university of groningen

faculty of science and engineering

Test selection, minimization and prioritization

Max Vincent Valk

September 1, 2022

university of groningen

faculty of science and engineering

University of Groningen

Test selection, minimization and prioritization

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of
Prof. dr. ir. G. Gaydadjiev
and
Msc. CEng MIET Lewis Binns, Msc. Harold Slegers

In collaboration with

ASML

Max Vincent Valk (s3246922)

September 1, 2022

Contents

Acknowledgments

I started the day with lots of problems. But now, after hours and hours of work, I have lots of problems in a spreadsheet.

Randall Munroe, XKCD 1906

I would like to thank professor dr. ir. Georgi Gaydadjiev for his ongoing guidance, support, and just critisisms during my thesis. Without his guidance, the final thesis would not have the form it has today. For providing a welcoming and open environment, in which any idea could be discussed and taken seriously, I would like to thank both Lewis Binns as well as Harold Slegers, and in general, the leveling department of ASML.

Of course, I would not be able to have finished my thesis, or let alone university, without the support of my loving parents. Even though the road that finally lead me to where I am today was long, with unusual twists and turns, they have supported me every step of the way. I want to thank them for the trust they have had in me, that no matter the path, the end would be all right.

I would like to thank my partner in life, Noemy Vergara Vera, for always being there for me. You helped me through the loneliest, most stressful periods of my thesis by always being there for me. The last two years have been challenging for the both of us, but I believe that we bring out the best in each other. Gracias por todo, y sé que vamos a continuar con eso en nuestra aventura bellísima juntos.

Lastly, I would like to thank Mr. Otter for being an excellent rubber ducky.

Abstract

A crucial component of the software development process is the performance of regression testing, which ensures that a piece of software remains functional under changes. However, for large or frequently changing software projects, the volume of required testing can outpace the resources and/or time available, resulting in the need for more efficient testing practices. In this work, we explored test runtime forecasting, outcome correlation and predictive test selection. We find that predictive test selection can also be efficiently used for test prioritization, outperforming several heuristics common in the literature. In addition, we propose improvements to predictive test selection by implementing an asymmetric loss function and pre-selection of tests based on historical runtimes, which lead to a test-time reduction of 31.1% whilst still maintaining a recall of above 0.9. We also found that runtime based forecasting at the level of test suites, rather than the traditionally used test cases, performed well and led to a reduction in test execution time of 24.3% whilst still preserving a precision of over 0.98. We have evaluated correlation-based minimization on the level of suites, and obtained a time save of 7.89% whilst maintaining a precision of 0.99%. Lastly, we propose methodological improvements to correlation-based minimization and runtime based prediction by suggesting the use of a validation set. Due to the nature of the dataset on which these models have been evaluated, it remains to be shown how these systems interact with regards to the time they can save in total, if deployed at the same time.

1 Introduction

In order to ensure stability of a piece of software it is vital that regression testing is performed when sections of a codebase are changed. Such tests are dedicated pieces of code meant to verify that the program still meets a series of requirements after alteration, and thus are executed after changes to the code, or on a periodic basis. However, such frequent testing gives rise to a variety of problems in larger software projects. The amount of resources and time needed to execute every test can become substantial, and even infeasible to do on a frequent basis. The solutions to this scaling problem fall in three categories: Test selection aims to identify tests that might fail from those that will be unaffected by (certain) changes, test minimization techniques identify tests that are redundant in order to reduce the run-time of all tests without affecting the quality of testing, and finally, test prioritization induces an ordering over all tests such that the chances of finding the most faults in the least time are maximized. Of note is the rise of machine learning approaches for these tasks, in particular, the works of Philip et al. [1] and Machalica et al. [2]. In this thesis, we will explore their approaches, in an attempt to improve and combine both state-of-the-art systems. Philip et al. introduce the FastLane system, which utilizes data-driven methods for both selection and minimization. In the Machalica et al. paper predictive test selection is presented, an approach which uses historical information about file changes and tests to perform selection. These systems have a number of favourable features in common. They can be re-trained frequently and automatically, allowing the system to stay up-to-date with its predictions. This does require running some tests on changes that have previously been labelled as either unecessary, or low-risk. However, this can be done during moments of relatively low load on the testing infrastructure, which will not interfere with the work of developers. Both systems can automatically do this process, making the use of such systems in a production environment feasible. Finally, both systems provide the administrator of such a system with a useful set of thresholds that can be adjusted to suit any degree of conservatiness with regards to the tests that should be run. This allows the system to be tuned to work well in any production environment. In this thesis, we will investigate these automated testing systems, attempt to improve on these methods, and combine them. FastLane offers a data-driven approach for test minimization, whilst predictive test selection offers a more sophisticated approach to test selection. Neither of these works address test prioritization, but the predictive test selection methodology at first glance appears to be very suitable for this task as it performs selection by assigning a risk score to each test case, which can be used as an ordering for a given change. The combination of these systems is expected to give rise to a potential system that will cover all three primary aspects for dealing with scaling issues in software testing. We will explore these approaches using the data of tests which have been performed in 2021 on the codebase of the ASML metrology department.

Moreover, we will suggest improvements to the methodology for both systems. One specific aspect that is not covered in both papers is the granularity at which test selection takes place. Test cases are often grouped together in sets called test suites (referred to as suites in this thesis), which cover the testing of a higher-level functionality in the program. Both aforementioned works only explore selection on the level of cases. We hypothesize that a selection on the level of suites could be more robust, that is, better at catching faults, than a selection at the case level. We envision that it may be easier to identify that a fault is likely to occur within an area of the code which is associated with a suite, than it is to identify which specific region within that area is resonsible for the fault. In order to further explore this idea, we will also consider another unit of organisation of tests that is not commonly used, but exists within the code organisation in ASML, the building block. Each building block has a set of suites associated with it, and corresponds to a full test of complex functionality.

Furthermore, in the paper on predictive test selection, a standard gradient boosting regressor is utilized. This approach uses a symmetrical loss, in which overestimating the risk of a test, and underestimating it, are both considered equally wrong insofar as the deviation from the correct answer is the same in absolute terms. The consequences of not running a test that would fail, and running a test that will pass, are quite different. A missed failing test could result in (temporarily) incorrect software, whereas running a passing test only has as a consequence the resources and time necessary for its execution. We hypothesize that we can achieve a better performance in terms of recall by taking this risk asymmetry into account during training by using an asymmetric loss function. Lastly, tests can vary wildly in duration, meaning that the required resources to execute such a test also vary drastically. As a result, two different methods for test selection that are equally good in test selection by quantity may vary in their performance on tests with differing durations. We envision that for tests with a shorter duration, it may not be worth it to predict an outcome, as the cost of execution associated with these tests is minimal, as compared to the cost of excluding failing tests. The latter will result in a delay of fault discovery, which can be a source of frustration for developers who have to go back to earlier work to resolve the issue, after which the test still needs to be executed. We will investigate how such a pre-selection based on runtime will affect the tradeoff between fault detection, and the time saved by applying test selection methods.

It should be noted that all of the aforementioned methods are *unsafe*, which has as a consequence that there exists a possibility of a test not being selected that would have failed on the code change for which we are selecting tests. In such methods, critical versions or sections of the program that is being developed still require a test with the full set of tests available. However, these methods still serve a purpose in that they can be used in earlier stages of testing, if they are capable of detecting most of the failing tests whilst providing a sufficient reduction in test execution time.

1.1 Research Questions

To summarize, this thesis focuses on the following research questions:

- Q1. Does a system combining predictive test selection, outcome correlation and runtime-based outcome prediction outperform either of the individual systems in terms of reduction in test execution time and fault detection capacibilities?
- Q2. Can predictive test selection be effectively used to perform prioritization?
- Q3. Will the introduction of different granularity levels in which the tests are grouped influence the performance of predictive test selection, outcome correlation and runtime-based outcome prediction?

For the first question, we will examine the approaches as proposed by Machalica et al. [2] and Philip et al. [1], which both adress different approaches to reducing the cost of running tests on testing infrastructure. In addition, we will attempt to improve the performance of both systems. In order to address all three topics common in the literature and to answer the second research question, we aim to assess the feasibility of using predictive test selection for prioritization. For the third question, we will explore how the recall-time saved tradeoff is influenced by varying at which level of hierarchical organization models are fitted over.

1.2 Thesis Outline

Chapter 2 introduces the works of Philip et al. [1] and Machalica et al. [2], as well as the literature on test selection, minimization and prioritization, and finally the machine learning techniques which have been utilized in this thesis. Readers familiar with these topics can safely skip these sections. Chapter 3 covers the creation of the dataset, as well as the variety of experiments that have been performed. Chapter 4 covers the results of these experiments. In Chapter 5 we will touch upon the implications of our findings. Finally in Chapter 6, we will summarize our findings as well as touch upon future work.

2 Background Literature

The techniques that are used to improve regression testing can in general be split up into three categories, namely selection, minimization and prioritization [3]. Test selection addresses the issue of finding which tests are relevant given a specific change, and therefore reduces the overall time spent on testing. Test minimization addresses the issue that some tests may monitor the behaviour of the software on similar requirements, and thus seeks to remove any tests that are redundant. The last category is test prioritization, which addresses the order in which the tests are performed. If a potential issue is present within the code, the time in which it is found is important, as the faster the developer receives feedback, the lower the risk of incurring a productivity penalty as a consequence of having to switch contexts between their new and old tasks [4, 5]. In addition, there may not be sufficient time to run all tests, even after an initial selection, and as such, prioritization seeks to maximize the errors found within a limited timeframe. In this chapter, we will first briefly introduce the systems as proposed by Philip et al. [1] and Machalica et al. [2] for adressing regression testing. Afterwards we will introduce a variety of approaches for selection, minimization, and prioritization, in order to provide an overview of the field. Lastly, we will provide a brief background to the machine learning techniques that are used for the methods that are utilized in this thesis.

2.1 FastLane and Predictive Test Selection

In the work of Philip et al. [1] the FastLane architecture is introduced, which reduces the amount of tests that are run in a three-step process. First, a classifier is trained which separates code changes that most likely are safe from those that need further testing. For this purpose features are extracted on the level of a commit, that is, when one or more files that have been altered, added or deleted. Examples of such features are the file types involved in the commit, the number of lines changed per file, and information with regards to the frequency of alteration of files in the commit over one, two, and six months, and since the creation of the file. A commit is deemed safe if, for example, it involves few alterations to file types that do not typically induce test failures, such as configuration files. Second, all tests that relate to the code that has been changed are selected. An initial set of tests is selected by considering for each program the set of all tests corresponding to it. This selection is then refined by inspecting tests that frequently run together, in order to identify any tests that correlate strongly in test outcome. For every set of correlating tests that is found, all but the test with the lowest average runtime are removed. Third, the remaining tests are then executed. The time a test takes to run is monitored, as there exist certain tests of which the runtime is bi-modally distributed. For such cases, the outcome of a test and its duration tend to be correlated. For example, the runs of a test that succeed swiftly typically pass, but those that fail tend to take longer. Such behaviour could be caused by failing tests that are waiting on time-outs to expire, among other reasons. For each test for which this behaviour is observed a threshold is determined via logistic regression, such that if a test execution reaches it, the outcome can be predicted with a high certainty. These tests are then aborted, and the prediction is used to assign the test outcome. With their system, Philip et al. perform minimization, by analyzing which tests correlate strongly in outcome, as well as a crude form of test-case selection by their identification of safe commits.

The work of Machalica [2] et al. takes a different approach which focuses on selecting the tests which are most likely to break. Their system, predictive test selection, utilizes only one model and analyses changes on a per-commit basis. Initially the set of all potentially relevant tests is selected by finding all files that relate to the changed code. Then, for each test t in the set of selected tests T and the commit C, the pair $\langle C, t \rangle$ is given to a regressor that assigns a score between 0 and 1, which can be interpreted as the risk of t failing due to the changes in C. After all such pairs have been scored, the tests which are associated with the highest scores are selected and actually tested. More specifically, the system always tests a percentage of the highest scoring tests, as well as any other test whose estimated risk surpassed a configurable threshold. This system only performs test case selection, in a more sophisticated manner than the approach chosen by Philip et al. [1].

2.2 Regression testing approaches

2.2.1 Selection

The goal of Regression Test Selection (RTS) is to find a subset of all tests available such that for a given set of changed files each test that could potentially identify an issue on the basis of those changes is included. An approach that does this consistently is deemed *safe* [6]. An important finding in this field is that there cannot exist any efficient procedure in selecting exactly this set [7]. Even the identification of test cases that are modificiation traversing, that is, tests that execute either new code, modified code, or used to execute deleted code, is an NP-hard problem [7]. Fortunately these stringent constraints hold only in the case of exactly selecting all tests that fufill these conditions. Therefore all approaches for test selection attempt to identify a superset of all modification-traversing tests.

Evaluation: The performance of RTS methods is primarily measured in *recall*. Let T denote the set of all available tests t for a particular version v of a Software Under Test (SUT), C the set of all changes in a software change to the SUT at version v, $S : \mathcal{P}(T) \times C \mapsto \mathcal{P}(T)$ a selection function, and $O : T \times C \mapsto \{True, False\}$ an oracle function such that $O(t,C) = True$ iff the set of changes C causes test case t to fail. A test $t \in T$ is said to be a true positive iff $O(t,C) = True$ and $t \in S(T,C)$, or, in other words, S correctly identifies t as a test affected by the code changes C. Then the set of all true positives can be defined as in Equation 1:

$$\mathrm{TP} = \{t \in T \mid O(t,C) \land t \in S(T,C)\} \tag{1}$$

A false negative on the other hand is a test case $t \in T$ that is not selected by S, but for which C causes t to fail. Then the set of all false negatives can be defined as in Equation 2:

$$\mathrm{FN} = \{t \in T \mid O(t,C) \land t \notin S(T,C)\} \tag{2}$$

Then recall can be defined as in Equation 3.

$$\mathrm{recall} = \frac{|TP|}{|TP| + |FN|} \tag{3}$$

The reason recall is used as the primary measure of performance is because including a test that will pass after execution is not as severe as excluding one that fails. Ideally we would like the set of selected tests to be as small as possible, so precision is concidered, but it is secondary to recall. To formally define precision, we first define the set of all false positives, or tests that are selected by S but for which there is no test $t \in T$ such that $O(t,C) = \mathit{True}$, as in Equation 4:

$$\mathrm{FP} = \{t \in T \mid \neg O(t,C) \wedge t \in S(T,C)\} \tag{4}$$

Then precision is given as in Equation 5.

$$\text{precision} = \frac{|TP|}{|TP| + |FP|} \tag{5}$$

Unfortunately, whilst recall and accuracy are frequently reported, it is difficult to make any meaningful comparisons between techniques in the literature. This is because there exist few commonly agreed benchmarks for the field [8]. The performance of a system should only be seen as indicative of one specific application. Due to this issue some authors compare their technique to random selections of tests [8].

Taxonomy: Methods for RTS can be subdivided into static and dynamic methods. Static methods aim to identify which tests are relevant by examining the structure of the project and the code itself [9]. The relationships between files are analysed and can be used in combination with information about which files have been changed in order to identify the set of tests that relates (in)directly to said change, or in other words, a superset of the modification-traversing tests. An example of such a method is firewall [10], which selects tests on the level of modules of code. In this approach, all modules that either have changed or use the functionality of changed modules have their corresponding tests selected. Other static methods examine such a structure at the level of classes or functions, such as class firewall [11, 12] and control call graph methods [13]. In more recent work Correia and Santos [14] introduced MOTSD, a static method that extracts code coverage of tests on the instruction-level and approached selection as a multi-objective task. They report a subpar performance, obtaining precision scores in the range of 0% to 1% and recall within 21% to 26%. On the other hand, dynamic methods analyse which procedures or files are invoked during the execution of tests on the previous version of the program, and deduce the relevant tests using that information. While there exist both static and dynamic methods with varying degrees of overhead, it should be noted that the overhead introduced by dynamic methods is on-line, meaning that additional resources are needed during testing to monitor which files are involved during a test execution. This has as a consequence that such methods do incur a speed penalty when executing tests. Nevertheless, this penalty has been found to not outweigh the benefits of test selection. A recent example of a dynamic method that has seen real-world adoption[1] is EKSTAZI, an approach that selects which tests to include on the basis of which files have been accessed by each test on previous executions [15, 16]. The relevant tests are selected by considering each binary file which has been changed in a newer version. The authors attempt to address the overhead of collecting runtime information by suggesting two passes, of which one does not collect new dependency information. The other can be run in parallel or at a later point in time.

[1]In open source projects of the Apache software foundation, `https://www.apache.org/` [15]

As aformentioned, another defining factor between approaches is the coarseness with which it analyses the relationship between the program and the tests is examined. Fine-grained methods typically examine how specific functions, basic blocks or lines of code relate to tests, whereas coarse methods examine this relationship on the level of files or classes. Fine-grained methods should theoretically be able to capture a smaller subset of all the available tests than coarse methods, however in practice the difference in test selection is negligible [6, 9]. In addition, fine-grained methods analyze code in more detail, which has as a consequence that the overhead necessary tends to be larger than in the case of coarse ones.

A final property of such test selection methods is the level at which they need to examine the behaviour of the program that is tested. Many methods that have been formulated (including the aformentioned control graph methods, and EKSTAZI) require language-specific or domain-specific solutions and require access to the code of the program, before any modifications have been made. This has implications with regards to the generalizability of such systems, as when one of these methods is employed it may not be the case that it is trivial to adapt it to a new programming language or to certain features within a language such as pointers and type coercion (see for instance [17]). In contrast, black-box methods only need to examine the input-output behaviour of the software on a set of tests, which can be reasonably expected to be present for any possible programming language. Machine learning approaches to regression test selection tend to fall in this category. Of particular interest is the study of Martins et al. [18], who explored the use of a variety of machine learning classifiers, and found that the best results were obtained by using either a random forest, or a logistic regression. Their methods are similar to Machalica et al. [2], in that they both utilize machine learning techniques for test selection, but they differ in that the former trains a direct classifier, and the latter first trains a regressor resulting in scores for each test, which are then selected over. In addition, Martins et al. did not consider the technique used by Machalica et al., gradient boosting, which may be more suitable for regression test selection as it performs well on datasets where there is an imbalance between samples of different classes. In the work by Memon et al. [19] it was found that the ratio between passing and failing tests was as skewed as 99:1. This imbalance was addressed in [18] by using class weights and both over- and undersampling during the process of fitting their models.

It should be noted that the initial test selection as performed in the systems of Machalica et al. [2] and Philips et al. [1] are related to the firewall method [10] in that the tests get selected on the basis of involved modules and build dependencies, excluding tests that do not directly relate to the changed files themselves. This approach should suffice in practice, but depending on the structure of the test suite, may actually be unsafe as it can be the case that there exists a test that reveals a fault in the software outside this selection. This can occur if the set of tests that has been selected for a particular firewall is unreliable. That is, there exists an input-output pair for which a test is fault-revealing, but it is not evaluated within the set of selected tests. If there also exists a test outside of the selected tests that exercises parts of the changed code with the input-output pair that reveals the fault, an unsafe selection of tests can occur [6].

