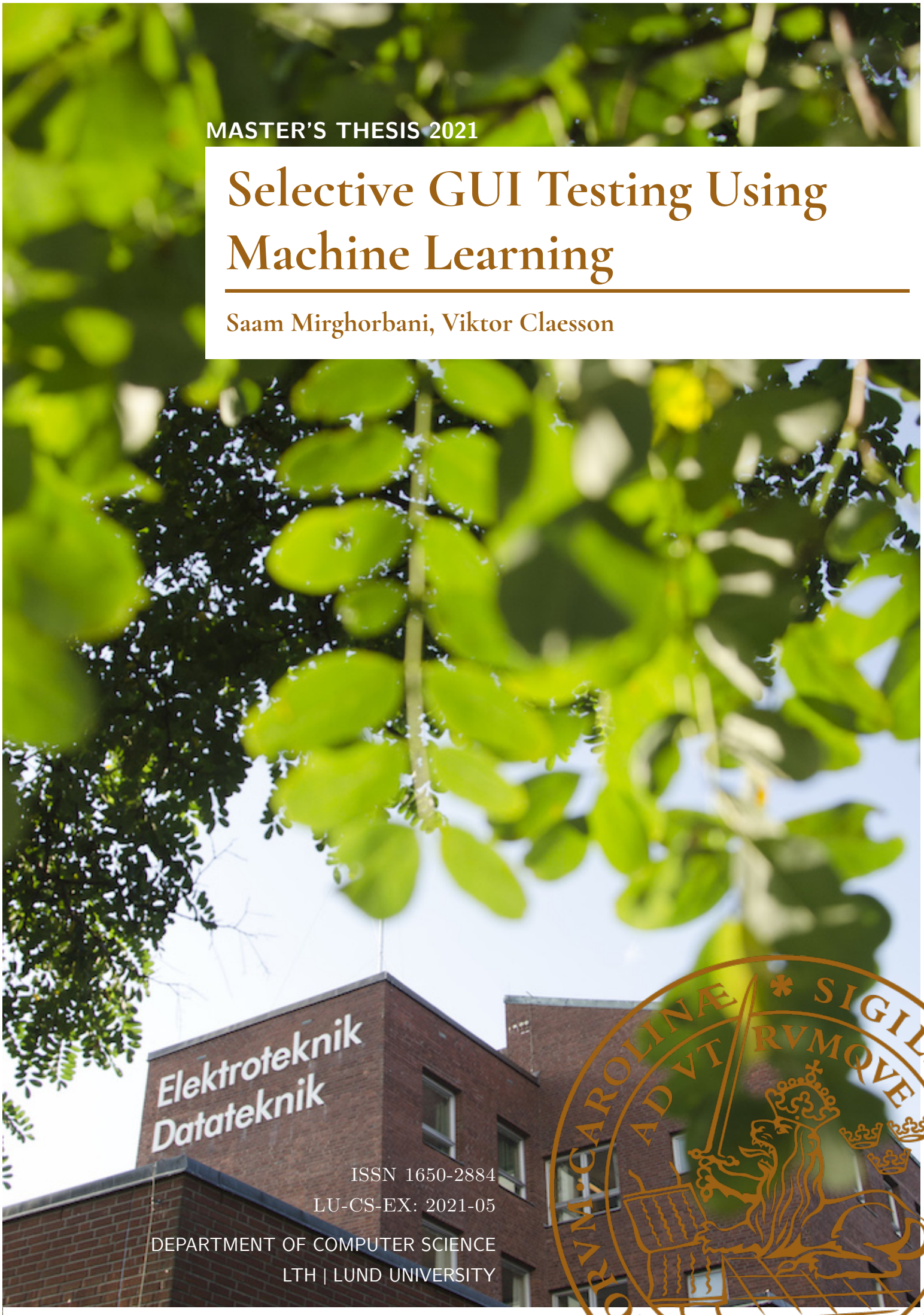


MASTER'S THESIS 2021

Selective GUI Testing Using Machine Learning

Saam Mirghorbani, Viktor Claesson

ISSN 1650-2884
LU-CS-EX: 2021-05

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-05

Selective GUI Testing Using Machine Learning

Selektiv GUI Testning med hjälp av Maskininlärning

Saam Mirghorbani, Viktor Claesson

Selective GUI Testing Using Machine Learning

Saam Mirghorbani
mat14smi@student.lu.se

Viktor Claesson
dat15vcl@student.lu.se

March 25, 2021

Master's thesis work carried out at Axis Communications AB.

Supervisors: Emelie Engström, emelie.engstrom@cs.lth.se
Linus Lindgren, Linus.Lindgren@axis.com
Magnus Walter, Magnus.Walter@axis.com

Examiner: Ulf Asklund, ulf.asklund@cs.lth.se

Abstract

Regression testing is performed to minimize the risk of changes breaking existing functionality in software. To reduce the time of testing, Regression Test Selection (RTS) strategies aim to select a subset of only the affected tests. However, in projects with frequent changes it is sometimes necessary to sacrifice some coverage for even shorter time of testing to receive feedback for every change. The objective of this thesis is to explore how machine learning can be applied as an automated RTS strategy for GUI testing. In this report, we design and evaluate multiple models using historical data from a project consisting of 60,777 data points, including 3,808 code changes and 35 unique test cases. We evaluate how effective different variables in the data are at indicating if a test will fail, and how well the model scales with the size of the project. We found that our best model is able to outperform our best heuristic, and can be used to select a trade off between coverage and time. When highly favoring coverage, it is able to reduce testing times by 52% while ensuring that 91% of failing tests are predicted.

Keywords: Regression Testing, Regression Test Selection, Machine Learning, Test Automation, Selective Testing, GUI Testing

Acknowledgements

We would like to thank Axis Communications AB for providing us with the opportunity and environment to realize this project.

Additionally, we want to thank our supervisor at Lund University, Emelie Engström for giving us invaluable feedback and suggestions throughout our thesis work.

Special thanks to our supervisors at Axis Communications, Linus Lindgren and Magnus Walter, for providing us with guidance and ideas to finalize this project. Finally we would like to thank Ola Söder, also at Axis, for his insight and contributions.

