.3.2.3, the ANN in Jahan et al. [13] has features based on modified modules among others. These features are fundamentally different from that of POM. As also mentioned in Section 1.3.2.3, the dataset for training the ANN required that test cases where labeled by performing experiments. Test cases were labeled either 0, 1 or 2, which consequently decreases the granularity of the orders produced. However, the experiments mentioned in Jahan et al. [13] are not elaborated upon.

3.3.1 The Structure of POM

POM is a sequential neural network that consists of three hidden layers, each with a certain number of neurons, an input layer and a output layer. The details of the model is comprised in Table 3.2, which include the number of neurons of each layer, as well as the activation functions that are used between each layer. Figure 3.3 shows an overview of the network.

Layer or Activation Function (AF)	**Type**	**Number of neurons**
Input layer	Linear	$\lvert\delta_f\rvert$
AF	Mish	N/A
Hidden layer	Linear	10
AF	Mish	N/A
Hidden layer	Linear	20
AF	Mish	N/A
Hidden layer	Linear	20
AF	Mish	N/A
Output layer	Linear	$\lvert\mathbf{p}\rvert$
AF	Softmax	N/A

Table 3.2: The structure of POM in a table format.

As previously mentioned, POM was inspired by DeepOrder [20]. However, the number of neurons for each layer differs slightly between the two, since the proposed structure in Table 3.2 was found to be better for POM. As mentioned in Section 1.3.2.4, DeepOrder has 10, 20 and 15 neurons in its hidden layers. As also mentioned in Section 3.3, the input features and output of the network is fundamentally different. The choice of activation function for DeepOrder is Mish, which is introduced in Misra [15]. The main motivation by Sharif et al. [20] for this choice, instead of more conventionally known activation functions such as ReLU, was mainly due to how Mish had overcome known flaws in ReLU. From testing and experimentation, the conclusion was that Mish performed better than ReLU in POM, which is why Mish was ultimately chosen over ReLU.

In Table 3.2, $\delta_f^\intercal = \begin{bmatrix} f_1 & f_2 & \cdots & f_n \end{bmatrix}$ denotes the vector of files that are present in the input layer and $\mathbf{p}^\intercal = \begin{bmatrix} p_1 & p_2 & \cdots & p_m \end{bmatrix}$ is the vector of test case failure probabilities that are present in the output layer. The probabilities are based on the input which are files. The last activation function of the model is a softmax function, which converts vectors into a probability distribution such that all probabilities sum to 1. In formal terms, for every prediction, $p_1, p_2, \ldots, p_m$, where $p_i \in [0, 1]$ it is the case that $\sum_{i=1}^{m} p_i = 1$. The predicted probability distribution is then compared to the ground truth with a cross entropy loss [16].

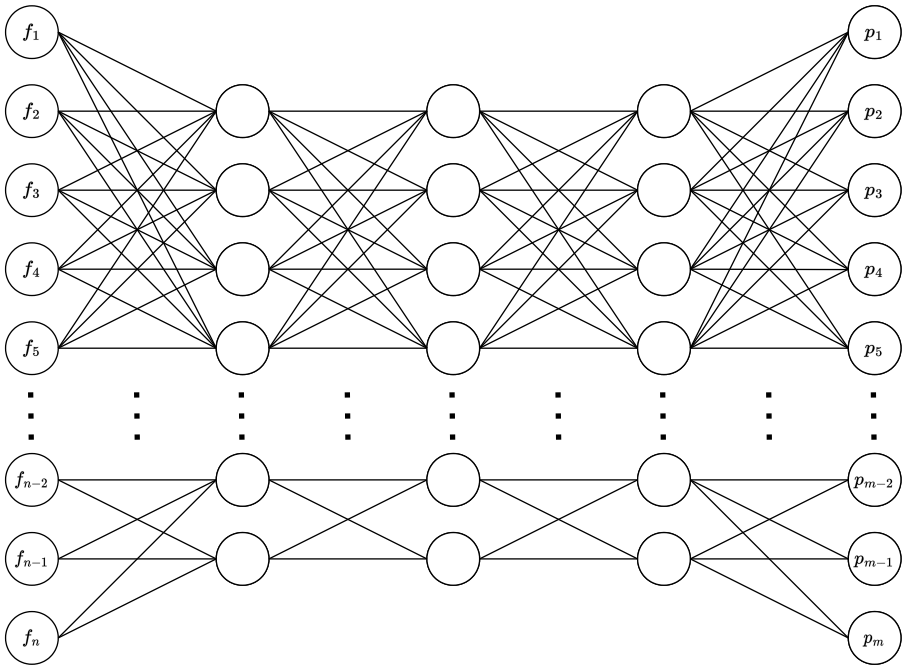


Figure 3.3: An overview of POM, that takes in file names, f, that are represented as bit vectors, and outputs test case failure probabilities, p.