2.2.2 Minimization

Test minimization, also known as reduction or filtering, is the process of identifying redundant tests in order to remove them. The key difference between minimization and selection is that in minimization, we identify tests that are unnecessary for any possible change, given the rest of the test suite, whereas in prioritization we only aim to filter out tests that are unnecessary given the set of tests and a specific change. Minimization is often formalized in terms of set of requirements R, $\{r_1, r_2, ..., r_n\}$ that must be fufilled for the SUT to be considered correct [3]. Each test t in the set of all tests T covers a set of requirements. Let $C : T \mapsto \mathcal{P}(R)$ denote a function that returns the set of requirements covered for any given test t. For a set of tests to be considered correct, it must hold that all requirements are covered by at least one test, or formally, $\forall_{r \in R} \exists_{t \in T} r \in C(t)$. Let $M : \mathcal{P}(T) \times \mathcal{P}(R) \mapsto \{True, False\}$ denote a function that is true iff for a given set of tests and requirements it holds that all requirements are covered by that set. The goal of a test minimization technique is then to find the minimal hitting set of tests over the requirements, which is the smallest set of tests such that all requirements are covered, or formally, we wish to find a set $T_M \in \mathcal{P}(T)$ such that $M(T_M) \wedge \forall_{T_O \in \mathcal{P}(T)} (M(T_O) \to |T_M| \leq |T_O|)$. Techniques for test minimization vary on the basis of how the set of requirements are determined, as well as how the minimal hitting set is approximated.

Evaluation: The performance of test suite minimization is often evaluated in terms of both the Percentage of Test Suite Reduction (PTSR), as well as the fault detection capability of the reduced set of tests [20]. These measures can also be combined with the Percentage of COVerage (PCOV) of the original set by the reduced set of tests in order to make comparisons between methods, where the goal is to maintain a high requirements coverage with the smallest set possible. Finally, evaluating these techniques only on set minimization does not take into account that the underlying goal of these methods is to reduce the time spent testing unnecessarily. To adress this the Percentage of Test Time Reduction (PTTR) can also be calculated as the ratio between the time spent testing the reduced set versus the total time needed to execute all tests. To obtain these metrics, programs that have known or seeded faults are typically evaluated. Unfortunately, for a minimal hitting set over the requirements to also still detect all faults requires that the requirements are entirely correct to begin with. This cannot be guaranteed by all techniques, and thus there is the possibility of filtering out tests that can identify a fault the remaining tests cannot [21]. As with RTS, there exists no commonly agreed upon benchmark for minimization, which makes comparisons between techniques on the basis of papers difficult [20], and the same precautions apply on the comparisons of techniques that have been evaluated on different code bases.

Taxonomy: The set of requirements can be established in a variety of ways. Some techniques simply assume the requirements that each test needs to fufill are already known. This is often an oversimplification which is applied in order to evaluate the ability of an approach to reduce the test set, as it may be infeasible to guarantee this in the context of real-world code bases that are subject to frequent changes. Furthermore, this limits the applicability of these techniques to code bases where such requirement tracking is not yet in place. Other approaches base the requirements on a metric of code coverage, where each requirement is one line of code, a method call, or another level of granularity. A suite minimization is then considered correct if each element that was covered by the full set of tests is also covered by the minimized set. A finer granularity of the coverage metric can result in a higher fault-detection capability [22], but this is not always the case [23]. Similarly coverage can also be determined based on the coverage of test cases of a model of the software rather than the code itself [24]. The requirements can also be derived from dynamically obtained execution profiles of the SUT such as in the work of Smith et al. [25]. It should be noted that, depending on how the requirements are determined, the resulting reduced test set may detect less faults than the original set [21].

Unfortunately, the minimal hitting set problem is NP-hard, and therefore the techniques used in minimization either have poor time-complexity (such as linear integer programming), or rely on approximations (such as genetic algorithms or heuristics) [20]. The heuristics that are frequently used rely on different greedy methods to extend a current candidate set of tests. A less common method for approximating a minimal hitting set, is clustering. In these methods, tests are clustered on the basis of each requirement they cover, and then from each cluster, a limited numbers of tests is selected, and the rest is discarded [26, 27]. This can come at the cost of discarding tests that detect faults, but are similar to (in the same cluster as) other tests that miss those faults.

As a notable exception to common approaches that seek to identify whole test cases that are redundant, the technique proposed by Vahabzadeh et al. [28] performs minimization by considering redundant statements in different cases . Test cases whose code is similar is then unified into a single test that performs the behaviour of all separate cases.

2.2.3 Prioritization

In environments where testing occurs frequently such as projects utilizing Continuous Integration [29], the computational power and time available may not be succifient to execute each test that has been selected by an RTS method or otherwise [30]. The topic of prioritization aims to deal with this issue by deducing an optimal ordering of the test cases such that an objective is maximized [3]. Typically these methods aim for an ordering such that the test cases that are executed first have a higher probability of failure. Such an ordering would also reduce the time between code submission and the time at which the first failure is found, and therefore provides quicker feedback to developers. The problem of prioritization is closely related to RTS in that a total ordering TO can be treated as a selection, if we select the head of such an ordering up to some $n \in \mathbb{N}$ s.t. $n \leq |TO|$.

Evaluation: The methods used for evaluating prioritization techniques are more involved than in the case of selection or minimization due to the added complexity of having to evaluate an ordering. A commonly used metric is the Average Percentage of Faults Detected (APFD) and its derivatives such as the Normalized APFD (NAPFD). The APFD metric is the area-under-the-curve of the total number of faults detected plotted against the fraction of the tests that have been included [31]. For example, assume we have a set of tests $\{A,B,C,D\}$ which we apply to a program with two faults, such that A detects one fault and B the other. Two possible orderings are $\langle A,B,C,D\rangle$ and $\langle C,A,D,B\rangle$, of which the first finds both faults within two tests, and the second needs to run all tests to find all faults. This is reflected in their APFD (see Figure 1a and Figure 1b), or the area-under-the-curve of the plot of the fraction of included tests as plotted against the percentage of faults detected. It can be calculated with Equation 6, in which m is the number of faults in the program, n is the number of tests, and TF_i is the minimum number of tests that has to be included in order to catch i faults [32].

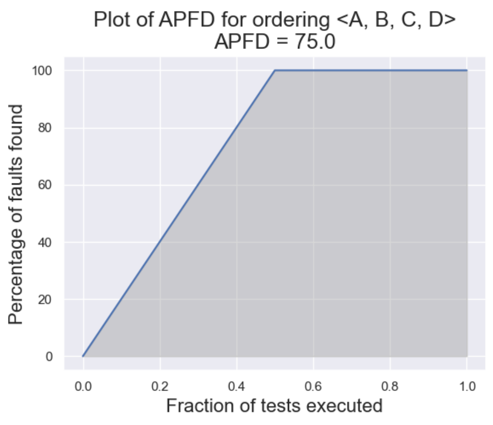
$$\mathrm{APFD} = 1 - \frac{\sum_{i=1}^{m} \mathrm{TF}_i}{nm} + \frac{1}{2n} \tag{6}$$

This metric assumes that it is possible to run any ordering of tests completely, but in settings where time is limited, not all tests may be considered in the final ordering. For such scenarios the NAPFD metric is utilized, which also considers the amount of tests that actually could have been executed under some constraints. Finally, the APFD metric treats all faults as equally severe, and all tests as equally costly. The APFDc (cost-cognizant APFD) formulation addresses this by modifying the original APFD formula to account for both [33]. Instead of computing the area-under-the-curve of the plot of fraction of tests included against the percentage of faults found, the new metric calculates the area-under-the-curve of the plot of the fraction of test *time* executed over the sum of all test times against the fraction of fault severities found over the sum of all fault severities. It can be calculated as in Equation 7.

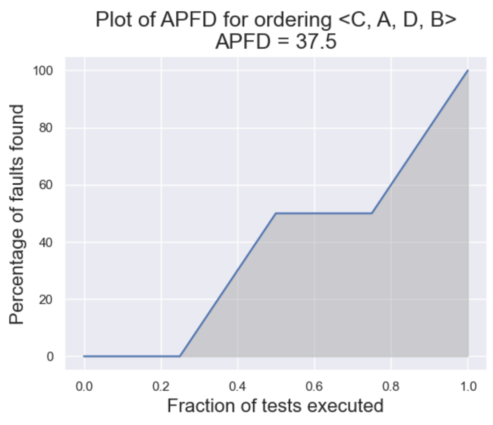
$$\text{APFDc} = 1 - \frac{\sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=\text{TF}_i}^{n} t_j - \frac{1}{2} t_{\text{TF}_i}\right)\right)}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{7}$$

Unfortunately these metrics are only easy to calculate on datasets where the faults are known, as the metrics consider faults and not failing tests. There are several approaches to deal with this mapping. Datasets where all faults are known, such as DEFECTS4J [34], can be used (as in Paterson et al. [35]), faults can be introduced by creating mutants [36], or by seeding faults by hand [37]. The latter two methods create artificial faults which may not be representative of real-world faults, or need a manual process to create them, thus reducing the size of datasets used in such studies due to the effort required. The first approach comes closer to a realistic evaluation, but still diverts from real-world datasets in that the data has been cleaned and carefully selected prior to publication. If one wishes to evaluate the performance on real, possibly complex software suites, to more accurately gauge the real-world performance of approaches, some assumptions need to be made. Specifically, some studies consider the two extremes that can easily be evaluated, namely that each failed test corresponds to one fault, or that one fault is responsible for each failed test (see for instance [38]). Both of these approaches are oversimplifications, as a single fault could cause multiple, but not all, tests to fail, or a single failed test can be due to multiple faults [39]. In addition, the APFD metric and its variants can only increase by detection of a number of distinct faults, but it also may be desirable for a method to detect a single fault multiple times. As developers only obtain feedback in the form of failed tests, different tests failing due to the same fault can help with localization [32]. When prioritization techniques are used in a situation where not all faults can be known, other metrics are used, such as evaluating how many tests in an ordering must be executed before a failing test is found (or the time required to do so). As with RTS and minimization, few benchmark datasets exist [40], but there are a number of heuristic-based ordering techniques that can be used in order to estimate the performance of a new method. Techniques are often compared against random orderings, as well as orderings simply based on the frequency of failure or execution time [32, 38]. Whilst it has been shown that the performance of techniques can differ drastically depending on what dataset it is evaluated on, in most cases performance improves as compared to these heuristic-based orderings [32].

Taxonomy: The approaches for obtaining such an ordering can be grouped into point-wise, pair-wise and list-wise methods [40]. Point-wise methods induce an ordering by first assigning a score to each case in isolation, by which each test case can be ordered to obtain $O(T_s)$. An example of a point-wise method is *G-clef*, which uses simple features such as frequency of revisions, that are then decayed over time [35]. These are used to score the risk of failure of each changed file, which then can induce an ordering over the tests related to these classes. Pair-wise methods aim to learn a partial ordering on test cases which can be used to obtain a total order-



(a) $\langle A,B,C,D \rangle$



(b) $\langle C,A,D,B \rangle$

Figure 1: An example of how APFD is calculated, on a set of tests $\{A,B,C,D\}$ applied to a program with two faults, and A and B both detect a different fault.

ing, and finally, list-wise methods obtain an ordering of test cases directly by considering each selected test together.

Prioritization is not solely done on the estimated probability of test case failure. As this technique is often utilized in a time-sensitive context, approaches have been formulated that also take the time for a test to execute into consideration [30]. Such methods are known as *cost-aware*, and are typically evaluated with APFDc [30, 38]. In contrast, *cost-unaware* methods avoid this issue by assuming that each test case will have an identical cost, for example, an identical execution time.

Unlike in RTS, it is not necessary to know what has been changed in the SUT for prioritization. It is possible to obtain an ordering which outperforms heuristic based orderings by only considering the test cases themselves. These methods are known as *change-unaware*. Such an ordering can be obtained by considering features of the test, for example, the frequency of which a test has failed prior, as well as features that are obtained either statically or dynamically, such as code coverage [38]. Methods that use several features based on the history of a test are common in the field [30]. It is possible for a prioritization technique to be both change-unaware and history-unaware. In such techniques the goal is to obtain a *general*, as opposed to *version-specific* ordering, that maximizes the probability of finding faults earlier without any context. This can be treated as a starting order for a new set of tests about which no historical data is available [32].

Recently there has been an increase of interest in utilizing Machine Learning techniques for prioritization [30, 40]. Methods utilizing supervised learning aim to train systems that perform point-wise or pair-wise rankings based on historical data [41], and general information available about tests such as natural language descriptions [37], among others. These approaches have been shown to obtain good results, but require frequent retraining in order to avoid drift and are less flexible in adapting to an environment which is changing rapidly. Because of this, approaches based on reinforcement learning have been explored [40]. It should be noted that it has been found that methods employing supervised learning can perform just as well as reinforcement learning [42], and whilst reinforcement learning techniques perform well on average, the performance as considered per testing cycle can be quite noisy [39]. This makes it difficult to estimate how long test execution will take before submitting code to a testing architecture. Despite this, such systems have seen succesful real-world implementations [43]. Unsupervised methods aim to identify groups of similar tests via clustering. For example, Arafeen and Do [36] cluster tests on the basis of similarities in their requirements, and Carlson et al. [44] cluster using code coverage, code complexity and historical data. After clustering, tests can be ordered such that tests that are executed one after another come from different clusters.

2.3 Machine Learning approaches

In this section, we will provide a brief overview of the machine learning techniques which have been utilized for this thesis. Readers who are familiar with these techniquescan safely skip this section.

2.3.1 Logistic Regression

A logistic regression is a statistical model which is used in classification problems [45]. In the case of binary classification fitted on a dataset with n features x_1 to x_n (independent variables), an equation of the form:

$$p(x_1,...,x_n) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+...+\beta_n x_n)}} \tag{8}$$

is fitted by finding values of the parameters β_0 through β_n through maximum likelihood estimation. This function calculates the estimated probability of a datapoint belonging to one of the two categories. Values above 0.5 probability are interpreted as a datapoint belonging to one class, whereas values below 0.5 belong to the other. After a logistic regression has been fitted, it can then be used for predicting the class of new datapoints by calculating the value of this function on each point.

2.3.2 Decision trees

A decision tree [46] is a machine learning method which can be used for both classification and regression. It is structured as a directed, acyclic graph, consisting of three types of vertices (also known as nodes): a root vertex, leaf vertices and intermediate vertices. These vertices are connected such that each non-leaf node has at least two edges leading to other vertices. Each tree contains one root node, from which each other node is reachable. The edges leading out of one vertex are all associated with one feature of the dataset. The feature is split into as many intervals, or sets, as there are edges leading out of the related vertex. Each leaf node is associated with an outcome, which is a class label in the case of a classifier, and a numerical value in the case of a regression. A datapoint can be assigned such an outcome by starting at the root node, and for each non-leaf vertex, following the edge to the next vertex such that the value of the relevant feature of the datapoint is within the set or range of values associated with that particular edge. In Figure 2, we see an example of a simple decision tree for test selection. Starting from the root, the tree predicts a test might fail (and thus needs to be included) if the number of executions of that test is below five. If it is equal to or greater than five, then the average pass rate of the last fifty-six days is examined in order to decide whether or not it should be tested.

Decision trees can be created manually, or inferred from data by recursively separating the data at a node such that some measurement of impurity is minimized. Common measurements of impurity for fitting decision trees on classification problems are entropy (Equation 9) and the gini index (Equation 10). In both equations, C represents the number of outcome classes at a vertex v, and p_i gives the probability of a datapoint at a vertex belonging to class i.

$$E(v) = \Sigma_C^{i=1} - p_i \log_2 p_i \tag{9}$$

$$G(v) = 1 - \Sigma_C^{i=1}(p_i)^2 \tag{10}$$

These are then calculated for each child vertex $v_c \in S$ that is the result of splitting the data at the parent vertex v_p, and averaged over while taking the probability of ending up at each child node

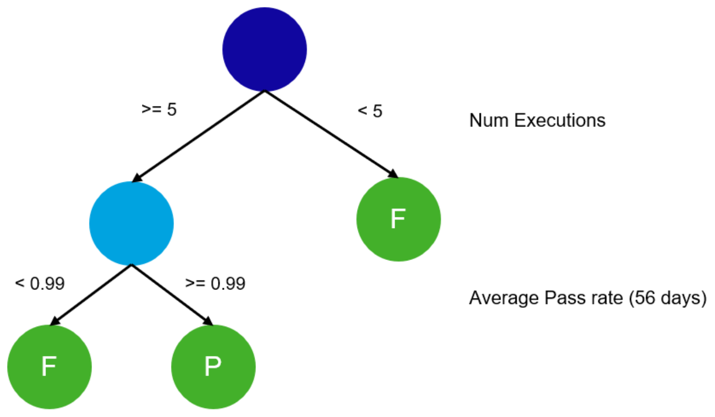


Figure 2: An example of a simple decision tree for test selection.

into account. Finally, we substract this from the entropy present at the parent node to obtain the information gain (IG) of a split, where a higher information gain means a better split:

$$\mathrm{IG} = E(v_p) - \Sigma_{v_c \in S} P(v_c) E(v_c) \tag{11}$$

Impurity is calculated differently in the case of regression. For instance, one can use the variance of the dependent variable in each node. Of particular interest is the Friedman mean squared error [47], which is defined as in Equation 12. The function i^2 is the least-squares improvement criterion of a set of datapoints at a node R, considering a binary split such that the resulting subsets are R_l and R_r, with $\bar{y}_l$ and $\bar{y}_r$ being the means of the dependent variable in the corresponding subsets. Finally, w_l and w_r represent the sum of weights associated with the samples in each subset. If the samples are unweighted, all weights are set to 1. The best split out of all possible splits at a node is one where the improvement criterion is found to be the largest.

$$i^2(R_l, R_r) = \frac{w_l w_r}{w_l + w_r} (\bar{y}_l - \bar{y}_r)^2 \tag{12}$$

When a certain condition is reached at a child node, such as a maximum depth being reached, or the number of datapoints remaining at the child node being below a threshold, the node gets turned into a leaf node. In the case of classification, the leaf is assigned the class of the most prevalent class in the data that gets classified at that leaf during training. In the case of regression, the average value of the dependent variable of the datapoints that end up in the leaf during training gets computed. How many children a parent vertex can have, as well as the maximum depth a tree can have, are often limited in order to attempt to combat overfitting.