Contents

Chapter 1
Introduction

Today regression testing has become an essential part of any software project for maintaining quality. Regression testing is an activity performed in order to ensure that changes in software do not break existing functionality. As software projects continue to grow there has been an increasing need to optimize this process by selecting a subset of tests that only test the parts of a system that are most likely affected by the change. This is referred to as Regression Test Selection (RTS) and can be achieved using a wide variety of strategies.

This thesis work was conducted at Axis Communications mobile applications department where graphical user interfaces (GUIs) are a main target for automated regression testing. Axis would like to improve upon their automated GUI testing process by providing faster and more accurate test results during development. A dynamic test selection strategy would achieve this by aiming to select only the most relevant test cases for each change. Additionally most of the changes do not introduce any regressions at all. In this case it would save a great deal of development time when the test suite is not being run unnecessarily, since there is some overhead time associated with building the application for GUI-testing.

In this report we investigate how to improve the capturing of failures while limiting the time of testing during development by applying machine learning (ML) on historical test data for GUI tests. Machine learning for RTS is a rather new research topic but has shown positive results in newly published sources [8, 9]. In conforming with the agile development process at Axis, we focus on minimising time-consuming human intervention of finding mappings between GUI tests and program code. We do this by designing a system that automatically finds these mappings by using historical code changes and test case outcomes. As finding these mappings manually is a time consuming task, we perform a high-level analysis and use machine learning to automate the task of finding correlations between changes and GUI test outcomes. To measure the effectiveness of our solution we benchmark against other history-based heuristics.

Our contributions to RTS research revolve around GUI testing, however the findings could potentially be extended to any type of abstract testing. We present and evaluate factors that influence outcomes of tests that are too general and broad in their coverage to be

mapped to specific functions in the source code of an application. We describe our method for creating our solution and present results of an empirical evaluation on various history-based heuristics for automated GUI test selection. Additionally, we evaluate how the effectiveness of algorithms based on machine learning for GUI test selection scales with the available change history and test data for an industry application.

Chapter 2
Background and Related Work

The aim of the following chapter is to provide the reader with necessary background information to understand the theoretical aspects of the presented solution. We begin by describing the problem and research questions, followed by a presentation of related work. Then follows an overview of machine learning and the algorithms that we have selected.

2.1 Problem Description

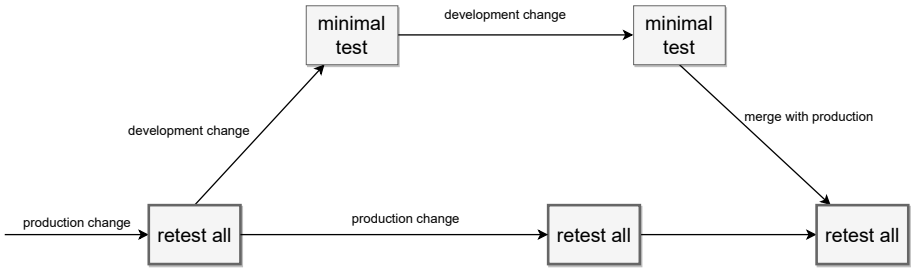


Figure 2.1: General example of the GUI testing strategy at Axis Mobile Apps.

The automated regression testing techniques used by Axis Mobile Apps for their GUI testing are *minimal testing* and *retest all*. The former is used as changes are continuously submitted for review towards their feature development branches, and the latter for deploying a finished feature to production, as depicted in Figure 2.1. The changes made on the development branches have to pass a few quality checks before being accepted. However, running the entire test suite on each change on a development branch would take too long. Axis has therefore opted to only test the most critical functionality when submitting feature-development changes. This ensures that the critical functionality still functions after every code change, but it could possibly miss other functionality. A more substantial testing is then performed

prior to merging a feature into the production branch, after which it should be ready for release. Hence Axis has chosen to run fewer tests during development to improve the developer experience.

Axis would like a more intelligent selection of GUI tests during development. A selection strategy that recommends test cases to run based on the code change and prior test outcomes. Due to time constraints during development, a safe RTS strategy that ensures that all affected tests are run is not ideal. Instead, Axis wants to be able to sacrifice coverage for time during development, since they are able to use the *retest all* strategy at a later stage. At the same time, they want to ensure that the coverage is as precise as possible.

One aspect that enables machine learning is that Axis uses a distributed version control system (DVCS) [2] to organize their project. A DVCS tracks all changes made to a project and has the ability to recreate old instances. From an old instance it is then possible to generate historical data by running tests and seeing what the most recent change was. Using the test results as labels then provides the opportunity of using supervised machine learning, explained further in Section 2.4.

The amount of available data is limited by how long the GUI tests have been used in the project and the size of the test suite. A known limitation of machine learning is that it requires a significant amount of data for training, even more so in the sub field of deep learning [12]. This puts limitations on which types of algorithms that can be used.

2.2 Research Questions

This thesis aims to investigate machine learning as an approach to RTS. We evaluate several algorithms and their selection performance for the GUI test suite of a mobile application.

In this report we also study how various information based on software changes and historical test case data affect the test outcome prediction capability of machine learning algorithms. Our intention is to find factors that can positively influence test outcome predictions, in addition to being general enough to be used for other projects that aim to apply machine learning for RTS.

Machine learning performs poorly if there is not enough data to use for training the algorithm. It is therefore of interest to investigate if a software application has enough data on change history and tests to be able to use machine learning to make satisfactory predictions.

Consequently, we formulate the following set of research questions that we will focus on answering in this thesis:

RQ1 **Selection performance** - How well do different machine learning algorithms perform at predicting tests that will fail based on information from code changes and test history, compared to available heuristics?

RQ2 **Feature importance** - What kind of information from code changes and test history is important for determining whether a test will fail?

RQ3 **Scalability** - How does the performance of machine learning algorithms for RTS on GUIs scale with the amount of available code changes and test cases?

2.3 Previous Work in History-Based RTS

Prior research has been made into using machine learning for RTS. Machalica et al. developed a gradient boosted decision tree classifier trained on a large data set of historical test outcomes [9]. Their model managed to reduce the total testing time by 50% while capturing 95% of test failures. This paper was one of our main sources of inspiration for this project. In addition to trying other machine learning algorithms, we also wanted to build upon their work by researching more factors that could influence the prediction of a test failure.