3.4 Test Case Selection, TCS

As discussed in Section 3.3.1, the model outputs a vector of probabilities, each of which is associated with a test case. To achieve a prioritization based on the model output, it is sufficient to order the list of test cases in descending order based on the probability. However, from the problem statement in Section 1.2, item (ii) states that the total time of the ordered list of test cases should not exceed a given time limit. POM does not take time as a feature into consideration. Instead, *Test Case Selection* (TCS) is done as a post-processing step. This section will describe how TCS is applied to the model output.

For all test cases in a test result, there exists an execution time for each test case. As such, each output probability is associated with the average execution time of the test cases. With this mapping between test case probability and the average time it takes to execute the test, the objective is to maximize the sum of test case probabilities subject to the time it takes to execute each test. Hence, the idea of using the knapsack problem (see Section 2.3) to apply such a time constraint seems appropriate.

To apply TCS to the test prioritization, an instance of the knapsack problem can be created. We are given the following: the vector of probabilities $\mathbf{p}$ produced by the model, a vector of average times, $\begin{bmatrix} t_1 & t_2 & t_3 & \cdots & t_m \end{bmatrix}$, a set S containing indices, i, for each test case (which are chosen based on p_i and t_i), and a time limit T. Applying the knapsack problem for this instance, we get the following:

$$\text{maximize} \sum_{i \in S} p_i$$

$$\text{subject to} \sum_{i \in S} t_i \leq T$$

In this thesis, dynamic programming is utilized to solve the knapsack problem, and the time complexity is pseudo-polynomial, which can be problematic when the input size is too large [14]. For this project, however, there were no problem with the running time of the algorithm.

To achieve a final, time constrained prioritization, only the test cases selected by the algorithm are included. The overall idea is depicted in Figure 3.4

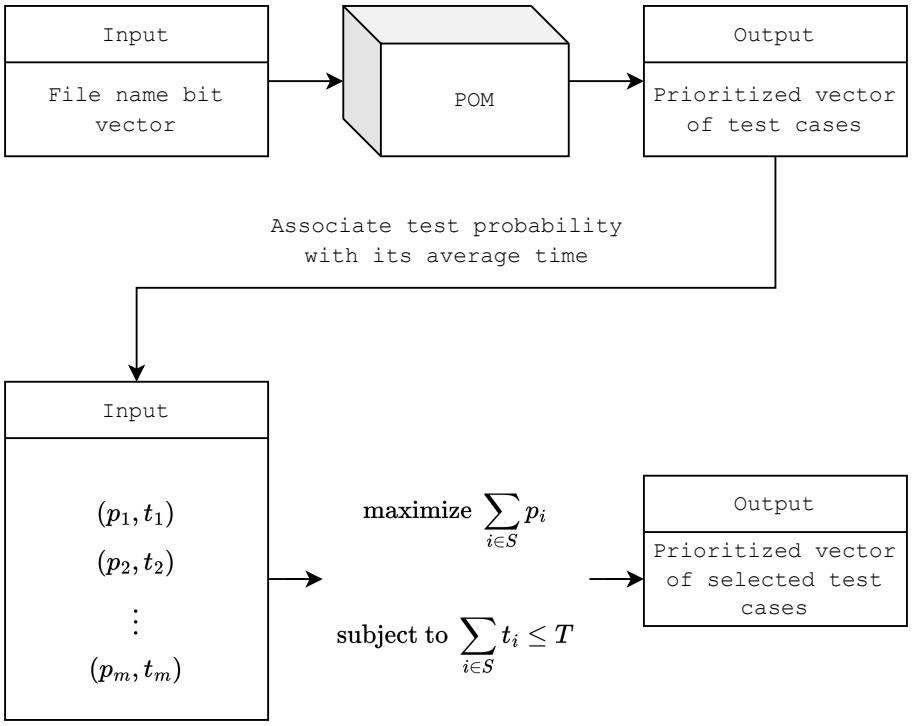


Figure 3.4: The workflow when considering time into the prioritization.