2.3.3 Gradient boosting

Gradient boosting [47] is a type of ensemble method, a method which utilizes a combination of weak machine learning models in unison. By combining many weak learners, the performance of the ensemble as a whole exceeds the performance of any individual learner, if the types of errors each learner makes is independent with regards to the other learners. Each weak learner is a simple machine learning model, typically a decision tree, in which case the ensemble is also referred to as a decision forest. These weak learners are constructed in a sequential fashion, such that each model that is added corrects for the errors made by the ensemble prior to adding the new model. In each iteration of boosting, the performance of the current ensemble is evaluated over the training dataset. For each training datapoint, a loss function is used in order to evaluate the difference between the prediction for each datapoint, and the actual answer. The derivative of this loss function with respect to the answer as previously predicted by the model is calculated, which is known as the pseudo-residual, and a new model is constructed that tries to predict this pseudo-residual. Finally, this new model is added to the ensemble, and the new output of model is equal to the output of the ensemble before the new model, plus the output of the new model scaled by a learning rate. For regression, the squared difference between the predicted answers F and actual answers y is typically used:

$$L(y,F) = \frac{1}{2}(y-F)^2 \tag{13}$$

The output of each leaf node is equal to the value γ which minimizes the loss between the correct output of the dependent variable y, and the sum of γ and the prediction given by the model which has been fitted thusfar on the samples R that have been assigned to the leaf:

$$\underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R} L(y_i, F_{m-1}(x_i) + \gamma) \tag{14}$$

3 Methods

In this chapter, we will describe the experimental setup of each experiment conducted. The outcomes of these experiments are presented in the next chapter. In Experiment I, we recreate and improve parts of the methodology of Philip et al. [1]. In Experiment Ia and Ib, we explore runtime based thresholding and correlation based minimization respectively. We then investigate the methodology of Machalica et al. [2], by first performing selection in Experiment IIa. In Experiment IIb, IIc and IId, we explore filtering out tests from the data based on historical execution time, using an asymmetrical loss and a combination of these two techniques. Finally, in Experiment IIe we evaluate the best performing model of the second experiment on the testing data.

3.1 Experiment I

In the work of Philip et al. [1], three systems are introduced. In this section, we will first introduce the data which was used for our experimentation, and then both recreate and improve upon two of these systems, runtime based thresholding and minimization by examining the similarity in outcomes.

3.1.1 Object of analysis

At ASML, for every test that is executed a record containing information about that run such as test duration and outcome is stored in the Test Results Database (TRD). Whenever an engineer submits their program for testing, they do so by also specifying a test plan, which is composed of a set of suites, each of which incorporates a set of tests. Furthermore, each suite is associated with a building block, a unit of organisation used in ASML to group similar functionality together. Due to the large software testing volume, this study limits itself to the analysis of the testing as performed by the metrology department. Unfortunately, there are no unique identifiers stored in the database for suites or cases. Instead, an identifier is constructed by computing a hash over other information that is stored. For suites, this hash is computed by combining a building-block identifier, the type of machine the test is for, and the names of the files specifying configuration data for the test. For cases, this is constructed by computing a hash over the suite hash, the test name, and the test description. Each suite has a total of four possible outcomes: ’PASS’, ’FAIL’, ’UNRESOLVED’ and ’UNTESTED’. A suite is assigned a passing status if all of the tests in that suite execution have passed and a failing status if one or more tests failed. The unresolved status is assigned to a suite if there was an issue during the setup of the testing environment, or if any test case within the suite has not completed before a specified timeout. If any test case has timed out, the remaining test cases within a suite are still executed, as the timeout is considered on a per-case basis. This timeout can be adjusted by the developers, and has a default value of 30 minutes. Finally, the ’UNTESTED’ status is given to suites that were skipped for execution, which is done manually by developers before submitting a test plan. Test cases have an additional possible outcome status, ’UNSUPPORTED’, given when an execution is attempted of a test on incompatible software.

3.1.2 Experiment Ia: Thresholding

For the first experiment we analyzed the suitability of part of the methodology as introduced by Philip et al. [1], which introduced the idea of identifying for which tests a threshold can be defined such that if a test executes for a longer time than the threshold, one can predict the outcome with a high degree of certainty. We extracted each record in the database from 2021 belonging to the metrology department at the level of both suites and test cases. From this record we obtain the hash identifying the test, the date, outcome and duration of that run. This resulted in data for 37,684 unique cases, and 17,948 suites, associated with 17,525,006 and 8,885,769 executions respectively. Records that had an outcome of 'UNTESTED' or 'UNSUPPORTED' were discarded from the data, resulting in 79,472 discarded case records and 53,177 discarded suite records. In addition, a dataset was constructed for selection on the level of building blocks, using the suite dataset. For each plan, and for each building block within that plan, all suites were selected. The start of the execution of the building block for that plan was set to the earliest start of all corresponding suites, and the ending time was set to the latest of these times. A building block was considered passing if all the suites that related to that building block for a single plan succeeded, and failing otherwise.

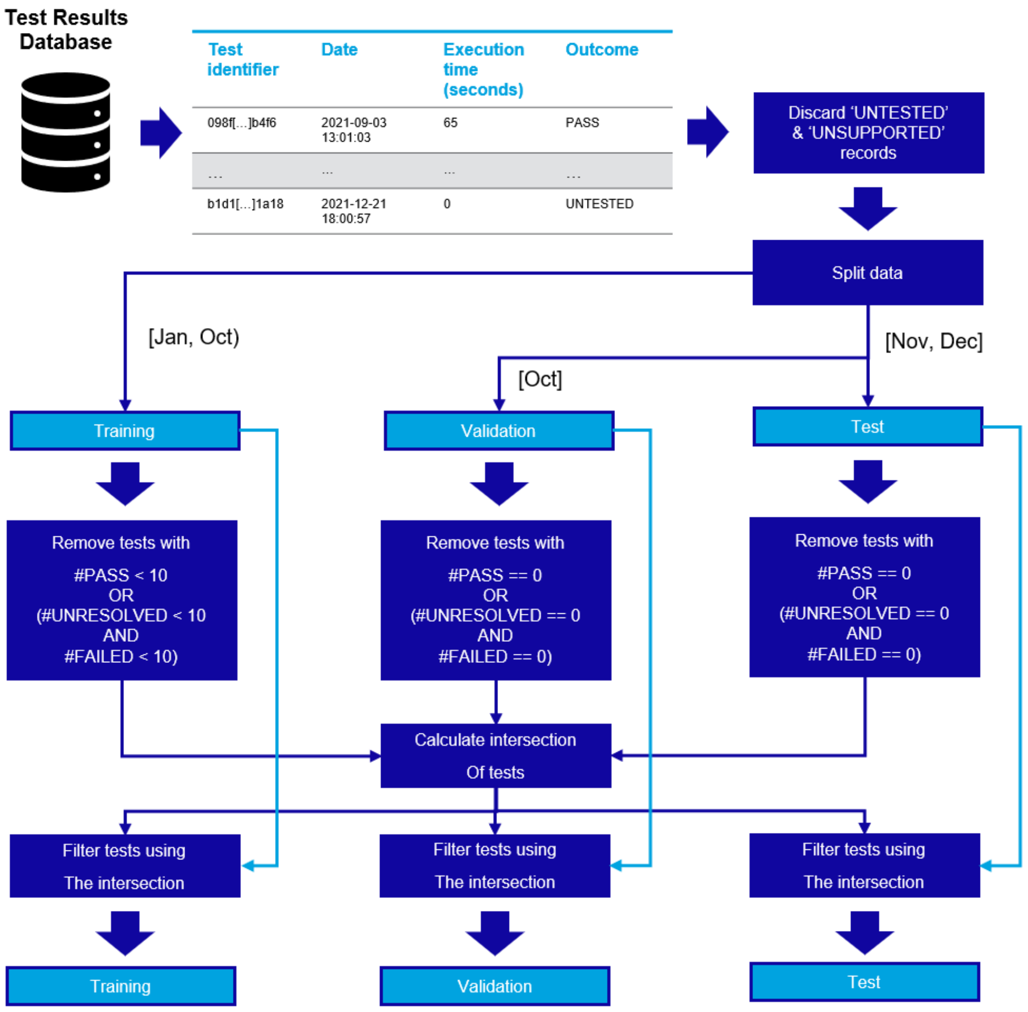

Test Results Database
Test identifier
Date
Execution time (seconds)
Outcome
098f[...]b4f6
2021-09-03 13:01:03
65
PASS
...
b1d1[...]1a18
2021-12-21 18:00:57
0
UNTESTED
Discard 'UNTESTED' & 'UNSUPPORTED' records
Split data
[Jan, Oct)
[Oct]
[Nov, Dec]
Training
Validation
Test
Remove tests with #PASS < 10 OR (#UNRESOLVED < 10 AND #FAILED < 10)
Remove tests with #PASS == 0 OR (#UNRESOLVED == 0 AND #FAILED == 0)
Remove tests with #PASS == 0 OR (#UNRESOLVED == 0 AND #FAILED == 0)
Calculate intersection Of tests
Filter tests using The intersection
Filter tests using The intersection
Filter tests using The intersection
Training
Validation
Test

Figure 3: The pipeline for the creation of the datasets for Experiment Ia.

For all three datasets, we perform the following operations individually: The dataset is split into a training, validation and testing dataset such that all records from January until October are included in the training set, October in the validation set, and the remaining records are included in the testing set. We then find all tests that have either less than 10 passing executions or less than 10 failing or 10 unresolved executions. These are then removed from the training data, as the number of executions associated with these tests is considered too low in order to use for prediction [1]. For the validation and training data we perform the same selection, but instead checking if at least one passing and one failing or one unresolved execution is present in the data. Finally, we only keep the tests for which there is at least one record in each individual dataset (training, validation and testing) and discard the rest. This pipeline is illustrated in Figure 3. It should be noted that the inclusion of a validation set deviates from the original implementation as suggested in [1]. More precisely, for each test in the training set, we attempt to fit two logistic regression models: One which attempts to separate the times associated with passing executions and failing executions, and the second which attempts to separate the times associated with passing executions and unresolved executions. We discard a model if it is the case that each point after the decision boundary (that is to say, after the threshold time) is predicted as passing. If a test has no datapoints for one of these categories (either no failing, or unresolved datapoints), then we fit only one model. If two models have been fitted, for each such test we choose one model by examining the decision boundary for both and selecting the model for which this boundary is later. We evaluate each model on the datapoints found in the validation set, and discard any model on the basis of two criteria. The first criterion is the fraction of incorrect terminations with respect to correct terminations, in which we consider a test execution terminated if the execution time of the run exceeds the decision boundary. A termination is correct if the terminated test would have been non-passing without termination. The second criterion is the fraction of correct terminations, but with a wrong label, over all terminations. A termination is considered correctly labelled if a terminated test execution is assigned the correct label (failing or unresolved), and incorrect if this is not the case (assigning failing to an unresolved test or vice versa). In our experiments, we considered two settings: Selecting models that had the correct outcome on termination, as well as selecting models that terminated correctly, without regard for the label. This pipeline is visualized in Figure 4. The models which meet the criteria are then finally evaluated on the testing dataset. For the sake of comparison, we also evaluated this exact same process, but after substitution of the validation set with the training set. With this setup, the performance of the fitted model is evaluated on the data it is fitted on, which is the approach originally used by Philip et al. [1].

3.1.3 Experiment Ib: Similarity of outcomes

We continued the assessment of the methodology as proposed by Philip et al. [1] by examining the similarity of outcomes between suites and building blocks that were executed together. Due to the involved computational complexity and quantity of data available, correlation at the case level was not considered. We examined the same records as in Experiment Ia, using the same time periods for the testing, training and validation set. In addition, we also retrieve the plan associated with each test execution. For each possible pair of tests in the training data, we examine how often they are executed in the same plan, how often such an execution results in the same outcome, and how often it results in a different outcome. From these pairs we discard all that have not been executed a minimum of 50 times together, those that have not passed together a minimum of 20 times, and those that have not failed together a minimum of 20 times or have not resulted in unresolved together a minimum of 20 times. For each remaining pair, we calculate the fraction of identical outcomes over the total number of times the tests have been executed together. We select all pairs for which this value is at least 0.99. We then find which of these pairs have also been executed at least once in the validation data, and filter out any pair for which this value drops below 0.99 on the validation set. For the remaining pairs, we calculate

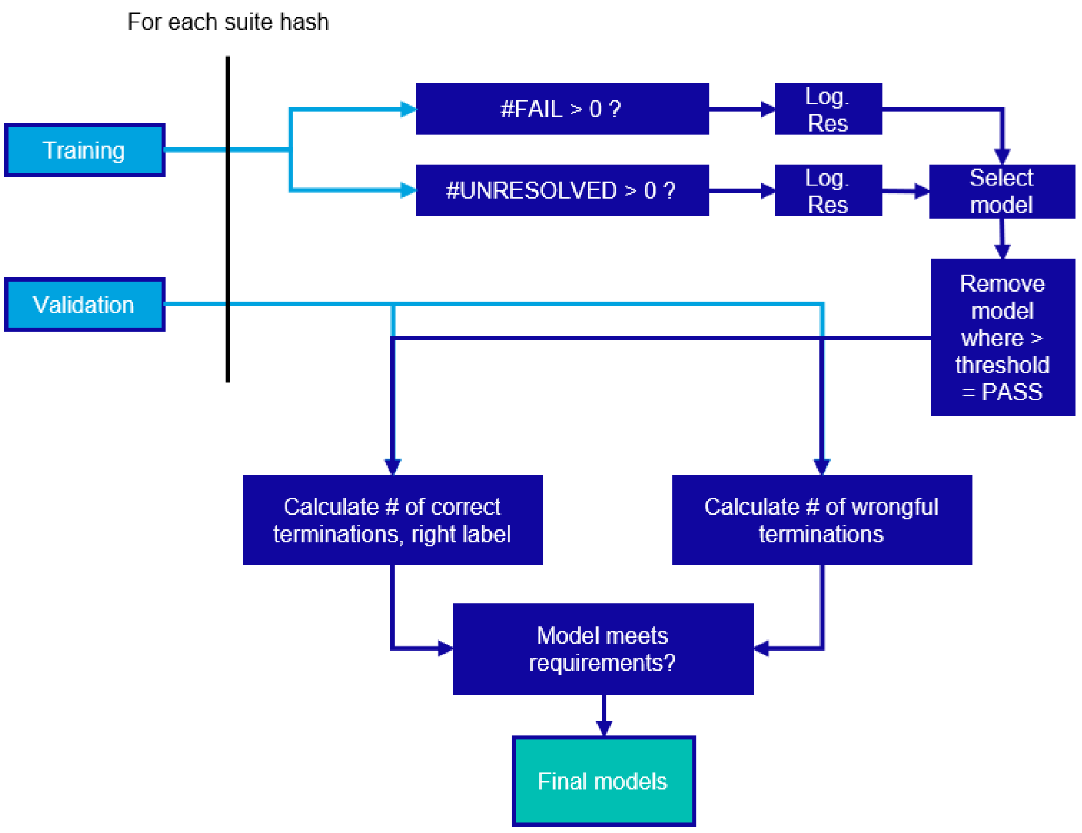

For each suite hash
Training
Validation
#FAIL > 0 ?
#UNRESOLVED > 0 ?
Log. Res
Log. Res
Select model
Remove model where > threshold = PASS
Calculate # of correct terminations, right label
Calculate # of wrongful terminations
Model meets requirements?
Final models

Figure 4: The experiment pipeline for Experiment Ia.

the mean execution time of both tests in the training data. Finally, from each pair we learn a rule such that if both are being executed together, the test with the shortest mean execution time in the training data is used to predict the test with the longest. These rules are then evaluated over the testing data. Finally, as with the previous experiment, we also conduct an experiment without validation set, which is identical in setup to the experiment as performed by Philip et al. [1], in order to compare approaches. In their original experiment and our recreation, the validation data is included as part of the training data, and no filtering is performed on the rules which have been learned from the training set.

3.2 Experiment II

In our subsequent experiments we explored the approach utilized by Machalica et al. [2] by exploring a variety of potential improvements. We train a model that, given a set of tests, returns a susbet of these tests that are the most likely to fail. In line with their work, we use a gradient boosting regressor for this task that, given information about a test and the code it was performed upon, assigns it a risk score which can be interpreted as the risk of that specific test failing if it were to be executed. When we perform a test selection on a set of tests, this score is calculated for each test, and then selected over. The tests are sorted using their risk scores, and the tests associated with the highest risks get included using a percentage-based threshold. In addition, we also include any test whose associated risk is over a differen threshold. Both of these values are configurable and can be used to alter the behaviour of the test selection method. For example, one can choose to always include each test that has a risk associated with it in the highest 20% of risks for a set of tests, as well as any test with an associated risk of 0.2 or higher. If one wishes to make a conservative selection, one can increase the percentage, lower the risk threshold, or both. For every experiment a sweep was performed over the minimum risk and percentage of highest risk included (0 - 1 in steps of 0.05, as well as 0.01, and 0 - 100% in steps of 5%, respectively).

3.2.1 Experiment II: Object of analysis

For this experiment a combination of information on tests and their outcomes, as well as information about code changes, had to be retrieved from separate sources and combined into a single dataset. The data about tests, their duration and execution times was retrieved from the Test Results Database, as in Experiment I, and information about the code changes was obtained from a code repository. First, we retrieve the date, ID and related source for every test plan which was executed within metrology in 2021 (Figure 5). The plan ID allows for the retrieval of all suites and cases that were executed. The source is a string identifier, which can be used to retrieve information about the code at the time of test executions. Some sources refer to parts of the codebase that are no longer stored, due to referring to experimental versions that have been deleted. Because of this, we then attempt to find each source existing in the Test Results Database in the repository. This resulted in 2,338 sources, each associated with a list of the files (including an absolute filepath in the repository system) which were changed prior to test execution. In total, 18,358,761 of changed files were found. In addition, we also find for each source all test plan IDs that are associated with them, as multiple plans may be executed on the same source. We use the plan IDs to retrieve all relevant information about associated suites and cases (Figure 6), including the constructed identifier for suites (as explained in section 3.1.1), the associated dates and times (the start, end and duration of each test), and the test outcomes. At this point, all the raw data that is used for the construction of the dataset is retrieved, and some small additional steps are taken prior to feature creation. As we would like to predict which tests will fail, all tests with the outcomes of 'UNSUPPORTED' and 'UNTESTED' (see section 3.1.1) were discarded from the dataset. We alter the dataset slightly by adding a shorthand building block ID, that is, only a numerical identifier that is associated to each building block, and use this instead of the full name associated with each building block. We create a mapping from each source to the last time of submission for any plan as tested on that source. Finally, some sources were found that were not associated with any file changes in the code repository. These nine sources were discarded from the dataset.