Lundsten developed a predictive test selection system, called EALRTS, that instead of relying on historical code defects introduced artificially generated mutations in the code to achieve similar results [8]. Using a Random Forest algorithm, EALRTS managed to reduce the number of tests selected by 60.3% while finding 95% of all failed tests. Since we had access to real data with code defects, this approach was not as crucial for us to take. However, we were interested in introducing artificial code defects to balance training data since our data was over-represented by passing tests. Introducing artificial data connected to test failures might have improved the prediction performance of our algorithms, but eventually we abandoned the idea due to time constraints.

One history-based approach consisting of creating links between source entities and tests was explored by Mahmoud and Hellgren [10]. It would construct a database of such links by analyzing test results. Then, when a change was submitted the system would query the database for each edited source entity, receiving a partial test suite to run. The regression testing was done for an android solution, using a test suite of over 68,000 tests. They were able to achieve up to a 99% reduction while ensuring a large portion of the failures were selected.

Another approach to history-based test selection was developed by Ekelund [4]. He created the Difference Engine, a software that produces a weighted correlation between code packages and test cases, after analyzing historical test data. These correlation values would then be used in conjunction with new changes to one or several code packages to select a subset of the test suite. On average the difference engine would recommend only 5% of the total test suite while its median and mean recall was 100% and 80% respectively. A major difference in this project compared to ours was the amount of available data. The project Ekelund based his work on had 1265 unique tests, for a large code base. He states that the database on changes was over 100 gigabytes, which is an average of 81 megabytes of information per test. In contrast, our project had 35 tests and only 4 megabytes of information per test. Additionally, the amount of correlation between tests and packages influences the performance. However, it is difficult to compare this aspect of Ekelund's project to ours.

2.4 Machine Learning

As early as in 1959, machine learning was defined by Arthur Samuel as a "field of study that gives computers the ability to learn without being explicitly programmed." [19]. In recent years it has become a hot topic of research, in large part due to the digital revolution which has enabled the computation and storing of vast amounts of data. In this section we describe different types of learning and algorithms that we used.

2.4.1 Supervised and Unsupervised Learning

Supervised learning is a branch within machine learning that deals with algorithms that map inputs to outputs, where the output is manually labeled [12]. Data is usually divided into two sets, one for training and one for testing the algorithm. The algorithms learn by training a model on input data and labeled output data, thereby tuning themselves to predict the output given the inputs. The model is then tested on a hold-out data set where it tries to predict the output.

Supervised learning can either be described as a regression or a classification problem. *Regression problems* involve problems where input variables are used to estimate a continuous output. An example of a regression problem is the task of predicting house values. *Classification problems* involve problems where input variables are used to predict a discrete output. In a multi-class classification problem, the goal is to predict one of several categories. If there are only two classes, it is called a binary classification problem. This is the type of problem that we are trying to solve in this thesis using algorithms to predict the classes *ignore test* and *select test*.

Unsupervised learning is a branch of machine learning that in contrast to supervised learning uses unlabeled data to find patterns [12]. It is commonly used as a means of clustering data, when the clusters (or labels) are not known in advance. Clustering algorithms use information from the data to categorize data points into how similar they are to each other.

2.4.2 Tree-Based Algorithms

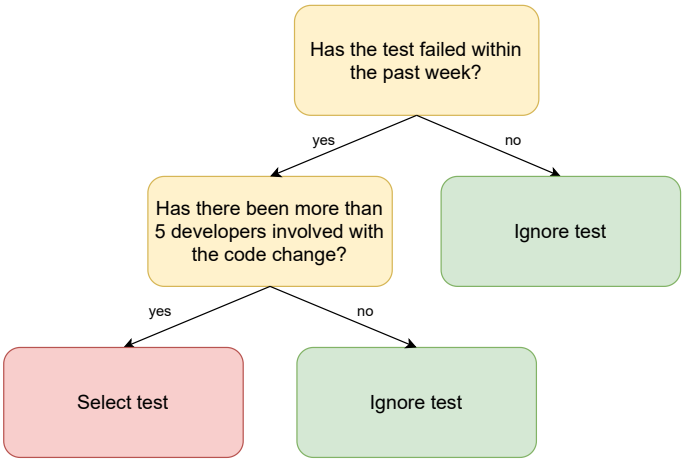


Figure 2.2: Abstraction of two splits in a decision tree, with leaf nodes representing the predicted classes.

The supervised machine learning algorithms we have chosen for this thesis are mainly tree-based algorithms. In contrast to many other machine learning algorithms, they do not require normalization of inputs and do not have their prediction performance affected by variables that bring no valuable information. They are also easy to interpret and therefore facilitate analysis of feature importance which we focus on with our research question **RQ2**.

The simple form of tree-based algorithm is a decision tree. A decision tree is built up by many nodes, where each node acts as a split on the data, dividing the data into two new

data sets on some criterion of a feature. The algorithm evaluates which splits are the most effective and places them closer to the root. The leaf nodes represent the predicted classes, as depicted in Figure 2.2. Essentially the tree divides the data into different classifications depending on many different features.

Random forest is another tree-based algorithm which consists of multiple decision trees, each trained on a random sample from the data set. Because it is composed of multiple different models, it is referred to as an ensemble algorithm. More specifically, it is a bagging algorithm. This means that each model makes its own prediction and votes for the class that it predicted. The majority vote becomes the output of the algorithm.

An alternative ensemble approach to bagging algorithms are boosting algorithms. They also consist of multiple models, with the difference that they are sequentially executed and attempt to correct the previous model's prediction error. One of the most popular boosting tree algorithms is eXtreme Gradient Boosting (XGBoost).