4

Results & Evaluation

In this chapter, the results of performing test case prioritization with POM and test case selection utilizing the knapsack problem are presented. To further assess the results, relevant baseline methods are utilized to compare the use of machine learning in a test case prioritization context.

4.1 Evaluation Methods

When evaluating POM, different methods had to be developed that suited the needs and context for test case prioritization and test case selection. From previous literature [6, 18, 20, 21], Average Percentage of Faults Detected (APFD) seems to be a common evaluation metric for test case prioritization. APFD is described as a measurement of the effectiveness of fault detection in a prioritized order [18]. However, APFD is not directly applicable to evaluate POM, since it requires faults to be known [6]. For this thesis, it it only possible to know whether a test case passed or not which is why alternate methods has been developed to achieve similar measurements.

4.1.1 Time Until First Failed Test Case, TUFFTC

The first method of evaluation is to consider is when the first failure occurs in the ordering. This can be used to easily get a sense of the meaningfulness of the prioritized result. This metric is referred to as *Time Until First Failed Test Case* (TUFFTC).

4.1.2 Test Coverage

To further assure that POM produces a meaningful prioritization, a form of test coverage is performed on the prioritized output. The test coverage is defined as follows:

$$\text{Test coverage} = \frac{\text{Number of correctly predicted test cases}}{\text{Total number of test cases that have failed}}$$

The definition by itself might seem straight forward. However, the quantities for "the number of correctly predicted test cases" and "the total number of test cases

that have failed" have to be defined as well.

The number of correctly predicted test cases is acquired by looking at the label dataset $\mathbf{Y}_{\mathcal{T}_{\text{failed}}}$ from Section 3.2. As discussed there, each row is one datapoint containing a number of failed test cases. Let k_d be the number of failed test cases for some datapoint d, then k_d is defined as

$$k_d = \sum_{i=1}^{m} \tau_{d,i}.$$

Let $\mathcal{T}_{\text{predicted}}$ be the set containing the first k_d test cases from the prediction vector of test cases for some datapoint d. Let $\mathcal{T}_{\text{failed}}$ be the set of failed test cases from the datapoint d. $\mathcal{T}_{\text{failed}}$ is also known as the label for d. It is the case that $|\mathcal{T}_{\text{predicted}}| = k_d = |\mathcal{T}_{\text{failed}}|$. From this, the number of correctly predicted test cases for this datapoint, is $c_d = |\mathcal{T}_{\text{predicted}} \cap \mathcal{T}_{\text{failed}}|$. In words, this is the act of checking which test cases in $\mathcal{T}_{\text{predicted}}$ are in $\mathcal{T}_{\text{failed}}$ and then count every occurrence of such test cases. This is just for *one* datapoint. The number of correctly predicted test cases would therefore be

$$\sum_{d=1}^{r} c_d$$

where r is the total number of datapoints. The total number of test cases that have failed can be defined as

$$\sum_{d=1}^{r} k_d.$$

4.1.3 Prioritization Methods for TCPP

For the TCPP evaluation, three baseline methods have been compared to POM and are described below:

- **Default**: refers to the original test suite, without any modifications to the ordering.
- **Prioritized**: refers to the prioritized test suite computed by POM.
- **Random**: creates a random ordering of the test suite for each iteration of testing phase. This is to avoid the risk of finding a superior ordering by coincidence.
- **Frequency sort**: creates an ordering in the test suite based on how often a test case fails overall, meaning that test cases failing more often will be prioritized.

The results of these methods can be found in the columns of Table 4.1 and Table 4.4.

4.1.4 Test Case Selection Evaluation Methods

For the TCS evaluation, dummy time data is sampled from the discrete uniform distribution for each test case. The reason for this is that time data for test cases was not available in the initial phase of the project.

Evaluating POM with TCS differs somewhat from the previous methods. Due to the time constraint, the prioritized output will be a subset of all test cases produced by POM, which can cause some trade offs. For instance, the time to execute the TCS prioritization can be reduced significantly. However, there is a risk that potential test case failures are excluded from the selected subset. An interesting point to investigate is whether a failure is included or not in the selected subset. Thus, the ratio between the number of failures in the selected subsets and the total number of failures is calculated. This ratio is referred to as *failure inclusion*. Test coverage is also included in the evaluation of the TCS variant of POM to further assess its performance. Note that in this case, test coverage is performed on the TCS test suite which is a subset of the prioritized test suite.