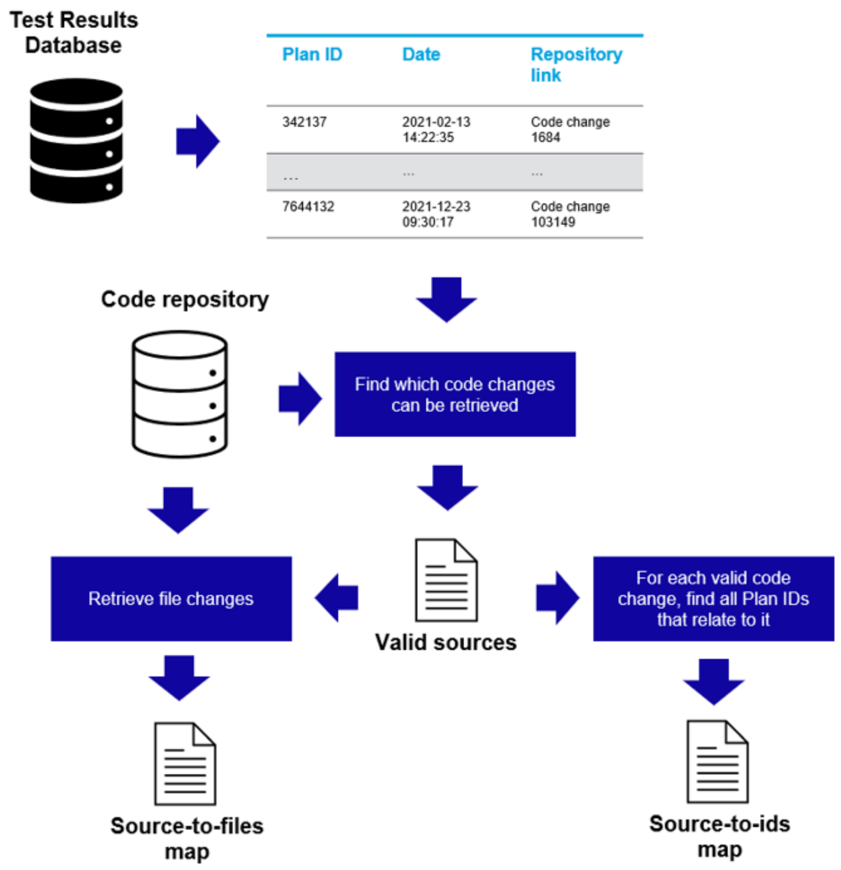


Figure 5: The first steps in creating the dataset involve finding which sets of file changes can still be accessed, and which tests were executed on them. Each file icon represents an intermediate artifact in the process of creating the final dataset.

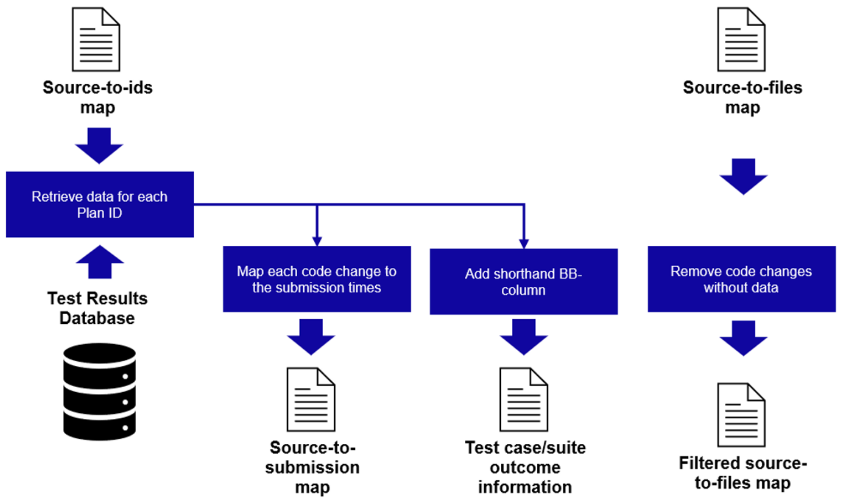

Source-to-ids map
Source-to-files map
Retrieve data for each Plan ID
Test Results Database
Map each code change to the submission times
Add shorthand BB-column
Remove code changes without data
Source-to-submission map
Test case/suite outcome information
Filtered source-to-files map

Figure 6: The subsequent steps in the preparation of the data consists out of compiling information with regards to when all tests for a source were executed, minor preprocessing to the test outcomes, and the removal of some sources that resulted in empty changefile lists.

3.2.2 Experiment II: Feature set creation

In the first step, we examine the file extension of every file that has been changed in every source (Figure 8). We chose to ignore file extensions which appeared fewer than $1,000$ times in all file changes. This was done as there are many extensions that were only utilized a few times, which provide little predictive value for test risk. This resulted in a reduction of file extensions from 954 to 166. The value of $1,000$ was initially tried, and the level of filtering achieved by this was deemed sufficient in removing irrelevant file types. Then for each source, we created a binary encoding of all filetypes that were present in that source. Next, we use the information with regards to changed files, submission times, and test outcomes associated with the sources, to compute several features for each source. We consider each file associated with a source, and keep track of how often that file has been changed, as well as how often a change in that file is associated with test failures. This is done by considering, for each time the file is changed, which fraction of the tests performed after that change resulted in a failure. In addition, these statistics are extended to the level of folders, that keep track of how often the files and folders contained within them fail and are associated with failing tests. For each source, these statistics are calculated and then aggregated over for different periods of time. We compute the mean, median, max and standard deviation of the number of changes and failures over a period of 7, 14, 28, and 56 days. This is done for the files in each source as well as for the parent and grandparents folders these files are situated in. We obtain these features by recreating the file system structure. For each set of file changes, we go over each changed file and keep track of the date at which it was changed, as well as what the fraction of failure of tests that ran on that change was. This information is also updated for the parent folder and grandparent folder of that file. For example, in Figure 7, if file k was changed, then we update these statistics for file k, folder 2 and folder m. Then to create the features for the source, we retrieve each change and failure rate associated with these files and folders over the aforementioned timespans. For the next step in the dataset creation process, we compute statistics with regards to test failures on the building block level. This is done by using the outcomes per suite, and then calculating the building block statistics on top. Each suite is associated with a building block, and thus we iterate over the results for each source, and then compute the fraction of passing suites per building block. We do this same aggregation for the duration of test executions. Next, we calculate for each building block the fraction of passing to all tests, the number of executions, as well as the mean, median and max duration of tests over 7, 14, 28 and 56 days. Finally, this same process is performed but on the suite and case level. The last step involves merging these different features into a single dataset, such that each row contains information about a single test (either on the case, suite or building block level), information about the files involved in the software change, and the aforementioned features about the test (Figure 9).

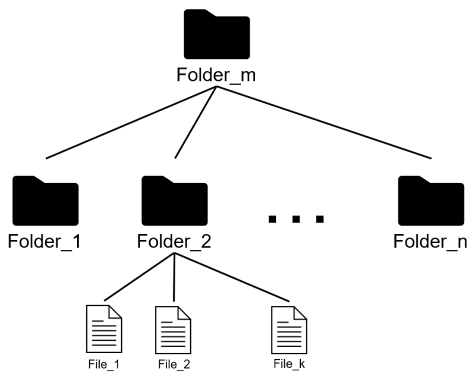


Figure 7: An example subsection of a file system.

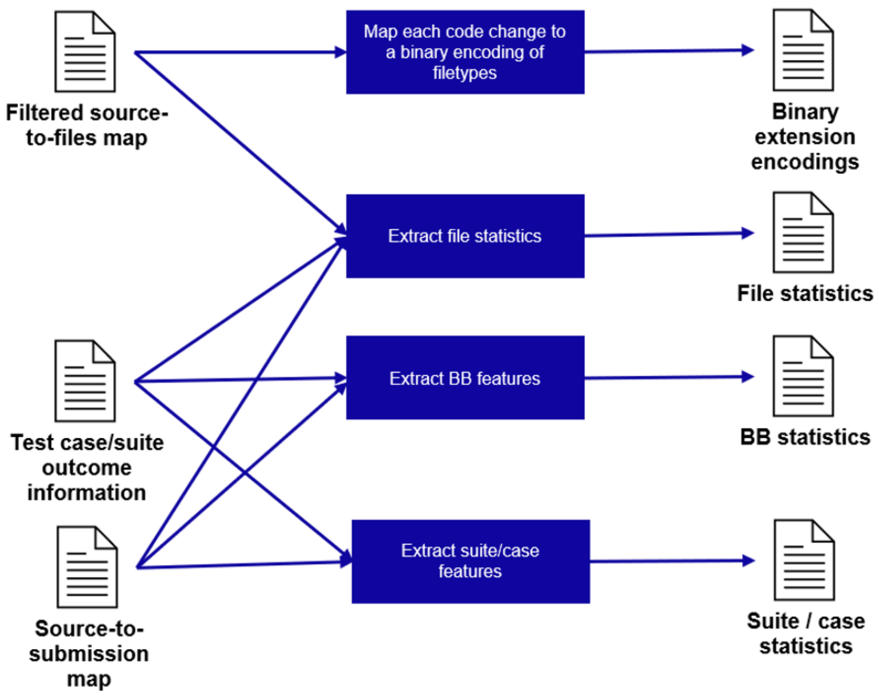

Filtered source-to-files map
Test case/suite outcome information
Source-to-submission map
Map each code change to a binary encoding of filetypes
Extract file statistics
Extract BB features
Extract suite/case features
Binary extension encodings
File statistics
BB statistics
Suite / case statistics

Figure 8: In the third step, we extract features from the data.

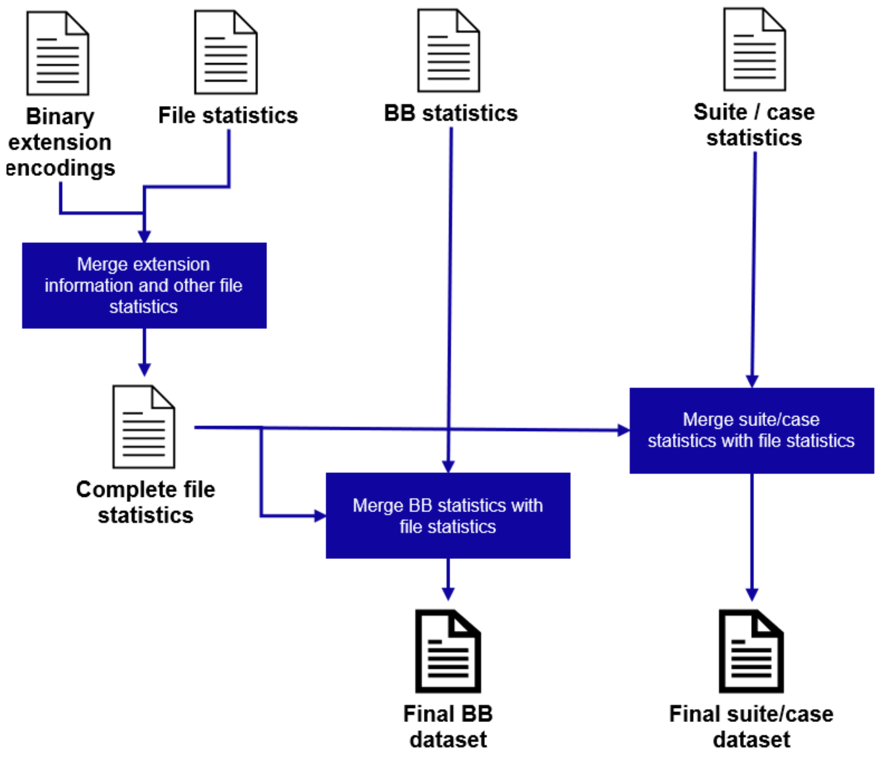

Binary extension encodings
File statistics
BB statistics
Suite / case statistics
Merge extension information and other file statistics
Complete file statistics
Merge suite/case statistics with file statistics
Merge BB statistics with file statistics
Final BB dataset
Final suite/case dataset

Figure 9: In the last step we combine all features into the final datasets.

3.2.3 Experiment IIa: Dataset split and feature selection

As test information is used which depend on the historical availability of data, all datapoints prior to the 9th of March 2021 are discarded. This resulted in 1,973 remaining sources. The dataset was then split in a test, validation and testing set in proportions of 60/20/20, as computed over the total number of sources present. For hyperparameter tuning and the exploration of various techniques in order to improve performance, our models were trained exclusively on the training set, and evaluated using the validation set. For the final results, the models were trained on both the training and validation set and evaluated on the testing set.
The training set of all three datasets, on the case, suite and building block level, were analyzed using the relief feature importance algorithm [48]. Next, models were trained using the top 10-100% of the features with the highest importance, as well as 25% and 75%. For each such featureset, gradient boosting regressors (using squared error training loss, and Friedman mean square error as a splitting criterion) were fitted using 50, 100, 250, 500, 750, and 1,000 estimators. Finally, a sweep was performed over the minimum risk and percentage of highest risk included hyperparameters (0 - 1 in steps of 0.05, as well as 0.01, and 0 - 100% in steps of 5%, respectively). For each model, we used the tradeoff between recall and time saved, and used this to select which features to include for further experimentation.

3.2.4 Experiment IIb: Pre-selection of tests

Based on Experiment IIa, it was found that the best overall performing level of granularity was selecting tests on the case or suite level, using all features (see section 4.3). In the previous experiment, we trained models to select over all test cases. Next, we attempted to fit the models after first performing a pre-selection on the tests we would predict over. This was done by examining the median runtime in a time-span of 56 days of each test. If this time was under a threshold, that test was excluded from the data for fitting the model. After the fitting had occurred, we evaluated the model on the validation data, by only applying it to tests for which the historical median runtime of the last 56 days was above the threshold. Any test below the threshold was assumed to be executed by default, and their execution times were counted towards the total execution time in the validation data. This was used for calculating the relative runtime reduction that was achieved using this technique. However, we did not include these tests in the recall metric. The intuition behind such a pre-selection was that the risk of missing a test that should have been executed, would not be worth it for the comparable small time save obtained by being correct. We fitted a series of models, varying the hyperparameters in the same manner as in section 3.2.3, and in addition, varying the pre-selection threshold (30s, 60s, 90s, 120s, 150s, 300s, 450s, 600s, 750s, 900s, 1,200s, 1,800s and 3,600s). The data that was excluded did not contribute towards the recall in our calculations, but their execution times were considered during the calculation of the time saved by any model.

3.2.5 Experiment IIc: Asymmetric loss

In predicting which tests may fail, there is an asymmetry in the severity of errors that can be made by the model. A test that should have been executed, but was not included due to a prediction, could cause potential issues with the reliability of the software, whereas executing a test that does not fail only has a cost insofar as the time and resources it needs to run. The loss metric used in prior experiments, the squared error, punishes both an over- and underestimation of risk with the same gravity. Therefore, we conducted an experiment in which we used a loss function such that an underestimation of the risk contributed more to the loss than an overestimation. For this we used an adjusted version of the squared error, also known as the 2-side quadractic loss [49], as depicted in Equation 15. The hyperparemeter a is used to scale the loss, which can be used to punish underestimation more severely, and b is used in the cases of an overestimation. The hyperparameters were varied as in section 3.2.3, and in addition, we varied the ratios of loss between under- and overestimation using values of 1:1, 2:1, 3:1, 5:1, 10:1, and 100:1.

$$L(y,F) = \begin{cases} \frac{1}{2}a(y-F)^2 & \text{if } y = 1 \\ \frac{1}{2}b(y-F)^2 & \text{otherwise} \end{cases} \tag{15}$$

3.2.6 Experiment IId: Asymmetric loss and pre-selection of tests

In our final experiment with regards to increasing the performance of predictive test selection, we combined the use of an asymmetric loss together with using a pre-selection based on historical run time. The hyperparameters were varied over in the same manner as in section 3.2.3, and on the basis of our earlier experiments, we considered different splitting times (30s, 60s, 90s, 120s, 150s, 300s, 450s, 600s), as well as asymmetric loss ratios (1:10, 1:100, 1:125, 1:250, 1:375).

3.2.7 Experiment IIe: Final model

Finally, we select the best model settings that have previously been found for the level of cases and suites, and re-train models using these settings on the train and validation data. We then use these models to evaluate the testing dataset. We determined the hyperparameters based on Experiment IId, and the best performance found for both the level of cases and suites was using an asymmetric loss function with ratio 1:100, performing a pre-selection on a duration of 60 seconds, using 250 estimators. These were obtained by, for each combination of asymmetric loss, pre-selection time and estimators (the non-risk hyperparameters), varying the hyperparameters for risk-based test selection (namely the minimum risk and the percentage of riskiest tests to included). For each combination of these non-risk hyperparameters, we varied the risk parameters as in section 3.2.3. We then observe the top 50 highest test times saved which have been obtained while maintaining a minimum recall of 0.9 up to 1.0 in steps of 0.01 (see section 3.2.3). For each combination of non-risk hyperparameters, and for each level of recall, we compute the median of these 50 best performers, and then we sum each median value of all recalls for each non-risk hyperparameter combination. We chose the non-risk hyperparameters which obtained the highest sum of medians using this method.
Afterwards, we selected the risk-based hyperparameters by considering which of these hyperparameters achieved the highest amount of time saved for each level of recall, resulting in one pair of risk-based hyperparameters for each level of granularity and for each recall. A second set of risk-based hyperparameters was also selected for each level of granularity, by considering the hyperparameters that were associated with the highest time saved that was equal to or below the median time saved for each level of recall. Once all of the hyperparameters were obtained, these models were then evaluated on the testing data. The final models were also evaluated on their ability to order tests using the APFD and APFDc metric, on the testing data. Before these latter statistics were calculated, each plan which only contained passing tests was not considered.

4 Experimental Results

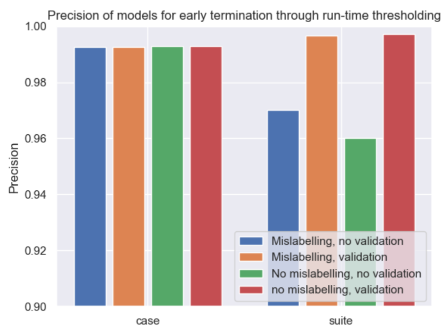
In this chapter, the results of our experiments are presented, and discussed in the subsequent chapter. As was the case for the methods, this chapter has been divided in two main experiments, I and II, in which the experiments based on the work of Philip et al. [1] can be found in Experiment I, and the experiments based on the work of Machalica et al. [2] in Experiment II. Experiment Ia discussed runtime thresholding and Ib correlation-based minimization. Experiment IIa discusses the results of feature selection, Experiments IIb-IId discuss using a pre-selection based on execution time, asymmetric loss and a combination of these two techniques. Finally, Experiment IIe discusses the results of using the best model found in Experiment IId and applying it to the testing data. It should be noted that in each experiment, the total time for suites is greater than that for cases. This is because only the duration for suites contains the time needed for setup. Therefore, when results with regards to time saves are shown, they are shown relative to the total suite time, as this time is the total time resources are actually occuppied during testing.

4.1 Experiment Ia

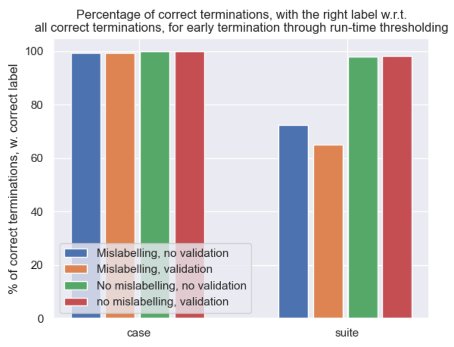
For each level of granularity, we fit a model that (dis)allows for mislabelling (early termination of any UNRESOLVED or FAILED test is considered correct, regardless of whether the correct label has been predicted), and that either uses the training set as the validation set (as per Philip et al. [1], labelled as 'No validation'), or uses a separate validation set. The results for these experiments can be seen in Figure 10. From these graphs, the results from using a selection on the level of building blocks have been ommitted, as the models fitted on this level performed poorly. In total, there were 106 building blocks. Using a separate validation set, this resulted in three fitted models that saved 0.0005% percent of all testing time in the testing data for building blocks (for both the models allowing and disallowing mislabelling). Using the training data as validation set, six models were fitted, but these models did not lead to any time savings. For suites, the precision obtained by the fitted models increased after the use of validation (Figure 10a), from 0.960 to 0.997 in the case of allowing for mislabelling, and from 0.970 to 0.997 when disallowing mislabelling. Selecting models on the basis of mislabelling terminations does increase the percentage of tests terminated with the correct label (Figure 10b). For suites this increased from 72.5% to 97.9% when no validation set was used, and from 65.2% to 98.13% otherwise. For cases, the increase was from 99.3% to 99.9% for both the experiments with and without a separate validation set. Models that were fitted on the suite level lead to the largest test runtime reduction, as calculated by taking the difference between the running time without the threshold, and the threshold, for those tests that run longer than it and that should be terminated (Figure 10c). Both models wasted a comparable amount of time, with waste being defined as the time up to the threshold for a test that was terminated incorrectly.