2.4.3 Support Vector Machines

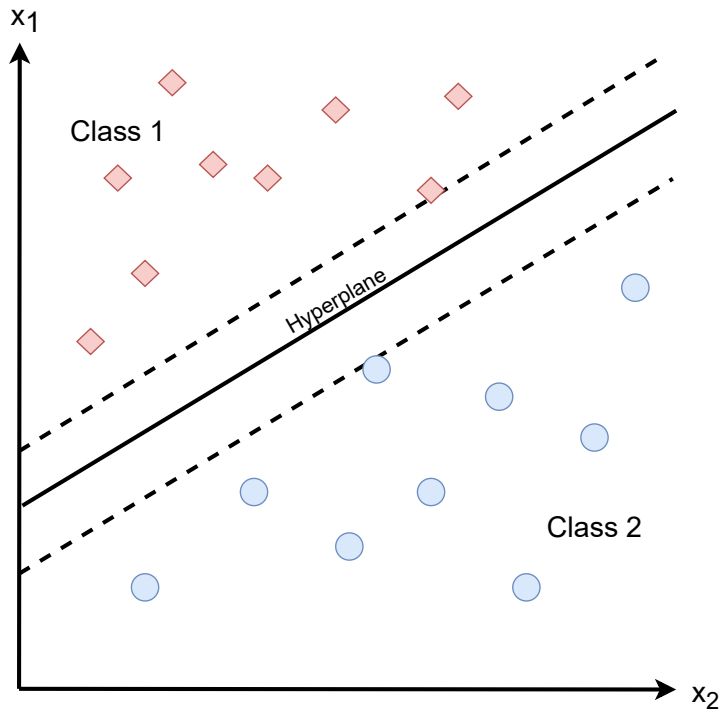


Figure 2.3: Example of a hyperplane dividing two classes in a two-dimensional space.

Support vector machines are supervised learning models that can be used for both classification and regression problems. The basic principle of a Support Vector Machine is to find a hyperplane that best divides the data into different regions, as the example shown in Figure 2.3. It does so by attempting to maximize the distance between the hyperplane and the closest point in the data. Support Vector Machines have been proven to work well with high-dimensional data [15], and was therefore a choice of algorithm to consider, due to the many features that we generated from our data.

Chapter 3
Methodology

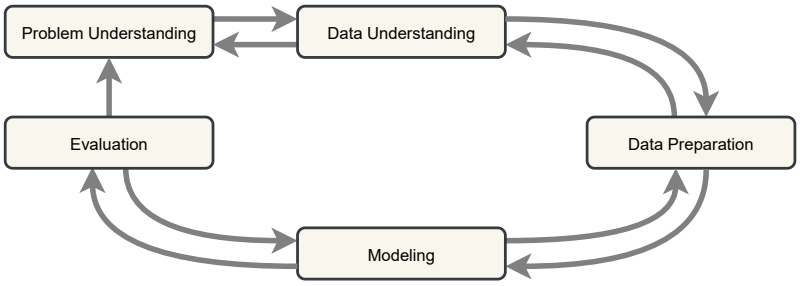


Figure 3.1: The iterative workflow during the thesis.

This chapter describes tasks performed during the thesis work. These phases are part of a process related to both research and design, shown in Figure 3.1, which is inspired by the CRISP-DM framework for data science [3]. The process is iterative and starts with the problem understanding phase.

3.1 Problem Understanding

In the problem understanding phase the focus is to explore the stakeholder requirements and how they can be translated into a machine learning context. This phase involves studying, planning and holding discussions with stakeholders about requirements and different approaches to take. This phase is constantly revisited due to the iterative nature of our work.

3.1.1 Literature Search

We initially performed a literature search to position ourselves within the field of regression testing and machine learning. We began by querying the web for articles related to machine

learning applied to RTS, using tools such as LUBsearch, Google Search and Google Scholar. We used variants of the keywords regression test selection, machine learning, predictive test selection and regression testing. From the initial search we then filtered for relevant articles and continued by searching for additional references within them. For certain articles [9, 13] we looked for metrics related to change impact in order to support our decisions for data collection and feature design. As a result we defined features that are presented in Section 3.2.2. Additionally, from some of the articles we found we used future work proposals as inspiration to what we could contribute with.

We soon realised that not much research had been done with applied machine learning for RTS. Therefore we expanded our search to include sources discussing other strategies for RTS, which we presented in Section 2.3. At this point we had already spent most of our time on a solution based on historical data. Hence we excluded most sources that dealt with other strategies that were not based on historical data, since they would have required a substantial turn in direction for us. In addition, we found that many of the methods proposed were either too specific for certain use cases or were based on stronger dependencies between code and tests than what we were able to get from the software application that we were designing our solution for.

A majority of the information search involved understanding concepts of applied machine learning. Since many topics are new and experimental, some of the information could only be found in non-peer-reviewed articles. In these cases we compared different articles and made experiments of our own to verify their correctness.

3.1.2 Discussions With Engineers

Weekly meetings were held with our supervisors; the department test lead and a developer with machine learning expertise. During these meetings we discussed our findings, obstacles we faced and approaches that we could take. We maintained a backlog of tasks that we updated after each meeting, based on input from all parties. Important design decisions are presented in the subsequent sections of this chapter.

An unstructured interview was held with an expert engineer at Axis within data management. He had been involved in a similar project at another department, where a machine learning strategy for test selection was attempted. We discussed different circumstances at our departments and how we could benefit from what they had learnt about applying RTS. Topics such as how to treat unbalanced data and introducing synthetic data were also discussed. The discussions resulted in the *Locality Strategy*, used as a heuristic to evaluate our solution. The strategy is further explained in Section 3.5.1 and discussed in Section 5.4.

3.2 Data Understanding

In the data understanding phase we focused on collecting data and verifying data quality. Initially during the thesis work there was no data to begin with. Therefore it was important to define, early on, what kind of data that needed to be collected and how it was to be acquired. It was also important to collect everything we could possibly need, so that the collection process did not have to be repeated multiple times. While we did not have to collect all of the data more than once, there were several instances where we had to recollect some of

it due to ambiguous results and compatibility issues with older versions. In the following subsections we describe how we decided what type of data to collect and how to collect it.

3.2.1 Analyzing the Integration Chain

For this activity our focus was to explore what type of data that could be extracted from the continuous integration process at Axis relating to code changes and test outcomes. First, we analyzed what information that could be obtained from a code change by using the DVCS of the project. Second, we studied the build automation tool to understand how it saved data for a completed test run. Finally, we analysed the test code to see what information could be extracted from it.

We introduced a set of features that could be obtained from the data and that could be used for training a prediction model. Since we required our model to handle previously unseen code changes, we needed the data that the model was being trained on to be generalized well enough so that the model could infer if a new code change and a specified test have the conditions for failing a test. Data can be generalized by abstracting it from its highly detailed state into features. These features are presented and further explained in Section 3.2.2.