When applying TCS, the knapsack method and a naive solution are compared. Further description of each method can be seen below:

- **Naive**: refers to a naive way to applying TCS to the prioritized test suite produced by POM. This done by selecting the first test cases, that together do not exceed a specified time limit.
- **Knapsack**: selects a subset of test cases based on the probabilities that has been produced by POM, with respect to a specified time limit.

The results of these methods can be found in the columns of Table 4.3 and Table 4.6.

4.2 Evaluation Results Using Real Data

Table 4.1 contains data on the average time it takes to find the first failed test case in the total list of test cases that failed. Each row represents the average TUFFTC for a specific project and each column describes which method is used. It is evident that the prioritized ordering finds the first failure faster in every instance tested compared to the default and the randomized ordering. However, the frequency sort performs better than POM in all cases.

Project	**Default [tu.]**	**Prioritized [tu.]**	**Random [tu.]**	**Frequency sort [tu.]**
Project 0	51229	2944	31180	2102
Project 1	11695	27	5106	0
Project 2	53908	3970	19794	643

Table 4.1: Average TUFFTC for the different projects with four types of orderings of real data.

The results from performing the test coverage validation using POM and on test data from the different projects can be found in Table 4.2.

Project	**Prioritized test coverage**	**Frequency sort test coverage**
Project 0	82%	82%
Project 1	88%	92%
Project 2	87%	87%

Table 4.2: Test coverage for the prioritized and frequency based orderings of real data.

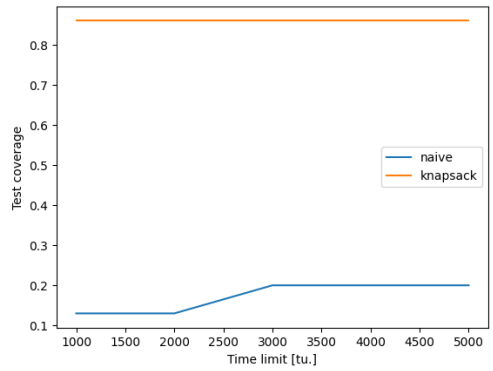
4.2.1 Test Case Selection Results

The Test Case Selection results can be found in Table 4.3, including different time limits used and its corresponding test coverage and failure inclusion by each method. A greater time limit usually implies that more tests are included in the prioritized order.

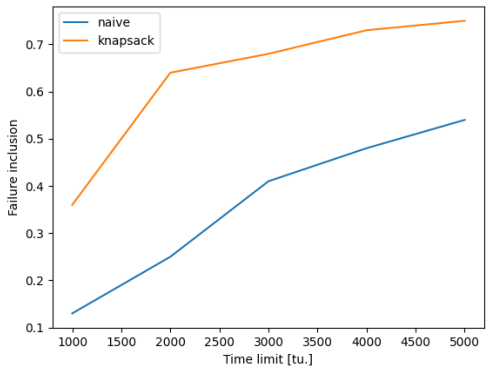
	Time limit	**Test coverage**		**Failure inclusion**	
	[tu.]	**Naive**	**Knapsack**	**Naive**	**Knapsack**
Project 0	1000	13%	86%	13%	36%
	2000	13%	86%	25%	64%
	3000	20%	86%	41%	68%
	4000	20%	86%	48%	73%
	5000	20%	86%	54%	75%
Project 1	1000	100%	100%	77%	76%
	2000	100%	100%	88%	88%
	3000	100%	100%	88%	88%
	4000	100%	100%	88%	88%
	5000	100%	100%	88%	88%
Project 2	1000	10%	60%	11%	42%
	2000	10%	60%	31%	54%
	3000	10%	60%	33%	68%
	4000	10%	60%	40%	59%
	5000	10%	60%	44%	62%

Table 4.3: Test coverage and failure inclusion result using different time limits for the TCS version of test prioritization (real data).