4.2 Experiment Ib

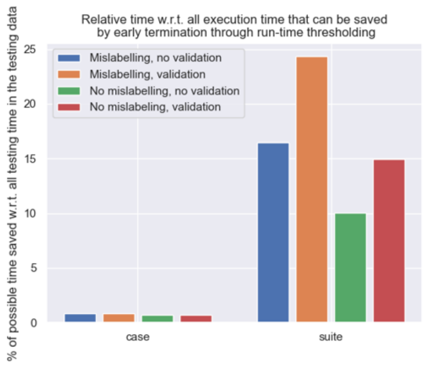
The outcomes of applying the rules as extracted from correlation can be seen in Table 1. Unfortunately, the rules obtained from finding correlations at the building block level with validation resulted in 0 actual predictions.



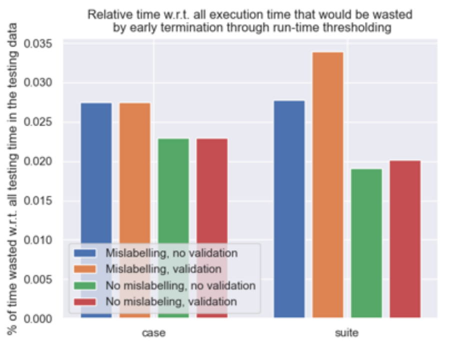
(a) precision



(b) mislabelling



(c) time saved by early termination



(d) time wasted by early termination

Figure 10: The performance of early termination at the level of cases vs. suites on varying metrics.

Granularity	Validation?	Precision	Time saved (%)	predicted tests (%)
Building block	Yes	-	-	0 %
Building block	No	0.991	0.12%	0.03 %
Suite	Yes	0.986	7.89%	15.1 %
Suite	No	0.979	10.14%	17.2 %

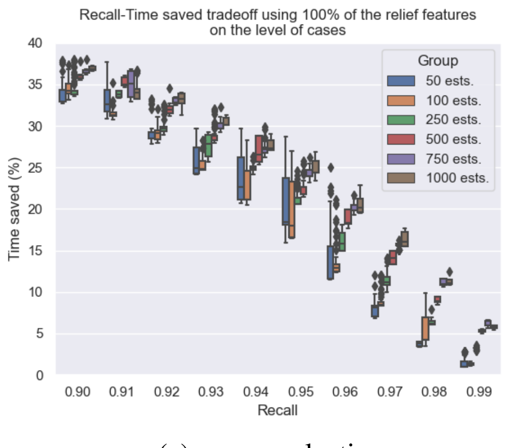
Table 1: The resulting precision and time saved after applying correlation-based prediction on tests on the suite and building block level, with and without validation.

4.3 Experiment IIa

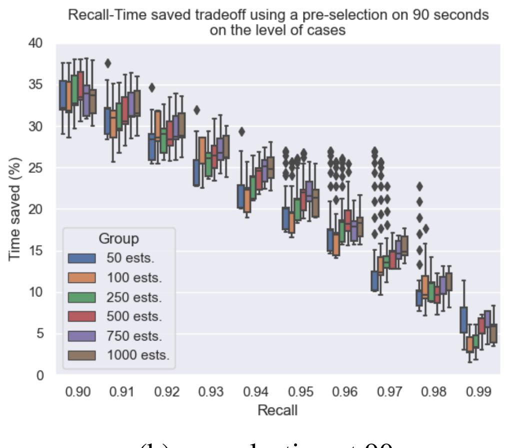
For each level of granularity, and each n% of relief features included, we display for each minimal recall the top 50 performers. We see, for each minimally required recall, what the time saved as a percentage of all tests is by these 50 best hyperparameter settings. The 50 best performers have been selected to allow for a better estimation of the performances that are possible, as this will allow us to see whether the amount of time saved might have been due to one very specific setting, or if there are multiple settings which can achieve a similar effect. We display only the top 50, given that many hyperparameters that have been explored result in very low time saves. The results of the best performers for each experiment on the case and suite level can be seen in Figure 12. For cases and suites the best performance was obtained by using all features, and for building blocks, this was obtained by using the top 10% of relief features. For all results, see Figure 18, Figure 19 and Figure 20 in the appendix.

4.4 Experiment IIb: Pre-selection on time

Based on the previous experiment, it was found that the most time could be saved whilst using either case or suite level granularity, using all features. The best performing splitting time was found to be 150 seconds for both cases and suites. The results for this split can be seen in Figure 12. For the detailed results see Figure 22 and Figure 23 in the appendix. Whilst there seems to be a small difference in performance, it should be noted that the models which have been trained on the data using a pre-selection resulted in good performance by models with a low amount of estimators used (Figure 11), due to a reduction in the size of the dataset.



(a) no pre-selection

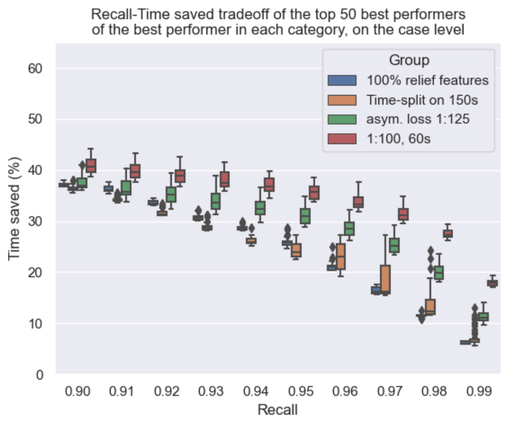


(b) pre-selection at 90s

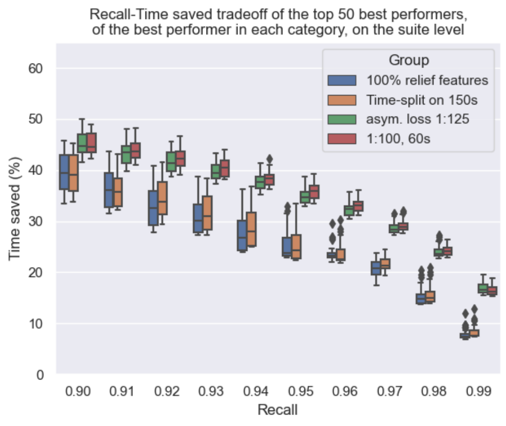
Figure 11: The performance of using a varying number of estimators on the recall vs. time saved tradeoff on the level of suites. We see that by using a pre-selection, the performance of models with a small amount of estimators increases as compared to the case in which this is not used, at higher levels of recall (0.97 and above).

4.5 Experiment IIc: Asymmetric loss

As with the previous experiment, we considered granularity on the level of cases and suites, using all features. The best performance on the level of cases and suites was found at using a



(a) case level



(b) suite level

Figure 12: The recall vs. time saved tradeoff obtained by the best-performing hyperparameters found for Experiments IIa-IId.

ratio of 1:125, which can be seen in Figure 12. For the full results, see Figure 24 and Figure 25 in the appendix.

4.6 Experiment IId: Asymmetric loss and preselection of tests

As with the previous experiment, we considered granularity on the level of cases and suites, using all features. The best performance on the level of cases and suites was found at using a ratio of 1:100, and a pre-selection splitting time of 60 seconds, which can be seen in Figure 12. For the full results, see Figure 26, Figure 27, Figure 28 and Figure 29 in the appendix.

4.7 Experiment IIe: the final model

The recall vs. time saved tradeoff for all considered hyperparameter settings of the minimum risk, as well as the top n% of risk to include, can be seen in Figure 13. Whilst this shows the best performing settings for each minimal desired recall, in a real deployment situation we have to base these hyperparameters on the performance of the model on previously seen data. In Figure 14 and Figure 15 we see the results of two such selections. Finally, we evaluated the suitability of using the final model for prioritization, and calculated the APFD and APFDc on the level of cases (Figure 16) and the level of suites (Figure 17), under the assumption that each test failure corresponds to a unique fault. For the APFDc metric, we assumed that each fault had identical severity, and used the actual test execution time in seconds as the cost. For comparison, we also show the resulting APFD and APFDc values for heuristic orderings, including a random ordering, ordering by execution time (in ascending and descending order, using the median execution time over the last 56 days), and ordering by passing rate in ascending order for various time windows (7, 14, 28 and 56 days).

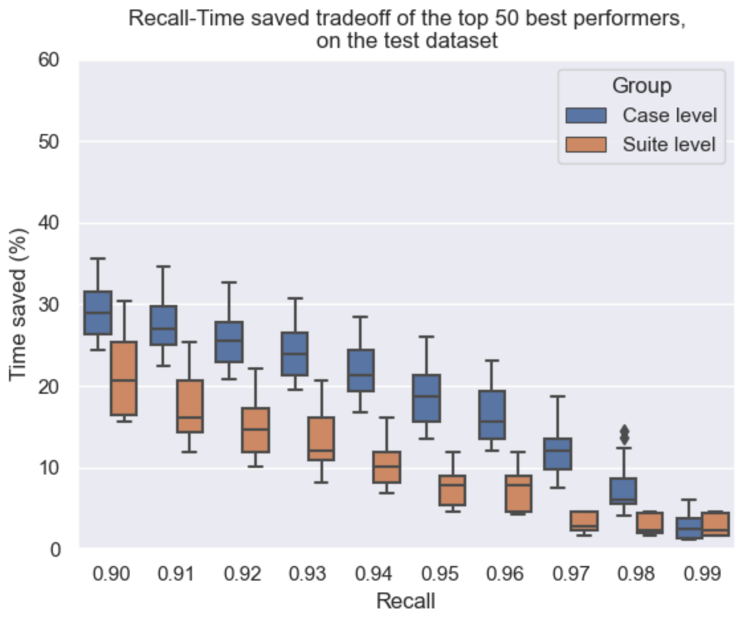
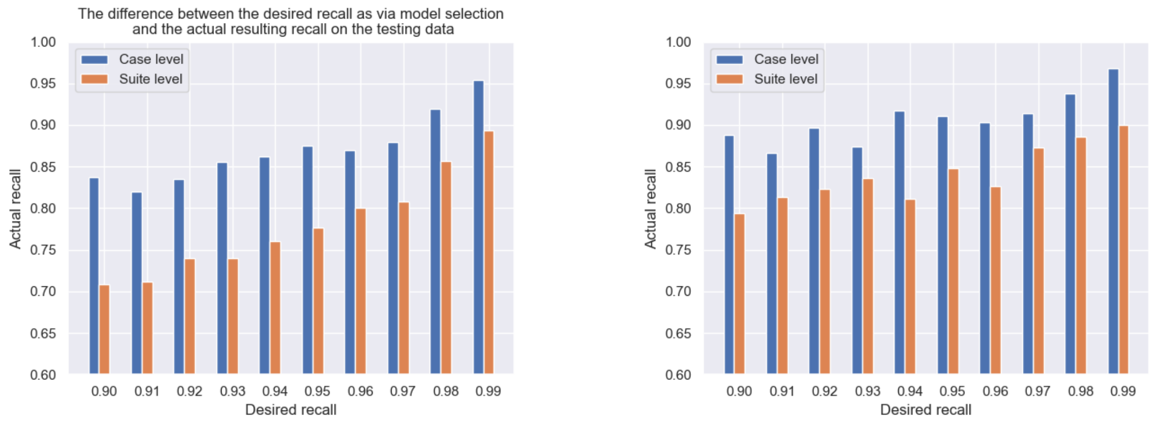
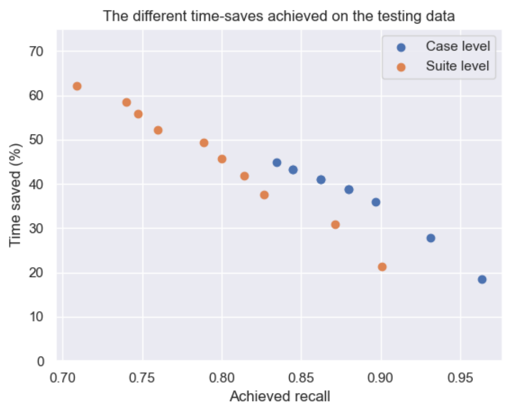


Figure 13: The recall vs. time saved tradeoff for models that were fitted on the test dataset.

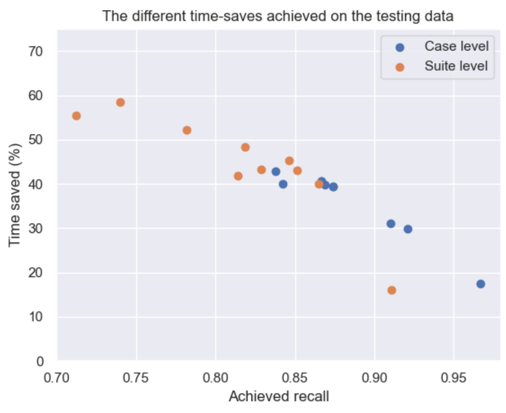


(a) best hyperparameters (b) median hyperparameters

Figure 14: In each graph, we see the difference between the desired recall for which the hyperparameters (min risk, top n% of risk to include) were selected using the validation dataset, and the actual recall of the model once applied to the testing dataset. In Figure 14a, we see the performance of the settings for the settings chosen by taking the settings which resulted in the highest amount of time saved for each level of recall, and in Figure 14b, we see the performance for the settings chosen by taking the settings which resulted in at most the median amount of time saved of the 50 best estimators for each recall level.

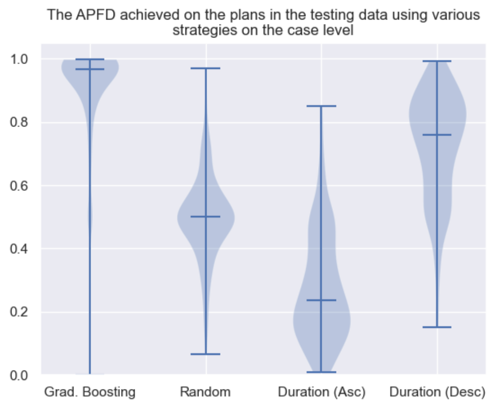


(a) best hyperparameters

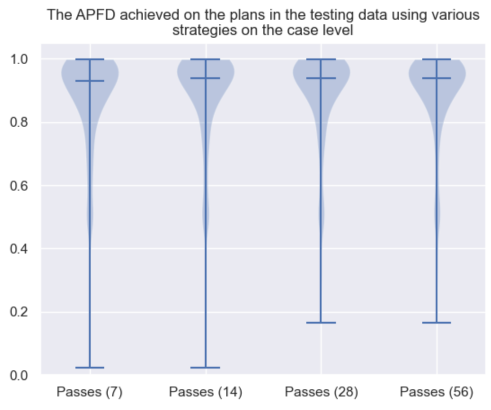


(b) median hyperparameters

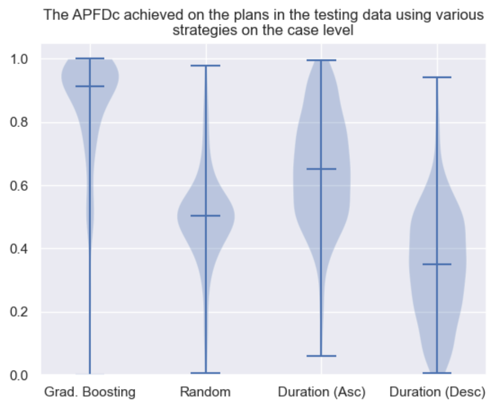
Figure 15: In each graph, we see the actual obtained recall, and the time saved at that level of recall, using the hyperparameter selection as explained in Figure 14.



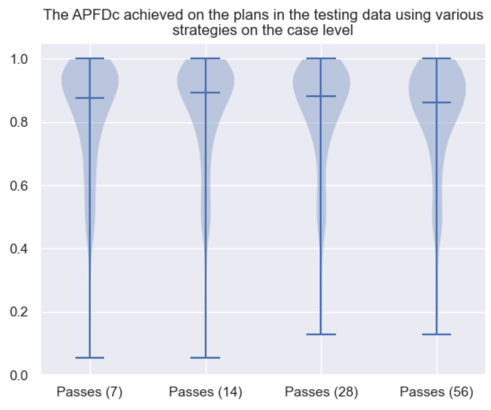
(a) APFD



(b) APFD

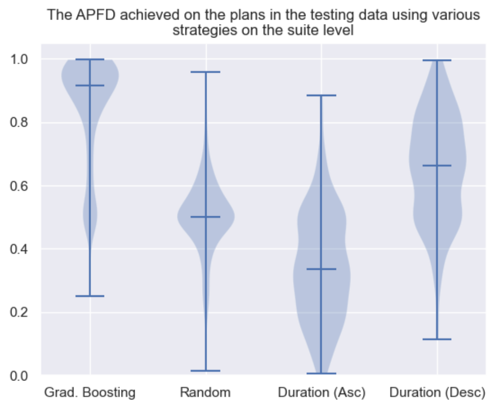


(c) APFDc

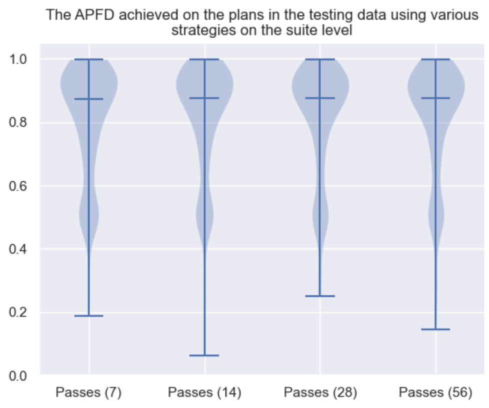


(d) APFDc

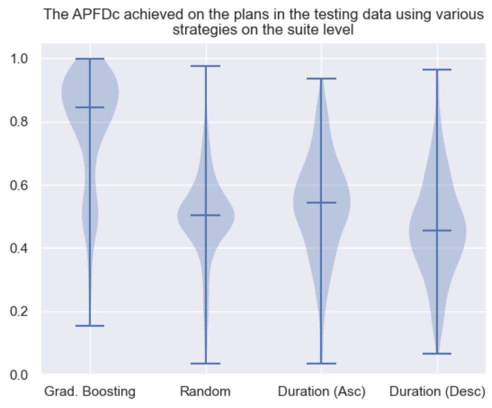
Figure 16: The distribution of the APFD and APFDc values for each test plan in the testing data, on the level of cases. We compare the ordering induced by using the risk scores from the gradient boosting regressor, and compare it against various heuristic orderings.