By using these features in a prediction model we could evaluate their impact on the prediction result using various methods presented in Section 3.5.3. Analyzing the results of these methods helped with answering **RQ2** about which information that affects the outcome of a test after a code change.

3.2.2 Defining features

Through analyzing the available data that could be mined from one of Axis's repositories, consulting with engineers at Axis and searching for examples in literature, we compiled a list of features that we hypothesized as relevant for predicting the outcome of a test from a code change. They were features that either relate to a code change, a test or a relationship between a code change and a test (cross features).

Features on a Code Change Level

- *Number of rows added/removed*: A code change including more lines of code introduces more areas where an error can occur.
- *Number of rows in included files*: As the lines of code increases in a file, so does the amount of variables and functions. Increasing the complexity of the file and the amount of places where an error can occur.
- *Number of files changed*: A code change including more files encompasses more areas of the source code, and in turn more places where an error can occur.
- *File extensions*: Some file types are more prone to breakages than others [13].
- *Author who made the change*: Code quality can differ between authors, and some authors work in areas of the code that could be more prone to bugs than others.

- *Number of files modified by multiple authors*: Files modified by 3 or more developers have a higher risk of breakage as opposed to files modified by 2 developers [13].
- *Amount of revisions to included files (over different periods)*: Files modified more frequently often cause breakages [13].
- *File is included or excluded from change*: Changes where the same files are changed should affect similar tests.
- *Number of changed files in different directories*: Some directories affect test code more than others.

Features on a Test Level

- *Historical failure rate*: Historical failure rate for a test is a good indicator for its probability of failing again [9].
- *Historical run time*: A test that has a longer run time generally has a higher complexity, and in turn more steps where it could be prone to failures.
- *Time since updated*: When new features are introduced, so are new tests that verify their functionality. Since these features are new, development tends to focus on them.
- *Test group based on failure occurrences*: Tests that usually fail together indicate that they test a common area. A grouping of tests based on their tendency to fail together could be helpful in that it uses information about other tests to infer the result of a test.

Cross Features

- *String similarity between paths of included files and test path*: Files that have words in either their path that are similar to words in the path of a test should indicate some connection [9]. For example, if a test has the path `app/test/testNavigation.kt` then a file that has the path `app/src/navigation/Navigation.kt` should have a stronger similarity than a file with the path `app/src/support/Feedback.kt`.

We aggregated all of the aforementioned features into a full list of features used as input to the algorithms that can be found in Table A.1.

3.2.3 Data Collection

Data collection consisted of two parts, collecting the data for a test run and collecting the information of a software change. This was possible because of the use of a DVCS, which enabled us to reconstruct old states of the project. From the reconstructed state a build automation tool could then be used in conjunction with the DVCS to gather the information we needed.

First, the information from the test run was collected by scraping result files generated by the build automation tool after running all the tests. In the output file we found the test name, package, arguments passed, result of the test and the run time. One issue encountered

was that sometimes tests would fail independently to the code change. A test that is non-deterministic, giving both a pass and a fail as a result when run multiple times, is called a flaky test. To deal with flakiness we opted to rerun all tests that failed up to a total of five times. To further ensure that the results were due to the code change and not other factors, we also opted to rerun changes that included any failures the next day. In this way any temporary issues such as network problems could be excluded.

Second, we used the DVCS tool to collect data about the software code change. The data included an identifier for the change, information on which the previous change was, date of change, historical file changes and information on contributors. In addition to the mentioned data the DVCS could also be used to estimate when the most recent change was made to each test, which was used to calculate the age of the tests relative to the change.

By far the most time consuming task when collecting the data was validating that it was correctly labeled. When reconstructing old states of a project there was a risk that old working code might not work in a more recent environment. To combat this we continuously monitored the results from the test runs to spot any indication of non-change test breakage issues. Some issues included altering the testing environment, the old project state not being compatible with newer versions of the android operating system on the emulators and the scraping tools we developed needed to be updated for old refactorizations of the project. When such a situation occurred we solved the issue and reran the test runs that had been affected.

3.3 Data Preparation

Data preparation and feature extraction are important aspects of data science. In this section we discuss the several steps we took to process the collected raw data into features. These steps resulted in the features that can be found in Section 3.2.2.

3.3.1 Cleaning Data

As discussed in Chapter 3.2.3, if a test failed we reran it multiple times. We also reran whole test suites for changes that resulted in failed tests. Consequently this meant that we had duplicated data. Most of the information was deterministic and did not change from run to run, so there was no need to chose which duplicate to discard. The exceptions were the test result and runtime. If a test had passed for a change in any run, we labeled it as passed and took the average runtime for all of its passes for that change. Otherwise it was labeled a failure and we took the average of its failed runs.

When running the tests we always ran all the test cases available at that project state. However, this was not the case during development where only the minimal test suite would run. Therefore test failures could occur that were not fixed in the successive change. When this happened successive changes would include a failure without having caused it, which in turn would be erroneous data for the machine learning algorithm. Since we could not determine if it was a fail or pass we could not correct the data. Instead we solved it by removing any data points that failed both in the current change and the previous one.

After cleaning, the final data set contained 60,777 data points. These data points belonged to a total of 3,808 software changes, containing a total of 35 unique tests. In total 644

data points were labeled as failures, and conversely 60,133 data points were labeled as passes. All the results presented in Chapter 4 are based on this data set.

3.3.2 Creating Features

Many features did not need any more processing other than simple calculations, for example counting the number of files with a given extension. However other features required more sophisticated methods to be transformed into a state that could be used in a machine learning model. In this section we describe the techniques we used to implement features. The high level descriptions of the generated features here are found in Section 3.2.2.

Aggregating Number Series

For every file included in a change we had several metrics recorded, these were the number of contributors involved in the revisions of the file, revisions of the file, lines of code added, lines of code removed, and lines of code in the file. Since each change could include many files this meant that we got a series of numbers for each of those metrics. However we needed to represent these with a single value instead of a list of values. So for each such numeric series we aggregated it in several ways. We computed the sum, max, min, mean and variance of the series and used those values as features.