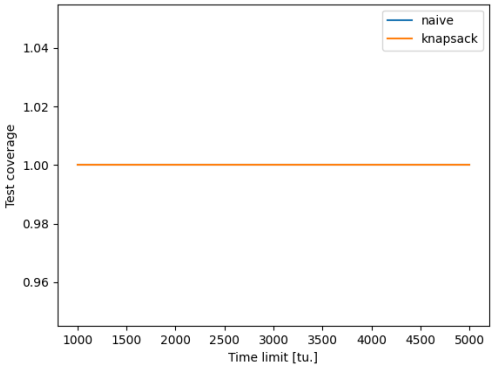
Each project was tested on different time limits, to analyze how an increased time limit affects the overall result. For this round of evaluation, an upper time limit of 5000 is chosen based on testing and re-testing with different time limits. Table 4.3 is visualized in Figure 4.1, where the orange line corresponds to the knapsack method, while the blue line refers to the naive method.



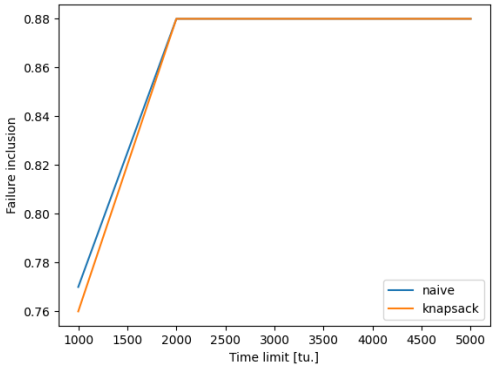
(a) Test coverage for Project 0.



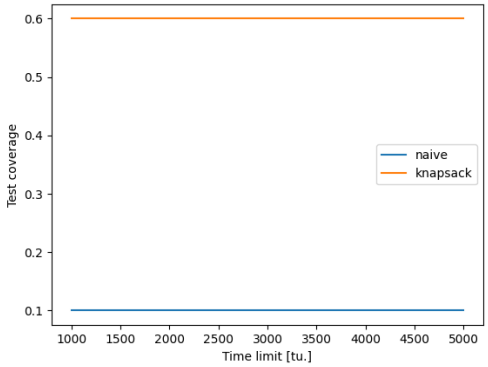
(b) Failure inclusion for Project 0.



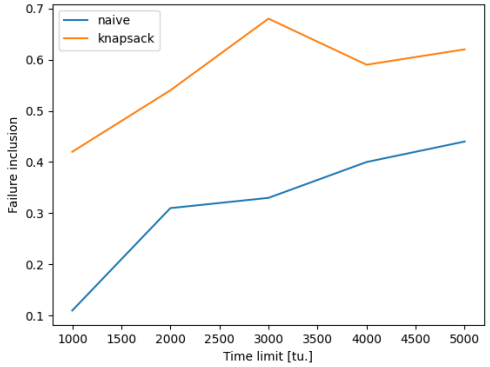
(c) Test coverage for Project 1.



(d) Failure inclusion for Project 1.



(e) Test coverage for Project 2.



(f) Failure inclusion for Project 2.

Figure 4.1: Comparisons between the naive method against the knapsack problem for test case selection using the real data.

4.2.2 Remarks on Results Using Real Data

While the results appear promising for POM, it is important to reflect over its meaning. For each project, there were roughly five to ten test case failures among a total of at most 30 test results that were used in the validation. In total, the number of datapoints varied between 150 to 200 between each project. Note that the amount

of validation data, and data overall in this step is extremely low in a deep learning context.

4.3 Evaluation Results Using Data Augmentation

This section aims to describe the results using synthetic data to further assess the possibilities of machine learning in test case prioritization. From Section 4.2, the results show that POM does not prioritize better than the frequency based method. It is probable that the low amount of data might be the reason for this, which is why the results in this section is based on generated data.

The amount of synthetic data in each project differed, and the number of datapoints in the test set for each project can be seen below:

- Project 0: 7806 rows of test data
- Project 1: 2720 rows of test data
- Project 2: 3198 rows of test data[1]

The average TUFFTC can be found in Table 4.4. Comparing the prioritized column, it seems like the model manages to prioritize a test suite more efficiently compared to the other baseline methods.

Project	**Default [tu.]**	**Prioritized [tu.]**	**Random [tu.]**	**Frequency sort [tu.]**
Project 0	7253	1288	7889	3976
Project 1	733	36	357	116
Project 2	488	72	954	127

Table 4.4: Average TUFFTC for the different projects with four types of orderings of synthetic data.

Moreover, the results from using the test coverage validation on the prioritized and the frequency based method are depicted in Table 4.5. It is evident from Table 4.5 that the test coverage of the prioritized method is higher compared to the frequency based method.