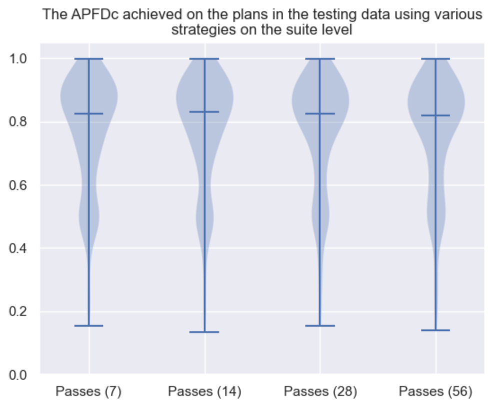
(a) APFD



(b) APFD



(c) APFDc



(d) APFDc

Figure 17: The distribution of the APFD and APFDc values for each test plan in the testing data, on the level of suites. We compare the ordering induced by using the risk scores from the gradient boosting regressor, and compare it against various heuristic orderings.

5 Discussion

Concerning the granularity on which tests are selected, we found that in all experiments, considering a higher level grouping above the level of suites, at least in the form currently used by ASML, does not outperform selection on the case or suite level. For runtime outcome prediction using a logistic regression, we found that the amount of time that can be saved is the highest on the level of suites, versus on the level of cases, with comparable recall (provided that a validation step has been applied), as can be seen in Figure 10. A contributing factor to this stark difference is that runtime prediction terminates an entire suite if the time the suite has been executing exceeds its threshold. From this it follows that all remaining test cases in the suite will not be executed. In the current system, such a stuck test case is allowed to execute up to its maximum execution time, which either is set at a default time or a custom time by developers, after which it will be marked as unresolved. This has no consequences for the next test case of the suite, which is executed normally after the previous test timed out. Based on these findings, we therefore suggest that runtime prediction should be applied on the level of suites, if it is important that every case in the suite passes, and otherwise, a selection on the level of cases is warranted. We do not expect that the difference in total running time of cases and suites over the same period to be a large contributor, as this difference mainly stems from start-up overhead which is included within the time of the suite, but not in the times of the cases contained within. For the experiments using runtime outcome prediction, the time spent on suites is 2.35 times that on cases in the testing period. While this contributes to the difference in time saved as in Figure 10, this difference is too large to be only explained by this effect.

We also see differences in the performance of predictive test selection on different levels of granularity (Figure 15). A selection on the level of suites results in a larger difference between recall on the validation and training data (Figure 14b) than a selection on the level of cases. In addition, the amount of time saved by selecting on the level of cases is greater than for suites (Figure 15), in the recall range from 0.9 to 1.0. This also holds if we examine the resulting best time saves by considering each possible value for the risk-based hyperparameters (Figure 13). Based on the recommendation of a test engineer at ASML, a recall of 0.9 is the minimally desired performance for a system to be functional during deployment, and therefore we conclude that test selection should be performed on the level of cases.

With regards to the various suggestions made to predictive test selection, the performance on the validation data suggests that using an asymmetric loss function leads to a better tradeoff between recall and time saved. However, we see a worse performance once these models have been applied to the testing dataset. We offer a few possible reasons. First, it was found that applying a pre-selection based on historical test runtime resulted in models with less estimators having a higher best performance on the validation dataset than models with more estimators (Figure 11). This effect is particularly visible for the higher recalls that have been considered (0.97 and above). We consider this to be in part due to the reduction in data on which the model is trained overall, and we suspect that tests with longer execution times behave differently than shorter ones. This is supported by examining the differences in APFD for ordering cases and suites by execution time (Figure 16, Figure 17). This would also explain why the best performing asymmetric loss ratio of 1:100 in Experiment IId (with pre-selection) was less asymmetrical than the best performing asymmetric loss ratio of 1:125 in Experiment IIc (without pre-selection). A higher asymmetric loss ratio implies more uncertainty present with regards to which tests will fail. This is compensated by forcing the models to be more conservative in their risk assesment, and hence, a higher loss for risk underestimation. This difference in the best performing asymmetric loss ratio implies that it is easier to identify failing tests given a longer execution time.

Without using a pre-selection, it was found that the best performing model on the validation set included all features. However, it may be the case that the reduced dataset that is being trained on in the case of using a pre-selection on time has different feature relevances, and because of this, the model may have overfitted due to having access to all features. Second, during the process of hyperparameter selection, multiple models were trained, which further increased the risk of finding models that happened to perform well on the validation set by chance or erroneous patterns. Finally, the testing data contained data points created using code changes which were performed at the end of the year. Due to the planning process used within ASML, in the month of December many changes are made before wrapping up for a new year, which may result in less predictable behaviour during this period of higher volatility.

We do observe that using a gradient boosting model trained for predictive test selection for prioritization outperforms several commonly used heuristics in the field (Figure 16 and Figure 17), both in terms of APFD and APFDc, on this specific dataset. Whilst this is a method used in the literature to evaluate the potential of a method, in order to further investigate the viability of this prioritization technique comparisons need to be performed against other state-of-the-art techniques. Until a commonly agreed benchmark exists in the field that can provide detailed information about historical test information as well as file changes, we consider this to be the next logical step.

As was mentioned within the method section, no identifiers are stored in order to track multiple executions of the same test case or suite. Because of this, an identifier had to be constructed on the basis of other information which was present in the database. However, this identifier is not unique, as it relies on information from a test specification file. Although it is discouraged, developers can specify multiple different tests in a single test configuration. If these suites all get executed with the same configuration file and on the same machine, then this results in multiple records in the test results database which will be assigned the same identifier. These collisions also occur at the level of case identifers. As we have seen in our experimentation on the level of the granularity of selection, a coarser level of granularity can be beneficial up to a point (selection on the level of suites resulted in a higher amount of time saved on validation data, for lower recalls (0.90 - 0.95), but on the level of building blocks this effect dissapeared), and thus we believe that these collisions in the identifiers harms the performance of all techniques applied in this thesis. We therefore argue that the performance which has been achieved could be additionally improved by increasing the quality of the data by ensuring that developers submit separate test specification files for each separate suite.

Regrettably, we could not evaluate the performance of combining predictive test selection with runtime thresholding or correlation-based prediction. This was because of the nature of the dataset that had been obtained. Only approximately 50% of the test outcomes (by execution time) could be linked to information about the code on which these tests were applied. This information is necessary for predictive test selection, and therefore predictive test selection could only be performed on a subset of all tests. Unfortunately this subset did not contain the tests which were found to correlate during Experiment Ib, and neither did it contain the tests for which a runtime thresholding model could be fitted in Experiment Ia.

6 Conclusion

6.1 Summary of Main Contributions

In this work, we:

- replicated the techniques of Philip et al. and Machalica et al. [2] on a novel dataset.
- have suggested improvements to the work of Philip et al. [1] by recommending the inclusion of a validation set.
- have shown that runtime prediction worked best on the level of suites as compared to cases, when comparing the potential time that could be saved (at most 24.3% vs. 2.0%).
- have shown that predictive test selection performed best on the level of cases, compared to the level of suites, where for each minimally required recall from 0.90 to 1.0 in steps of 0.01, both the best and median time saved of the top 50 performers was greatest on the level of suites.
- have shown that selecting at a higher level of organisation, such as on the level of building blocks within ASML, is not a viable strategy, resulting in a best time save of 5.3% with a recall of 0.90.
- have suggested improvements to predictive test selection by using an asymmetric loss function, as well as selecting over which tests should be predicted based on historical runtime.
- obtained a maximum time save of 31.1% whilst still maintaining a recall of above 0.9 using predictive test selection on the level of cases. This is considered viable for real-world use.
- have shown that predictive test selection is a viable technique for prioritization by comparing it to heuristic orderings on the basis of a random ordering, historical execution times, and historical failure rates. Predictive test selection obtained a higher median APFD and APFDc than any heuristic.

6.2 Future Work

In the data which was examined for this thesis, there was no overlap between the data on which predictive test selection, and runtime based prediction or correlation using minimization, could be applied. As a result, it remains to be shown what the influence on time saved is when applying all of these techniques together. In order to do this, the next step within ASML is to try to link more results from the Test Results Database to code changes. In this thesis, we have only considered the data of 2021 as a whole, where 80% of the data was used for training and evaluating models, and these models were then finally evaluated on the remaining 20%. Because of this, the difference in date between the earliest training and testing record is a period of months. It may be the case that patterns in the data drift over time such that this early data is not useful for predicting over such a large span in time. To address this issue, we suggest experiments that vary the amount of months that are included in the training and validation data. We envision that this approach will also address the gap that we have seen between the desired recall on which we select hyperparameters on the validation data, versus the actually obtained recall using these on the test data. Currently this difference is large, the difference was found to be as high as 0.28 on the level suites, and 0.15 on the level of cases. For predictive test selection to be deployed effectively, its performance is required to be more predictable, to ensure that the system maintains at least a minimally acceptable recall. Finally, we were not able to evaluate test case correlation on the scale of a year due to the amount of data present, and the high time complexity of calculating correlation. If we consider smaller periods in time, it will become feasible to obtain this information as well.

The findings with regards to the improvements made to runtime based prediction, correlation-based prediction and predictive test selection, have only been examined in the context of this dataset. Given the lack of benchmarks in the field, and considering the amount of information that needs to be present for all features to be constructed that are used in the models, we consider it to be unlikely for such a dataset to become available in the near future. Therefore we suggest the comparison of other state-of-the-art methodologies on ASML data, in order to provide more evidence for the efficacy of the proposed improvements in this paper. An important open question is how these models would perform once utilized in deployment. As the quantity of tests that will be executed will be lowered once these models are used, and the execution time is reduced due to the application of thresholding, it is necessary to run a fraction of the tests on which the models were applied in full. This is necessary in order to verify that the model is working as intended, but this fraction needs to be small enough to still lead to a sufficient reduction in execution time. However, a small fraction also leads to a reduction in the quality of many temporal features and thus, possibly a degradation of performance. We propose a pilot study within ASML, whereby a simulation is executed alongside the regular testing system without selection in place. The simulation then calculates which tests would have been included if selection were to be applied, and create features only using these tests. We can then compare the outcome of the simulation, to applying predictive test selection on features using the full information obtained from the regular testing system.

Bibliography

[1] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "FastLane: Test minimization for rapidly deployed large-scale online services," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 408–418, IEEE, 2019. event-place: Montreal, QC, Canada.

[2] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection." http://arxiv.org/abs/1810.05286, 2019. Accessed: 2022-03-30.

[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, pp. n/a–n/a, 2010.

[4] M. Czerwinski, E. Horvitz, and S. Wilhite, "A diary study of task switching and interruptions," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 175–182, 2004.

[5] S. Leroy, "Why is it so hard to do my work? the challenge of attention residue when switching between work tasks," *Organizational Behavior and Human Decision Processes*, vol. 109, no. 2, pp. 168–181, 2009.

[6] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996. Conference Name: IEEE Transactions on Software Engineering.

[7] G. Rothermel, *Efficient, effective regression testing using safe test selection techniques*. PhD thesis, Clemson University, 1996.

[8] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[9] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 583–594, 2016.

[10] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*, pp. 290–301, IEEE, 1990.

[11] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *JOOP*, vol. 8, no. 2, pp. 51–65, 1995.

[12] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 241–251, 2004.

[13] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pp. 9–pp, IEEE, 2005.

[14] D. Correia, R. Abreu, P. Santos, and J. Nadkarni, "MOTSD: a multi-objective test selection tool using test suite diagnosability," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1070–1074, ACM, 2019.

[15] M. Gligoric, L. Eloussi, and D. Marinov, “Practical regression test selection with dynamic file dependencies,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 211–222, 2015.

[16] M. Gligoric, L. Eloussi, and D. Marinov, “Ekstazi: Lightweight test selection,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 713–716, IEEE, 2015.

[17] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo, “Testtube: A system for selective regression testing,” in *Proceedings of 16th International Conference on Software Engineering*, pp. 211–220, IEEE, 1994.

[18] R. Martins, R. Abreu, M. Lopes, and J. Nadkarni, “Supervised learning for test suit selection in continuous integration,” in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 239–246, 2021.

[19] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, “Taming google-scale continuous testing,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 233–242, 2017.

[20] S. U. R. Khan, S. P. Lee, N. Javaid, and W. Abdul, “A systematic review on test suite reduction: Approaches, experiment’s quality evaluation, and guidelines,” *IEEE Access*, vol. 6, pp. 11816–11841, 2018.

[21] G. Rothermel, M. J. Harrold, J. Ostrin, and C. 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)*, pp. 34–43, IEEE, 1998.

[22] P. S. Kochhar, F. Thung, and D. Lo, “Code coverage and test suite effectiveness: Empirical study with real bugs in large systems,” in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pp. 560–564, IEEE, 2015.

[23] D. Leon and A. Podgurski, “A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases,” in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pp. 442–453, IEEE, 2003.

[24] A. E. V. B. Coutinho, E. G. Cartaxo, and P. D. d. L. Machado, “Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing,” *Software Quality Journal*, vol. 24, no. 2, pp. 407–445, 2016.

[25] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, “Test suite reduction and prioritization with call trees,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 539–540, 2007.

[26] G. Kumar and P. K. Bhatia, “Software testing optimization through test suite reduction using fuzzy clustering,” *CSI transactions on ICT*, vol. 1, no. 3, pp. 253–260, 2013.

[27] C. Coviello, S. Romano, and G. Scanniello, “Poster: Cuter: Clustering-based test suite reduction,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 306–307, IEEE, 2018.

[28] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 210–221, IEEE, 2018.

[29] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[30] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, p. 106268, 2020.

[31] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pp. 179–188, IEEE, 1999.

[32] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[33] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pp. 329–338, IEEE, 2001.

[34] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[35] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 346–357, 2019. ISSN: 2159-4848.

[36] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *2013 IEEE sixth international conference on software testing, verification and validation*, pp. 312–321, IEEE, 2013.

[37] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 361–368, IEEE, 2016.

[38] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 324–336, 2020.

[39] H. Spieker, A. Gotlieb, D. Marijan, and M. 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*, pp. 12–22, 2017.

[40] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *arXiv preprint arXiv:2106.13891*, 2021.

[41] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 975–980, 2016.

[42] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1–12, 2020.

[43] S. Kirdey, K. Cureton, S. Rick, S. Ramanathan, and M. Shukla, "Lerner - using RL agents for test case scheduling." `https://netflixtechblog.com/lerner-using-rl-agents-for-test-case-scheduling-3e0686211198`. Accessed: 2021-11-30.

[44] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 382–391, IEEE, 2011.

[45] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic regression*. Springer, 2002.

[46] S. Suthaharan, "Decision tree learning," in *Machine Learning Models and Algorithms for Big Data Classification*, pp. 237–269, Springer, 2016.

[47] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[48] K. Kira and L. A. Rendell, "A practical approach to feature selection," in *Machine learning proceedings 1992*, pp. 249–256, Elsevier, 1992.

[49] H. Schabe, "Bayes estimates under asymmetric loss," *IEEE transactions on reliability*, vol. 40, no. 1, pp. 63–67, 1991.

Appendices

A Additional plots

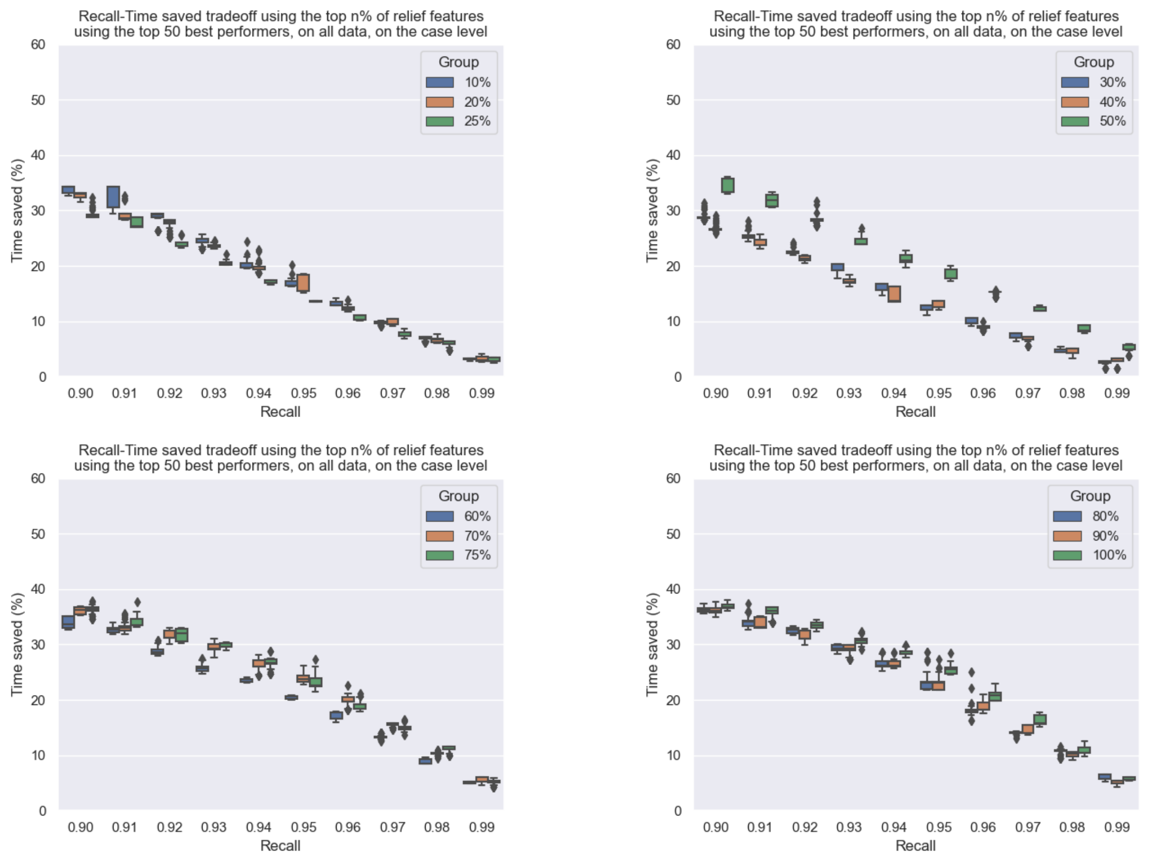


Figure 18: Multiple models were fitted on the test case dataset using the top n% features as found using the relief algorithm. For each minimally required recall, we see the time saved (as percentage over all tests) by the top 50 best performing hyperparemeter settings.