One-Hot Encoding

ChangeID	Author	TestGroup	…	TestFailure
413	john_doe	1		0
614	mary_sue	2		1
984	foo_bar	3		0

ChangeID	Author_john_doe	Author_mary_sue	Author_foo_bar	…	TestGroup_1	TestGroup_2	TestGroup_3	TestFailure
413	1	0	0		1	0	0	0
614	0	1	0		0	1	0	1
984	0	0	1		0	0	1	0

Figure 3.2: Categorical features before (top) and after (bottom) one-hot encoding.

Because some of the machine learning algorithms we use do not work with non-numerical values as features, we needed to convert text-based features to numerical values. A common method of doing this is to one-hot encode categorical variables [17]. Viewing our features in a tabular format, where rows represent a change and a test outcome and columns represent features, categorical features are transformed as shown in Figure 3.2.

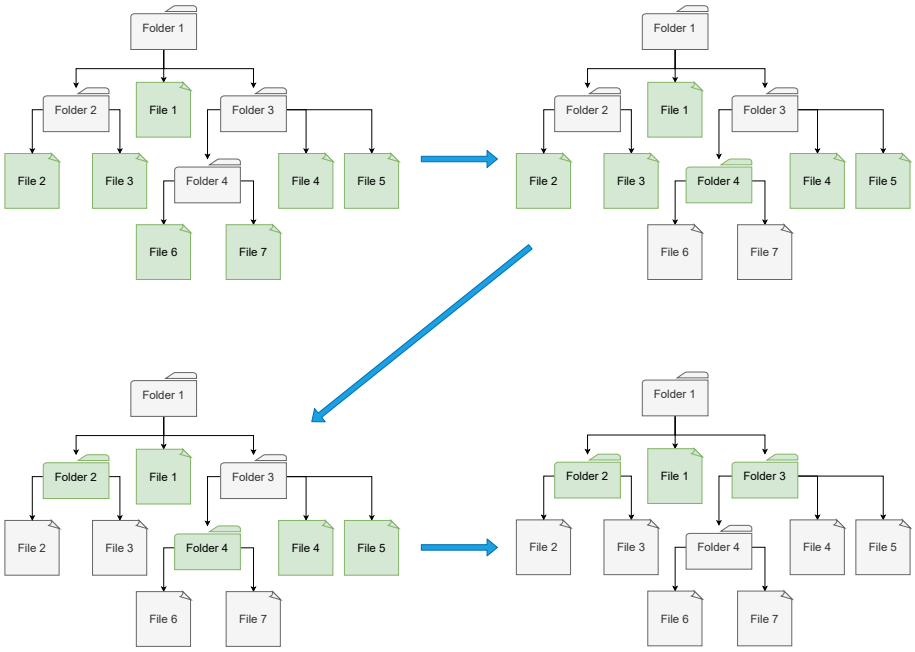


Figure 3.3: How the one-hot encoded file features (upper left) would be reduced down to 3 features (lower right) for an example project structure. The marked files and folders are items, and the percentage of files changed in those items are used as features for the model.

Encoding Changed Files

Initially we tried one-hot encoding all the files into features. However since the project includes many files it was not a sustainable solution since it produced well over 3000 features. To reduce the features needed to represent the files we instead represented it as the percentage of files edited inside an item, where an item could be either a single file or a folder.

An example project structure and how it can be reduced is shown in Figure 3.3. The method we used for reducing utilized that each file in the project could only be part of one item-feature, meaning if two items shared a common folder, that folder would not be an item. It would start by using all files as items, and then iteratively reduce the number of items until sufficiently few were left. At each reduction it would select the item that would reduce the total number of items the least, if there was a tie it would select the one at the lowest depth in the file tree. This ensured that the item-features always are of as similar size as possible.

Clustering Tests Into Groups

To create the test grouping feature from Section 3.2.2 we experimented with unsupervised learning. We chose to use the K-means algorithm to cluster test cases into groups based on their tendency to fail together, due to its simplicity and ability to scale with data. As features to the K-means algorithm we one-hot encoded all changes as described in Section 3.3.2. This meant that after one-hot encoding we had a categorical feature representing every change, as shown in Figure 3.4. Due to the high amount of changes, and thus dimensions, we decided to reduce the amount of features. This is because high dimensional data poses several challenges to clustering [20]. For this we used feature extraction with Multiple Correspondence Analysis

ChangeID	TestID	…	TestFailure
413	test_case_1		0
413	test_case_2		1
413	test_case_3		1
614	test_case_1		1
614	test_case_2		0
614	test_case_3		0

TestID	ChangeID_413	ChangeID_ 614	...
test_case_1	0	1	
test_case_2	1	0	
test_case_3	1	0	

Figure 3.4: Categorical features before (top) and after (bottom) one-hot encoding.

(MCA). MCA can be used to reduce the dimension of categorical features so that a high percentage of the original variation in the features can be explained in fewer dimensions [16]. When applying MCA we made sure that a number of features were extracted such that 99.9% of the variance in the original features was maintained.

When using the K-means algorithm, one has to specify the number of clusters beforehand. We ran the K-means algorithm for $K = 2$ up to $K = 35$ clusters. To select the optimal number of clusters we used the silhouette scores and the elbow method. A higher silhouette score indicates a greater separation between clusters [11]. The elbow method can be used to find an optimal K by analyzing the Sum of Square Errors (SSE) and selecting a K close to the "elbow" of the curve [23]. Using SSE and silhouette scores plotted in Figure 3.5, as well as manually analyzing if the clustering was reasonable, we selected K=17. This meant that we had 17 different groups that each of the 35 test cases were assigned to.