Project	**Prioritized test coverage**	**Frequency sort test coverage**
Project 0	55%	23%
Project 1	73%	66%
Project 2	71%	60%

Table 4.5: Test coverage for the prioritized and frequency based orderings of synthetic data.

[1] A different train and test split is used due to the amount of data generated for project 2.

4.3.1 Test Case Selection Results using Data Augmentation

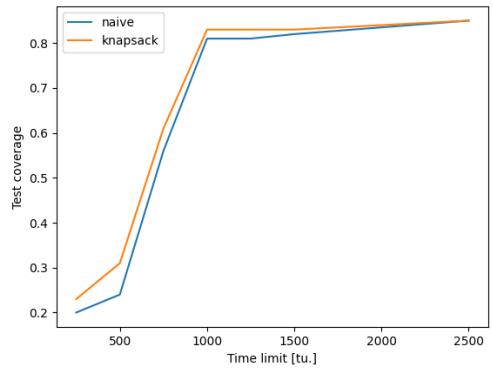
The results of using TCS with the augmented data can be seen in Table 4.6 and Figure 4.2. The time limits differ substantially compared to the limits used in Section 4.2.1, due to the general increase in the amount of data. It was found that using more data, an upper time limit of 2500 was found to be enough to test due to the execution time and the results achieved.

	Time limit	**Test coverage**		**Failure inclusion**	
	[tu.]	**Naive**	**Knapsack**	**Naive**	**Knapsack**
Project 0	250	20%	23%	14%	25%
	500	24%	31%	16%	27%
	750	56%	61%	38%	48%
	1000	81%	83%	55%	65%
	1250	81%	83%	55%	62%
	1500	82%	83%	56%	62%
	2500	85%	85%	58%	62%
Project 1	250	65%	81%	37%	81%
	500	74%	86%	47%	64%
	750	91%	86%	47%	64%
	1000	96%	97%	79%	83%
	1250	98%	98%	79%	82%
	1500	99%	99%	79%	82%
	2500	99%	99%	79%	82%
Project 2	250	59%	64%	26%	64%
	500	93%	94%	41%	68%
	750	94%	94%	55%	75%
	1000	94%	95%	55%	65%
	1250	96%	96%	68%	72%
	1500	96%	97%	70%	71%
	2500	98%	98%	71%	70%

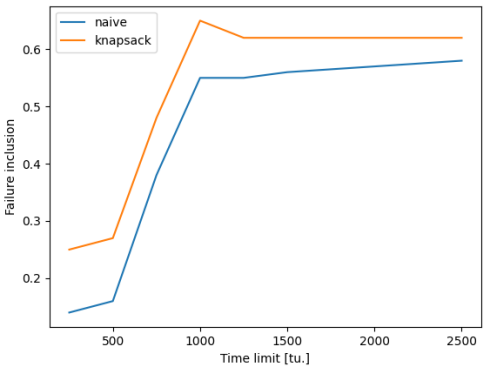
Table 4.6: Test coverage and failure inclusion result using different time limits for the TCS version of test prioritization (synthetic data).

4.3.2 Remarks on Results Using Data Augmentation

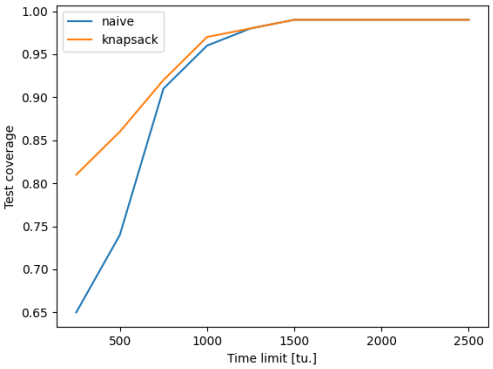
It is important to note that the generated data does not completely reflect the reality of the actual data. The main of goal of utilizing data augmentation is to explore the possibilities of machine learning in test case prioritization.



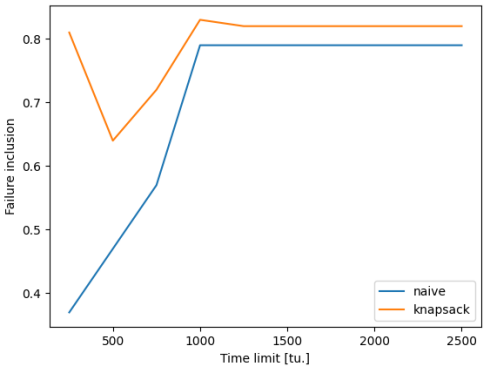
(a) Test coverage for Project 0.