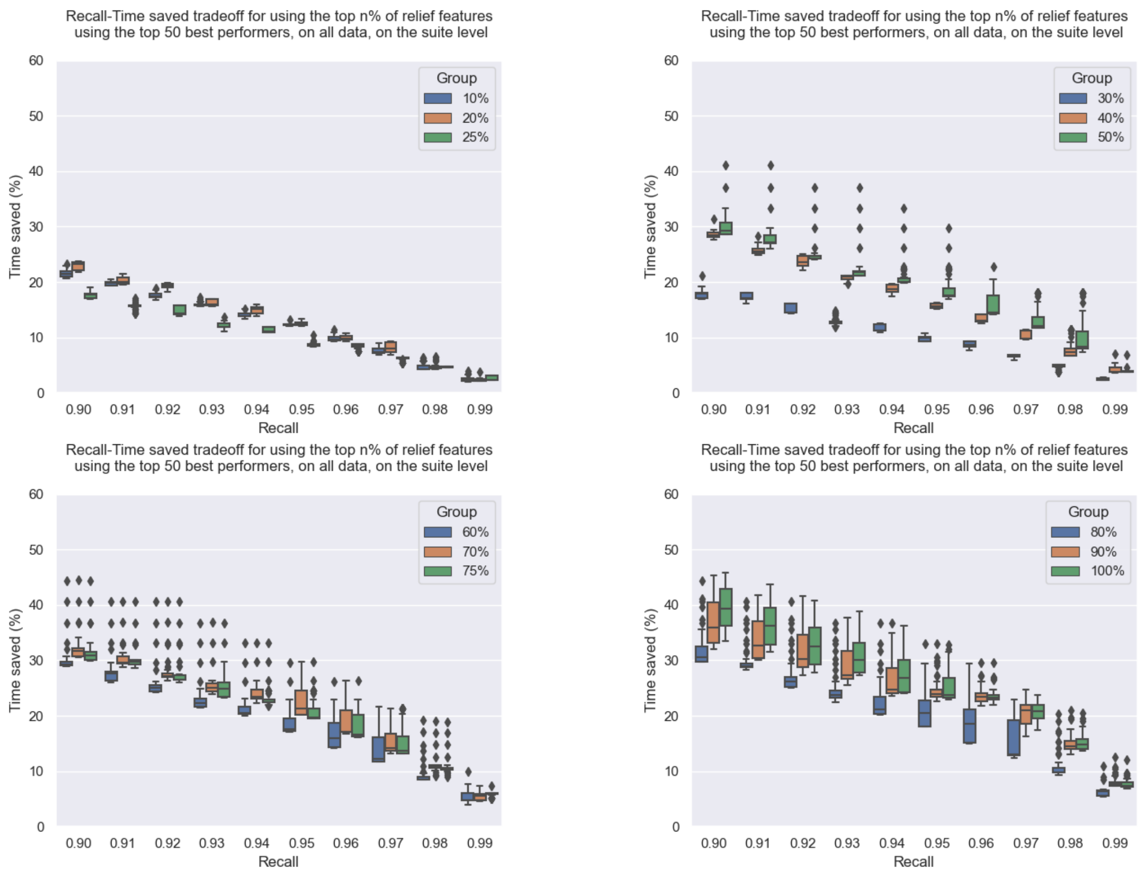


Figure 19: The results of fitting models using the top n% of features as found using the relief algorithm, on the level of suites. For a zoomed in version of these graphs, see **??**.

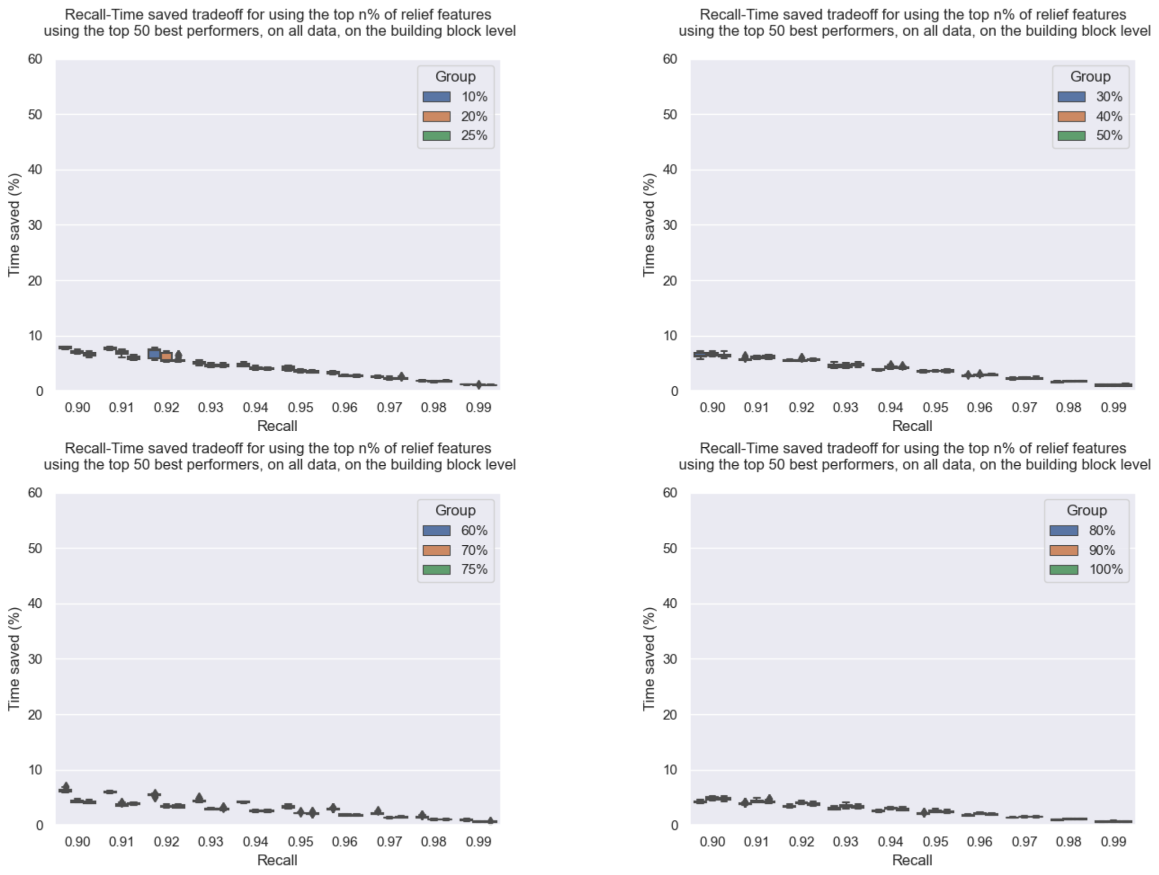


Figure 20: The results of fitting models using the top n% of features as found using the relief algorithm, on the level of building blocks. For a zoomed in version of these graphs, see Figure 21.

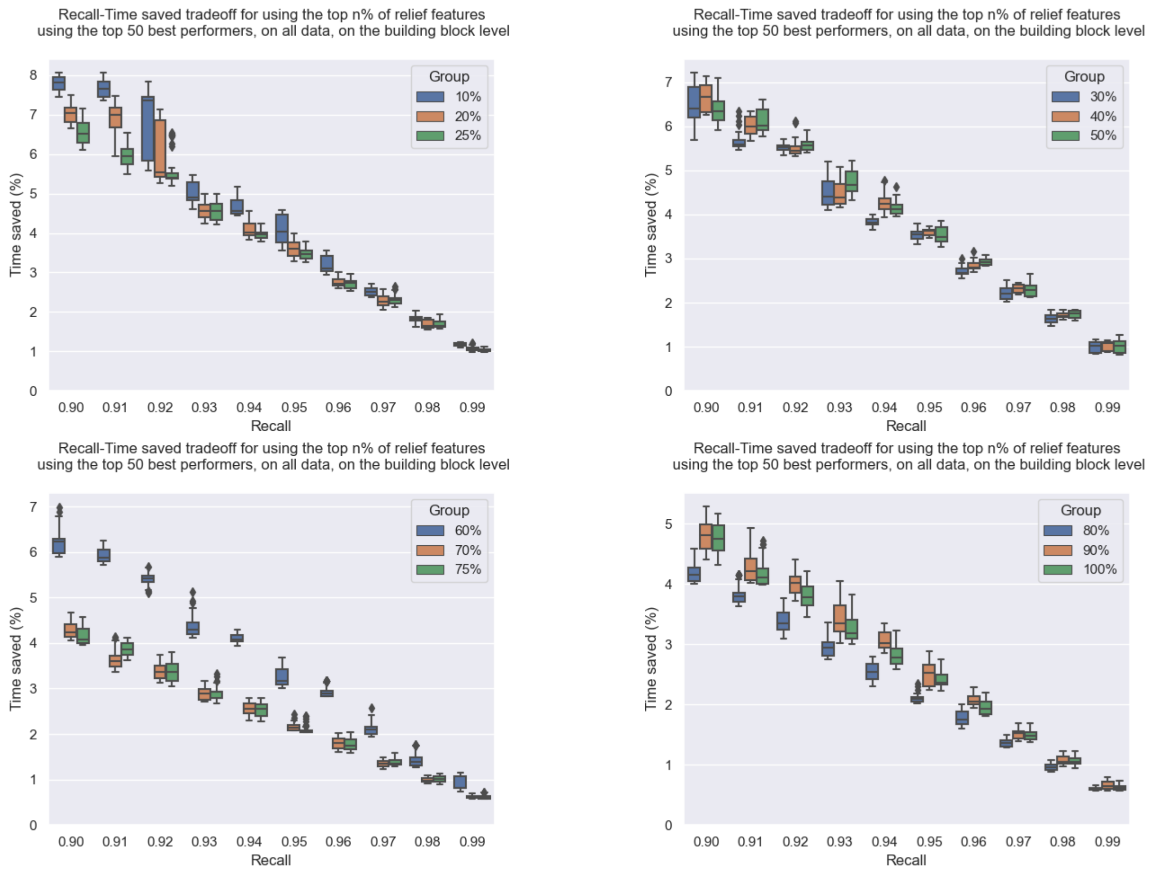


Figure 21: Multiple models were fitted on the building block dataset using the top n% features as found using the relief algorithm. For each minimally required recall, we see the time saved (as percentage over all tests) by the top 50 best performing hyperparemeter settings.

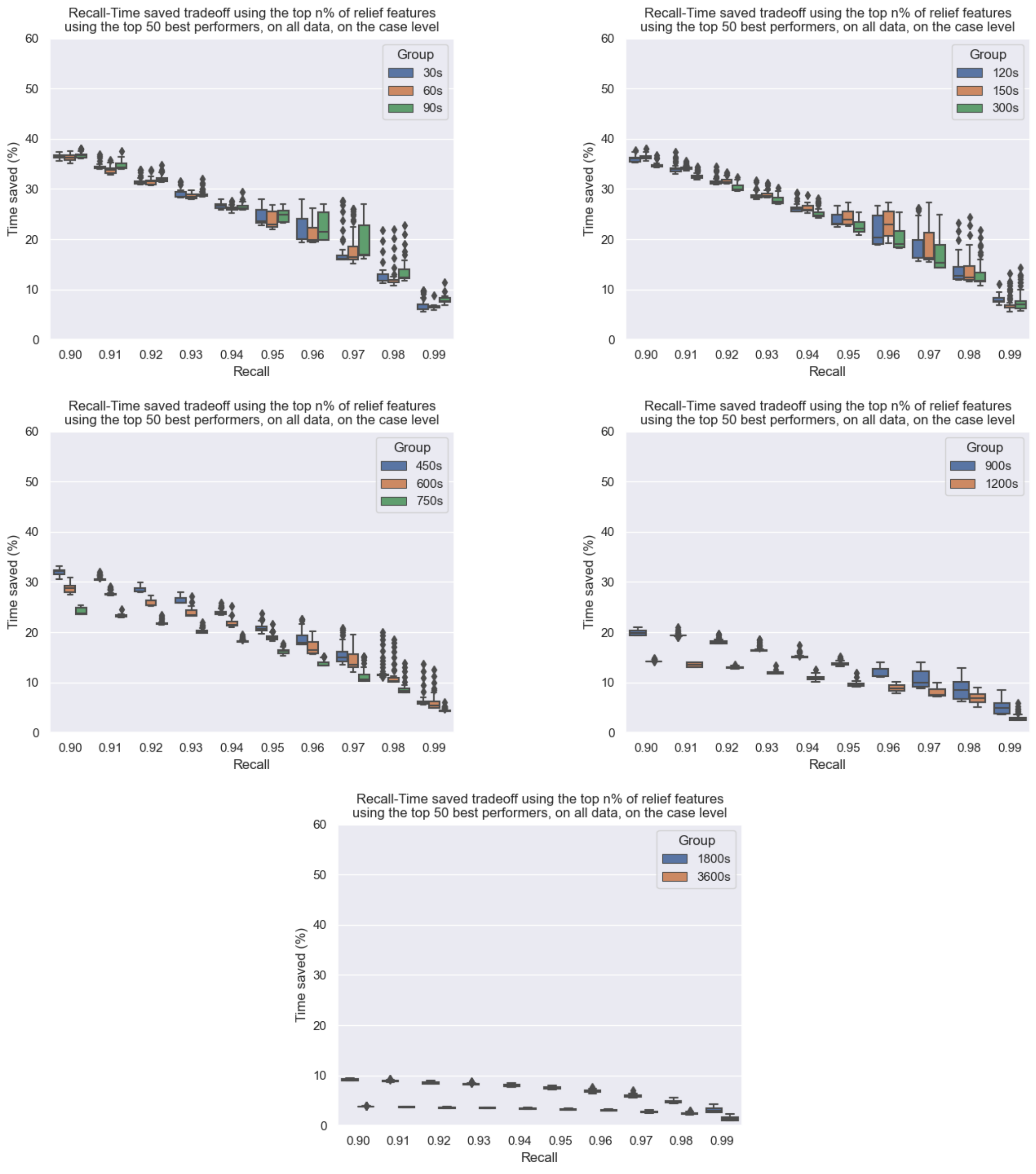


Figure 22: Multiple models were fitted on the test case dataset using all features. Prior to fitting the models, all tests with a duration below the threshold were filtered out of both the training and validation set.

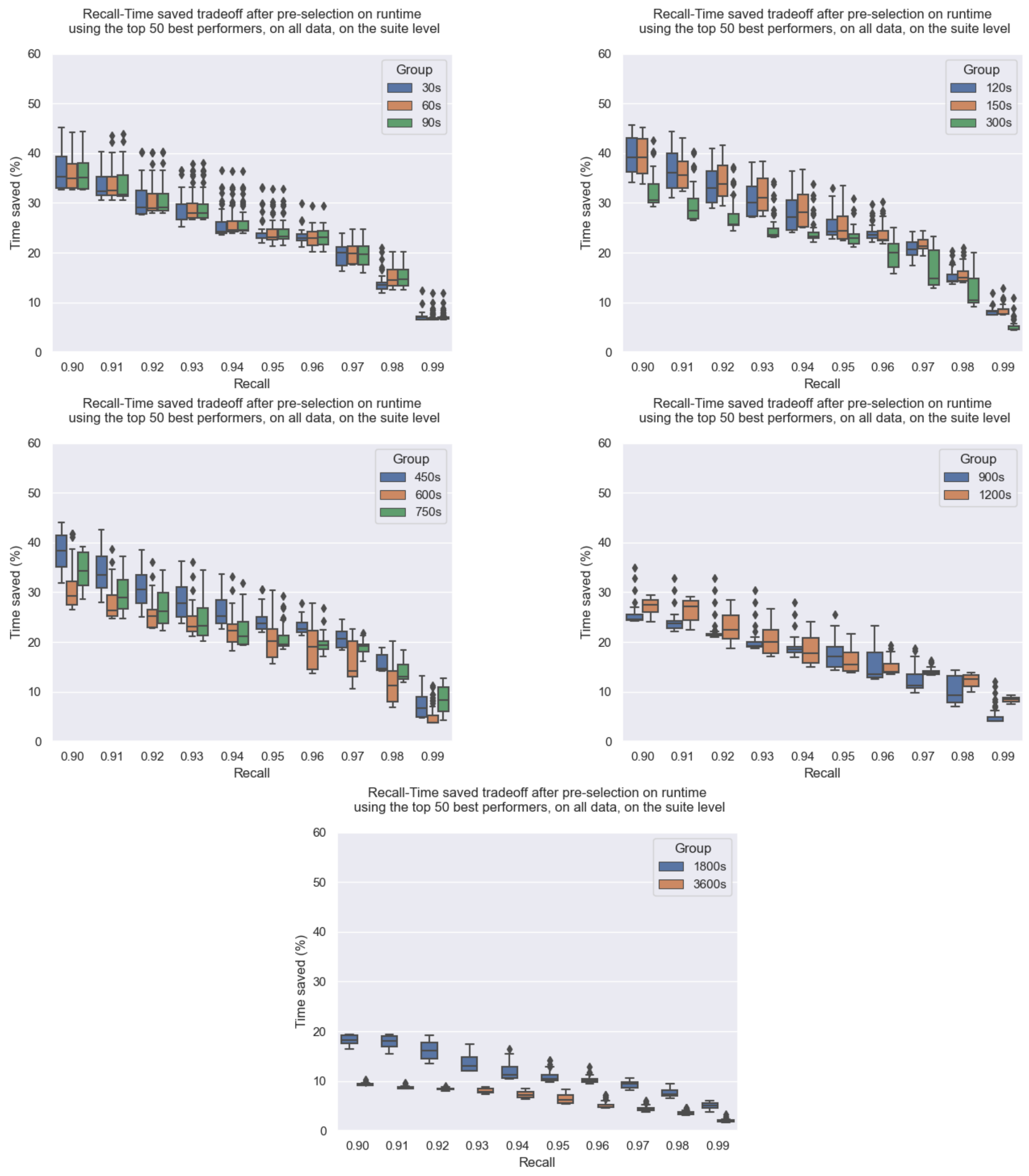


Figure 23: Multiple models were fitted on the test suite dataset using all features. Prior to fitting the models, all tests with a duration below the threshold were filtered out of both the training and validation set.