Similarity Between Test Name and Changed Files

To create a similarity score between a software change and a test case we used cosine similarity [21]. Cosine similarity is a measure of the similarity between text documents. It constructs

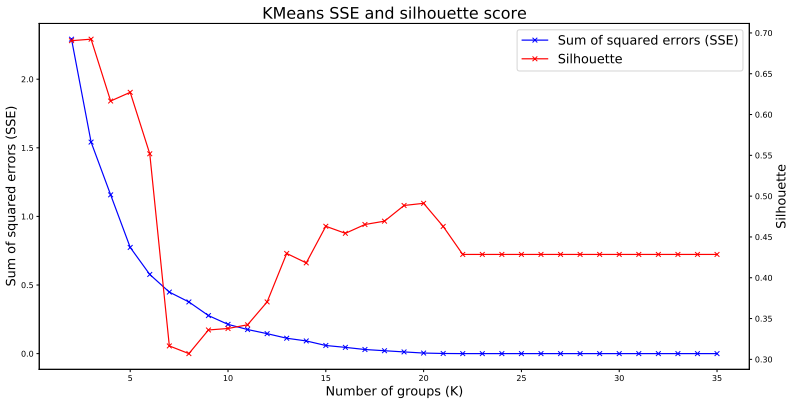


Figure 3.5: SSE and Silhouette scores plotted for each number of clusters K. The optimal cluster is chosen in regards to a high silhouette score as well as the elbow point of the SSE graph.

a similarity score by turning the documents into vectors and then evaluating how much the angle differs between the vectors. The documents are transformed into vectors by representing the count of each unique word on separate axes. We transformed the software change and test case into text documents by evaluating words present in the file paths and test name, then applied cosine similarity to get a score.

One disadvantage with cosine similarity is that every axis is orthogonal. While it is good for dissimilar words, synonyms and otherwise related words are treated equally distinct from each other. However, this is much more common in normal texts since you want to vary the language. In programming the language is less varied since it relies a lot on naming and keywords.

3.3.3 Feature Scaling

Feature scaling is an important step for support vector machines. Since the algorithm works with distances, a feature with a larger value would have a larger impact on the prediction. Therefore the features need to be scaled so that their values are in similar ranges. Feature scaling can be done with either normalization or standardization, depending on the distribution of feature values. Standardization is performed when the data is assumed to be normally distributed. This was not the case when we analyzed our data, so we decided upon using normalization. This meant that we transformed every feature to a range of $[0, 1]$. The transformation was done using equation 3.1.

$$x'_t = \frac{x_t - x_{min}}{x_{max} - x_{min}} \tag{3.1}$$

Where x_t is a specific value of x, x_{min} is the minimum value of all x, x_{max} is the maximum value of all x, and x'_t is the normalized value of x_t.

3.4 Modeling

In this chapter we describe how we select our models and how they are trained and tested on the available data. We used Decision Tree, Random Forest, XGBoost and Support Vector Machine during modeling, which are described in Section 2.4.

3.4.1 Hyperparameter Tuning

The machine learning algorithms we used utilize different types of parameters whose values control the learning process. These are called hyperparameters. As opposed to model parameters which are estimated from the data itself during training, hyperparameters are set manually before model training. To get an optimal performance from the model, hyperparameters are tuned to suit the problem. This is commonly achieved by trying different combinations of hyperparameters and measuring the model's prediction performance when using them to find the best hyperparameters suited to the problem.

The search for hyperparameters can be done in several ways. An exhaustive grid search uses all combinations of specified parameters to train a model and find the best combination. We used exhaustive grid search for algorithms that were fast to train and had few hyperparameters, since an exhaustive search will find the most optimal parameter combination from the specified values. For models that have many parameter combinations we used a randomized form of grid search, due to polynomial time complexity with an increasing number of hyperparameters to tune. Randomized grid search has the benefit of randomly sampling from the parameter combinations instead of trying every combination. For more accurate results we use K-fold cross validation with K=5 and K=3 for grid search and randomized grid search respectively. The process and benefits of K-fold cross validation are further explained in Section 3.4.2.

The grid search uses a scoring criteria to select the best hyperparameters. As scoring criteria for the grid search and the randomized grid search, we used the equation 3.2 for balanced accuracy score.

$$\text{Balanced accuracy} = \frac{\frac{tp}{tp+fn} + \frac{tn}{tn+fp}}{2} \tag{3.2}$$

Balanced accuracy was chosen as scoring criteria since it takes into account both positive and negative predictions and resulted in the highest trade off curve over *recall* and *time reduction*, explained in Section 3.5.2.

3.4.2 Cross Validation

When we evaluate a machine learning model, we need to make sure that the model is tested on previously unseen data as to simulate how it performs in a real setting. A common practise in the field of machine learning is to reserve a fraction of the data for testing, and train on the rest. This is referred to as a train-test split. A model becomes biased towards samples that it has been trained on, which means that it will score higher if it is evaluated on the same samples. Splitting the data set into training and testing can also reveal if the model is

overfitting. If the model scores higher on the training set than the testing set, it means that the model is not able to generalize well to unseen data, and is overfitting on the training data.

Splitting the data into one train and one test set is usually not enough. Especially if the data size is small or the distribution of classes are skewed. It might be the case that most of the valuable data, in our case data on test failures, happens to be in the test set which the model cannot use to learn. The opposite is also possible, failure samples in the test set may be so few that the measured performance will be influenced by chance, rather than showing the models actual performance.

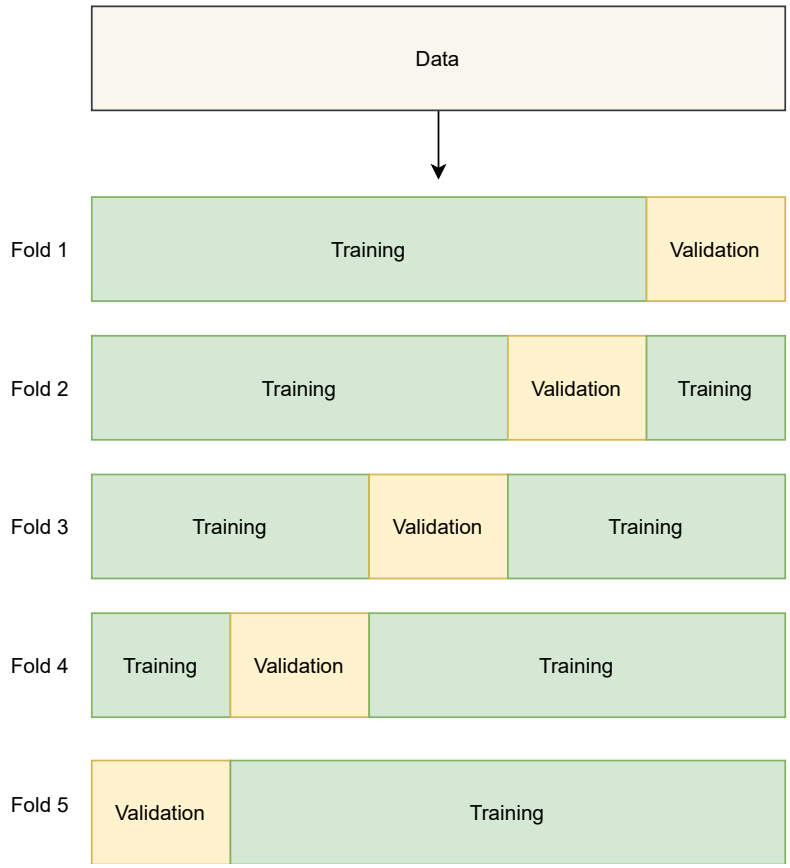


Figure 3.6: Example of 5-fold cross validation [1]