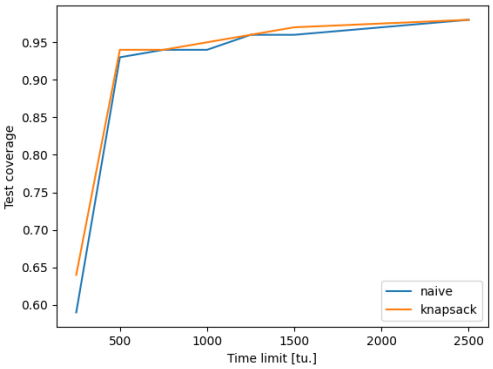
(b) Failure inclusion for Project 0.



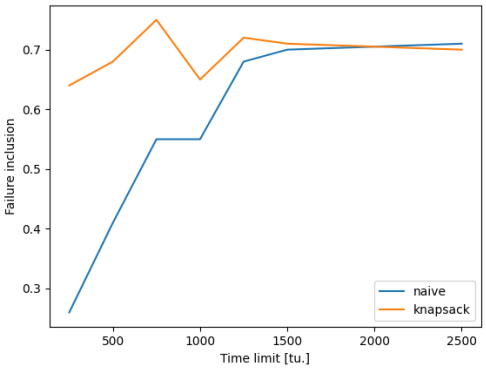
(c) Test coverage for Project 1.



(d) Failure inclusion for Project 1.



(e) Test coverage for Project 2.



(f) Failure inclusion for Project 2.

Figure 4.2: Comparisons between the naive method against the knapsack problem for test case selection using synthetic data.

5

Discussion & Conclusion

This chapter aims to discuss the results presented in Chapter 4 and how the project can be further developed to produce stronger results.

5.1 Discussion

From Section 4.2, it could be observed that POM managed to prioritize a test suite, especially when looking at the time it takes to find the first failure. However, comparing the time of failure with the model output to the frequency based method, it could be concluded that the frequency based method performed slightly better. Investigating the cause for this, the amount of real data and the number of test case failures in that data provided turned out to be insufficient to train a neural network. Moreover, it is difficult to validate the output from POM, since the optimal test ordering is not known. Therefore, during the training of the network, the predicted test suite produced by POM is compared to the original test suite. Then, to produce a final ordering, POM's output probabilities are sorted in a descending order which consequently favors the frequency of test case failures. This is the probable reason for why the frequency based method is such a good contender to POM from the results in Section 4.2.

To attempt to improve the performance of POM, data augmentation was introduced. With this, new data could be generated based on the data that was available to increase the data size and possibly its diversity. The hypothesis was that with more data available, POM should be able to perform better than the frequency based method, which was indeed the case. However, it is important to note that once augmented data is introduced, the meaning of the result changes. For instance, the result can not be used by the company, since it does not rightfully reflect the reality. However, the results based on data augmentation can be a good indicator of what is needed to achieve good results using machine learning and neural networks. In this case, increasing the data size, while also introducing more diversity to the data is beneficial. Based on the results in Section 4.3, one of the main takeaways is that the data seems plausible in a real world setting, which in turn allows POM to perform well. One possible solution to achieve this could be to spend more resources into creating datasets specifically for test case prioritization. This is to introduce more test case failures in the dataset to increase the diversity, to allow a model to better understand how file changes relate to test case failures.

Furthermore, utilizing data augmentation to generate data, it is hard to avoid that the generated data will be preferable over the original data. This is due to the generative model, since it is trained to generate expanded datasets, which ultimately will favor POM. Hence, it is important to be cautious about the generated data and to reason about how and why it produces the result it does. Based on the structure and looks of the generated data compared to the original data however, it seems plausible in this case that it could be likely to appear in a real world scenario.

Looking at the test coverage column in Table 4.3, the measured values rarely change as the time limit and the failure inclusion both grow. The hypothesis is that the number of test case failures in the test data are low and the quality of the prioritization by POM is quite low. Thus, as the time limit grows, new failures will not be detected since their priority value might be low, or that more failures does not exist in the test data.