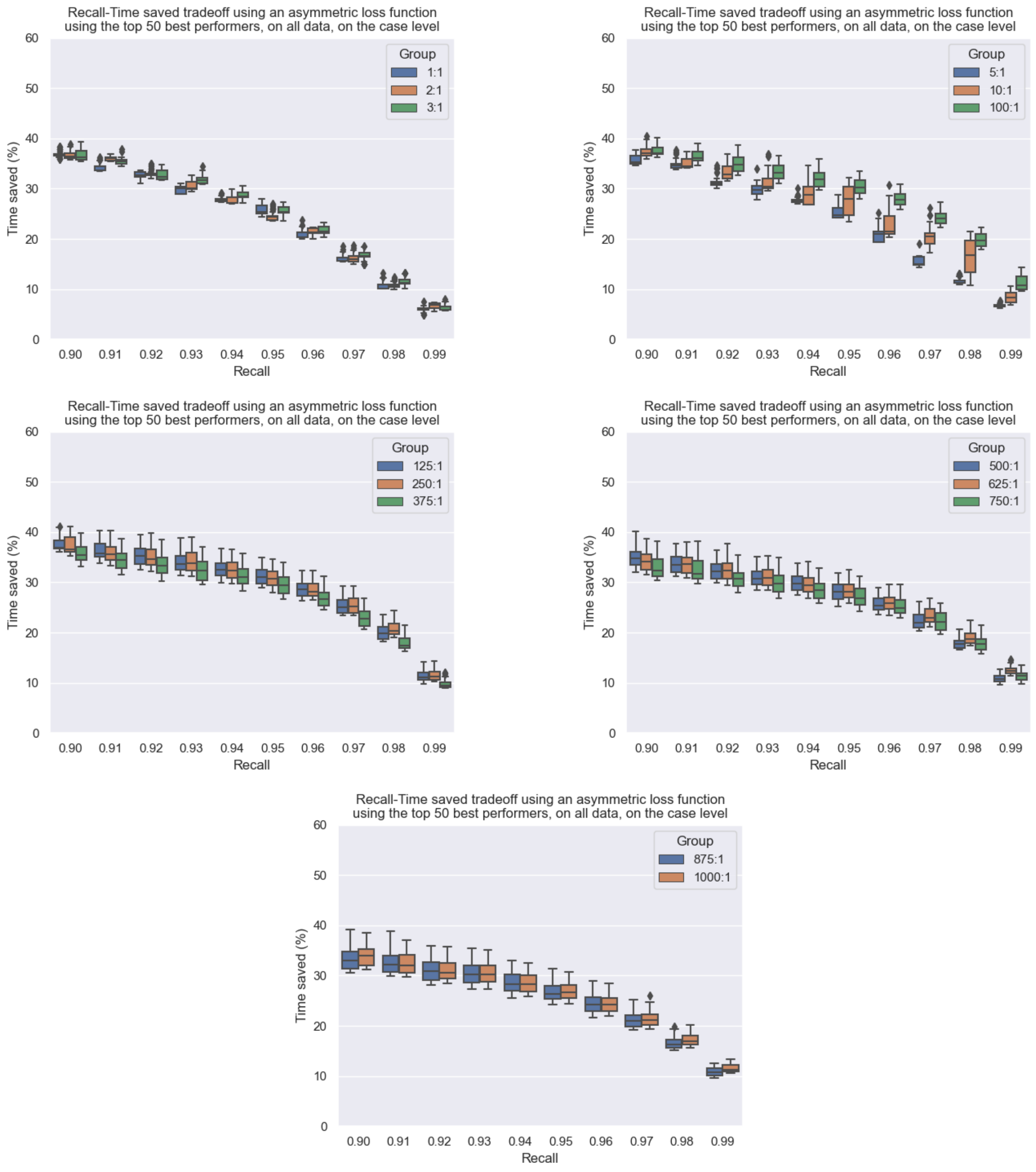


Figure 24: Multiple models were fitted on the test case dataset using all features. The models have been trained using various ratios between loss for overestimation and underestimation.

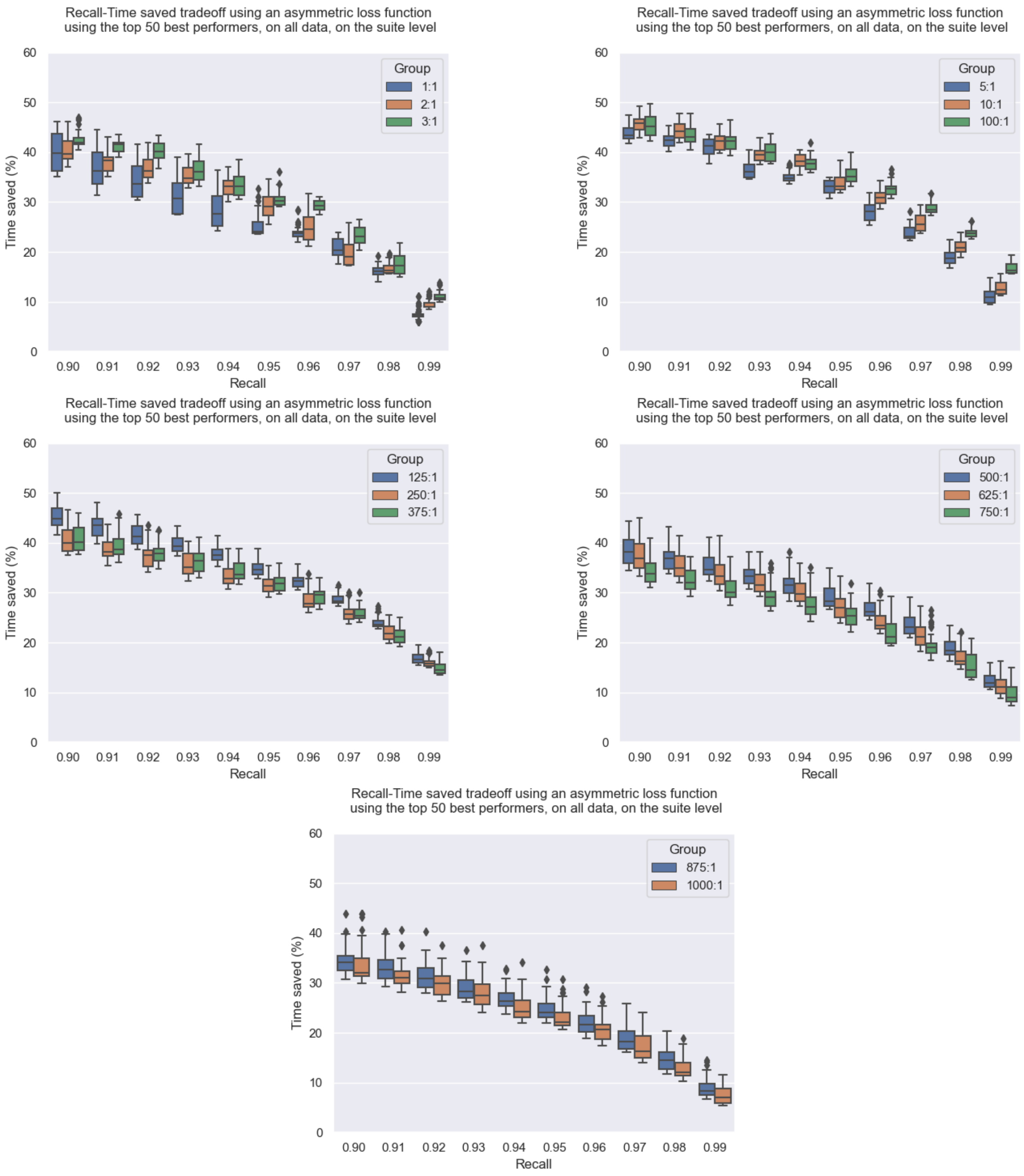
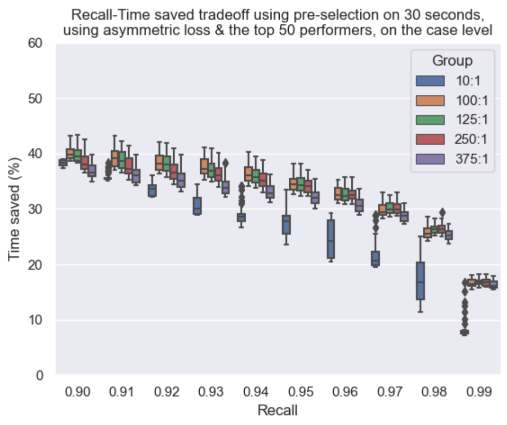
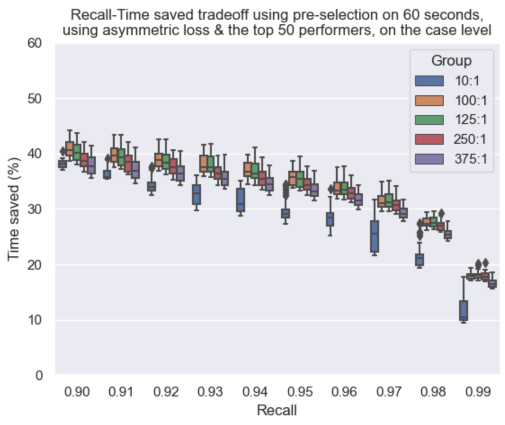


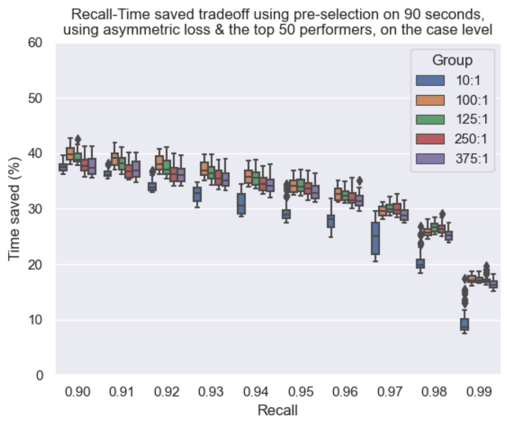
Figure 25: Multiple models were fitted on the test suite dataset using all features. The models have been trained using various ratios between loss for overestimation and underestimation.



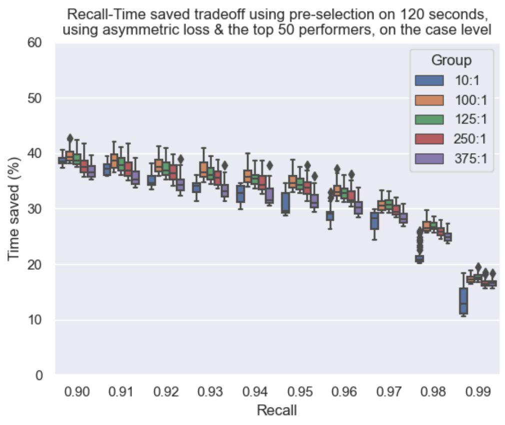
(a) 30s



(b) 60s

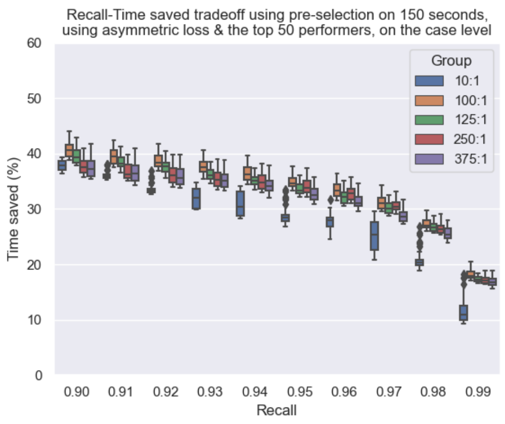


(c) 90s

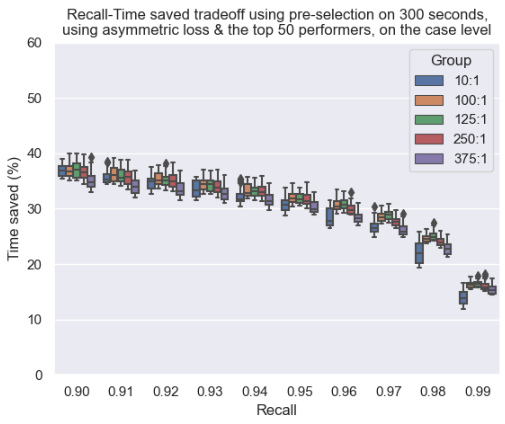


(d) 120s

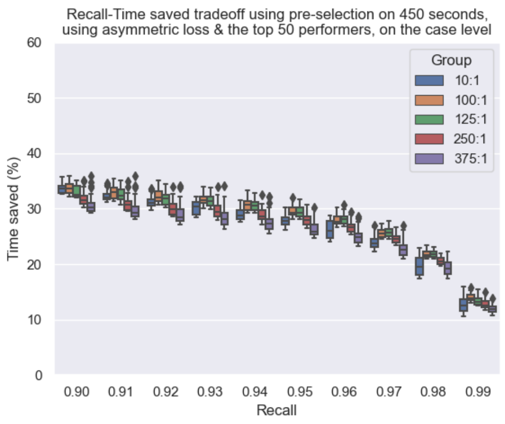
Figure 26: Multiple models were fitted on the test case dataset using all features. The models have been trained using various ratios between loss for overestimation and underestimation, as well as various pre-selection splitting times.



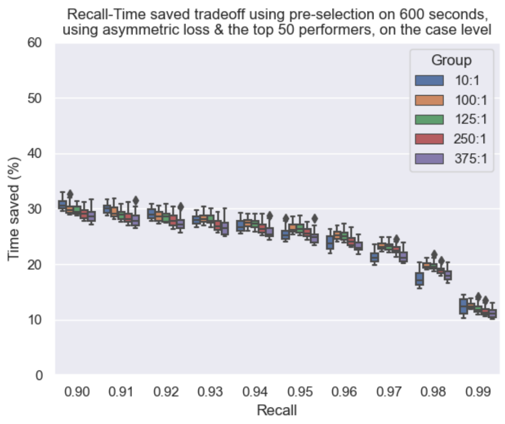
(a) 150s



(b) 300s

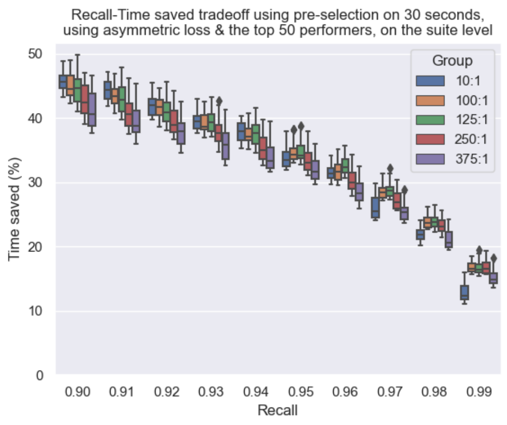


(c) 450s

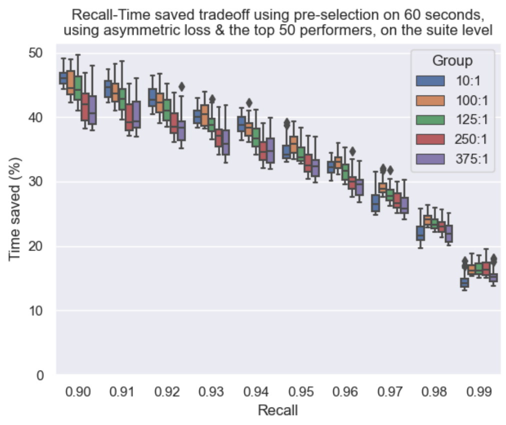


(d) 600s

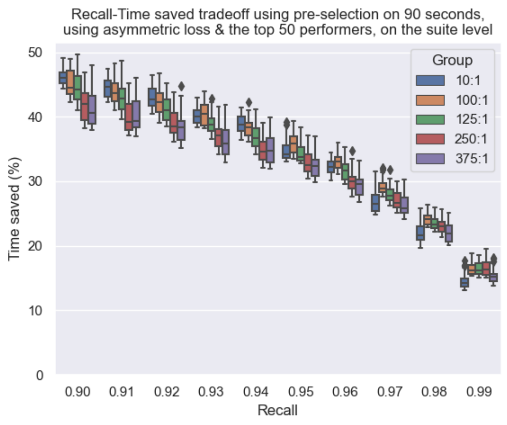
Figure 27: Multiple models were fitted on the test case dataset using all features. The models have been trained using various ratios between loss for overestimation and underestimation, as well as various pre-selection splitting times.



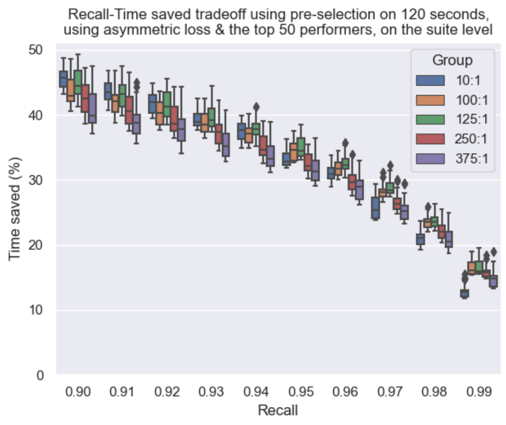
(a) 30s



(b) 60s

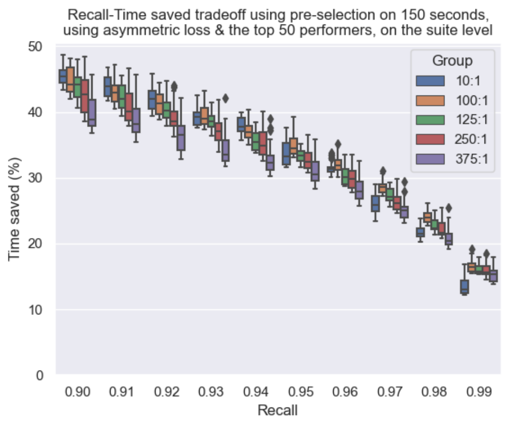


(c) 90s

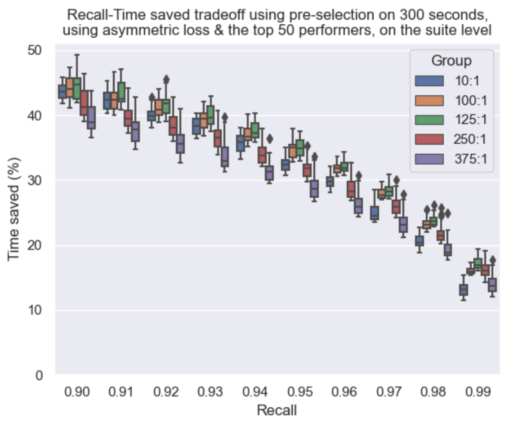


(d) 120s

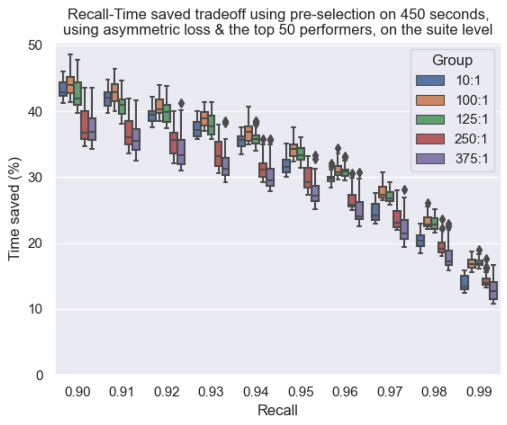
Figure 28: Multiple models were fitted on the test suite dataset using all features. The models have been trained using various ratios between loss for overestimation and underestimation, as well as various pre-selection splitting times.



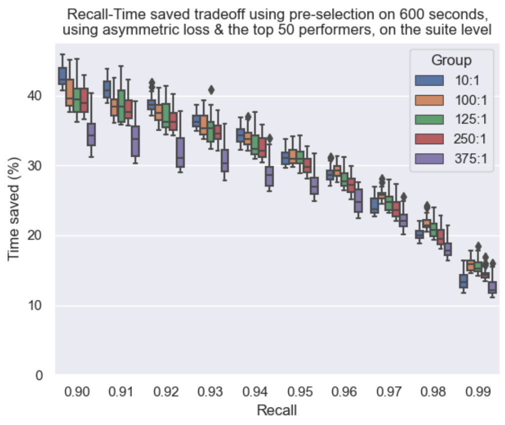
(a) 150s



(b) 300s



(c) 450s



(d) 600s

Figure 29: Multiple models were fitted on the test suite dataset using all features. The models have been trained using various ratios between loss for overestimation and underestimation, as well as various pre-selection splitting times.