Comparing POM to DeepOrder by Sharif et al. [20], the key difference is the input. DeepOrder processes individual test case features and predicts a probability to eventually provide a prioritization while POM looks at file changes exclusively. It is not apparent which of the methods is preferable; it could depend on the data that is available. If each test case contains useful features to determine a priority, using test cases as input might be more desirable. Also, if time is included in the test case features, it would be easier to construct a model that takes time into consideration in the final priority. As for today, POM prioritizes a test suite which is then processed by another algorithm to apply TCS to the problem. However, without such information for each test case, looking at file changes might be preferable since they usually can be easily found in the version control history.

5.2 Conclusion

In this thesis, a machine learning based approach for TCPP, using deep neural networks is presented. While the results from Section 4.2 were not along the lines with the initial hypothesis, drastic improvements could be seen in Section 4.3, when data augmentation was introduced. What could be observed is that when a diverse and sufficient amount of data exists, the results of using POM were remarkable, indicating that the approach works rather well in the context of TCPP.

Furthermore, as a post-processing step, a solution for TCS was implemented utilizing the knapsack problem. The implementation considered time as an aspect that introduces a time constraint to the original problem. Looking at the TCS results, they are not as convincing compared to the TCPP results. However, it can be observed that using the knapsack problem in a TCS setting works.

5.3 Future Work

As discussed in Section 2.1.1, two types of alterations were proposed: file and code alterations. In this project, only file alterations were touched upon, due to the limited

time frame and scope of this master thesis. However, as mentioned in Section 1.3.2.2, Busjaeger and Xie [5] used text content similarity as a feature in their SVM model. Their method for this were based on the work of Saha et al. [19], while adding their own modifications. With this, code alterations might serve as a promising aspect that merits a full exploration in the context of DNNs.

Another aspect to explore further is to construct a dataset specifically for test case prioritization and observe how it performs on real data. With the knowledge gained from this thesis, it seems feasible to spend an extra amount of effort on creating a dataset with high quality. In addition to using data from changes in production, it could be of value to perform a numerous amount of iterations to observe which test cases covers different files and add it to the current dataset. This is to gain more representation of each test case failure, which in turn would introduce diversity into the dataset, allowing for better predictions from a machine learning model.

Finally, exploring different test case selection methods could be of interest. In this thesis, a naive method and the knapsack problem is utilized and compared for test case selection. However, these methods are experimental, especially the knapsack problem in a test case selection context. To further analyze different test case selection methods, Graves et al. [10] mentions several test case selection techniques which can be a good source for inspiration.

Bibliography

[1] Ahmad Alwosheel, Sander van Cranenburgh, and Caspar G Chorus. Is your dataset big enough? sample size requirements when using artificial neural networks for discrete choice analysis. *Journal of choice modelling*, 28:167–182, 2018.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

[3] Paula Branco, Luís Torgo, and Rita P Ribeiro. Smogn: a pre-processing approach for imbalanced regression. In *First international workshop on learning with imbalanced domains: Theory and applications*, pages 36–50. PMLR, 2017.

[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[5] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: An industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 975–980, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2983954. URL `https://doi.org/10.1145/2950290.2983954`.

[6] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12:185–210, 2004.

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[8] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *stat*, 1050:10, 2014.

[9] Google. Advanced Course - GAN, 2022. URL `https://developers.google.com/machine-learning/gan`. [Visited on 2023-05-02].

[10] Todd L Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.

[11] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. *Advances in neural information processing systems*, 30, 2017.

[12] Mohamad H Hassoun et al. *Fundamentals of artificial neural networks*. MIT press, 1995.

[13] Hosney Jahan, Ziliang Feng, SM Mahmud, and Penglin Dong. Version specific test case prioritization approach based on artificial neural network. *Journal of Intelligent & Fuzzy Systems*, 36(6):6181–6194, 2019.

[14] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[15] Diganta Misra. Mish: A self regularized non-monotonic activation function. *arXiv preprint arXiv:1908.08681*, 2019.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[17] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43. IEEE, 1998.

[18] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[19] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 268–279. IEEE, 2015.

[20] Aizaz Sharif, Dusica Marijan, and Marius Liaaen. Deeporder: Deep learning for test case prioritization in continuous integration testing. *CoRR*, abs/2110.07443, 2021. URL `https://arxiv.org/abs/2110.07443`.

[21] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 12–22, 2017.

[22] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

[23] YData. YData Synthetic, 2023. URL `https://github.com/ydataai/ydata-synthetic`. [Visited on 2023-05-25].