To mitigate this problem we evaluate our models using K-fold cross validation. With K-fold cross validation the data is divided into K splits where K-1 splits are used for training and 1 split for validation. A new model is trained K times until each split has been used for validation. We use 5-fold cross validation, as shown in Figure 3.6, which means 4 splits are used for training and 1 split for validation. We evaluate the performance by averaging over the validation score of each fold. Using cross validation we can also measure the consistency of the model. If performance varies greatly between folds, it could indicate that the algorithm has not learned from the data and is only guessing.

When splitting the data into either train and test or train and validation sets, we need to make sure that information from one does not leak into the other. If every data point in the data set consists of unique values this will not become a problem since the point can only be included in one of the splits. However, in our data set there are data points that partially contain duplicate information. Since the data set contains both test features and change

features, there are data points that may have duplicate change information or duplicate test information, but not both. The model is expected to know about the tests when making a prediction, but it will most likely not have seen the exact same code change before. Therefore we want to make sure that this is represented in the evaluation by only letting a change be present in either the training or testing split. This is demonstrated in Figure 3.7. Instead of simply splitting the data we assign each change an identifier, a random number between 0 and 1, and then we perform the split on the identifier.

While the data contains changes from both production and development branches, we only let the validation subset contain data from the development branches. We use both types of branches for training since we want the model to learn as much as possible from code changes. However, since the model is only going to be used for the development branches, we chose to mimic this scenario with the validation data by only using development branch changes for validation.

Additionally, to further simulate a real scenario when evaluating our models, we omit changes made by testers from the validation data. We took this measure since we are only interested in how the models perform when a developer makes a change. When the testers make a change it is usually to create or modify GUI tests and therefore they will make sure that all tests are passing before they submit the change. However, we keep changes made by both testers and developers in the training data since otherwise a system needs to be in place for keeping the model up to date with new employees and whether they are testers or not. Also, an added benefit is that it will give the model more data to learn from. As a result, our splits will not be divided perfectly into the fractions depicted in Figure 3.6, which is discussed later in Section 5.6.

ChangeID	Change feature 1	Change feature 2	Test feature 1	Test feature 2	Split identifier
1	555	222	4	6	0.4
1	555	222	5	6	0.4
1	555	222	6	7	0.4
2	666	777	6	4	0.7
2	666	777	5	7	0.7

Figure 3.7: Example data of two code changes. The split between training and testing data is made using the *split identifier*, such that rows marked in the figure with the same split identifiers are all put in the same set. This is so that information about a unique code change is not used for both training and validation, thus more accurately simulating unseen data in the validation set.

3.5 Evaluation

In this section we show how we evaluate our machine learning models and what measures we took to get a correct evaluation. We describe the different ways that we measure our models,

how we calculate feature importance and how we evaluate the scaling capability of models.

3.5.1 Heuristics

We defined several strategies as baselines to measure how well the machine learning models were performing. The ones compared were the following:

- Department current strategy
- Random selection strategy
- Locality strategy
- High risk test strategy

Department current strategy works by running a single test case that tests the most basic functionality. The test case can be summed up as a test that only navigates through the application's different views. If the mobile applications department at Axis is going to switch to a strategy based on machine learning, it needs to perform better than the current strategy. The current strategy therefore acts as a baseline in this thesis work.

Random selection strategy is a simple strategy with a probability p to select a test and a probability $1 - p$ to ignore it. We chose to include this strategy as it serves as an easy to implement, bare minimum and general solution. It has also been shown to outperform other test selection methods [5].

Locality strategy is based on the assumption that if a change causes a test to fail, the probability that the same test will fail again in the near future is high. The locality strategy works by computing for each test case in a change, how many seconds ago since the test had previously failed. The test case is then selected only if it has failed within a specified time period.

Finally, *High risk test strategy* works by calculating the historical failure rate for all tests and picking the ones with the highest failure rate and only running those.

3.5.2 Selection Metrics

Since it was most important for us to implement a test selection strategy that would miss as few test failures as possible, while at the same time reducing the execution time of the selected suite, we chose metrics based on these conditions that we evaluated our strategies on.

Every sample falls into one of the four following categories.

True Positives (tp), samples that were selected when they should be selected.

True Negatives (tn), samples that were ignored when they should be ignored.

False Positives (fp), samples that were selected when they should have been ignored.

False Negatives (fn), samples that were ignored when they should have been selected.

Recall, defined with equation 3.3, is a measure of how well the strategy includes the relevant samples. Relevant samples in our problem are the samples causing a test failure, while the non-relevant samples are those that did not contain any failures. Therefore, in order to make sure that as few test failures as possible are missed, we want to maximize True Positives while minimizing False Negatives, which is the case when the model has a high recall.

$$\text{Recall} = \frac{tp}{tp + fn} \tag{3.3}$$

Selectivity, defined with equation 3.4, is a measure of how well the model correctly ignores non-relevant samples. Since non-relevant samples are the ones without any failures, it is of importance to the overall reduction of the model to have a high *selectivity*. Together *recall* and *selectivity* are part of the definition for *balanced accuracy* in equation 3.2.

$$\text{Selectivity} = \frac{tn}{tn + fp} \tag{3.4}$$

Selection reduction, defined with equation 3.5, is the proportion of all the samples that the strategy points out as non-relevant, and therefore will not be selected. This metric is important as it gives a measure of by what magnitude that the size of the selected test suite can be reduced.

$$\text{Selection reduction} = \frac{tn + fn}{\text{all samples}} \tag{3.5}$$

Time reduction, defined with equation 3.6, is a variant of selection reduction that takes into account the average run time of every test.

$$\text{Time reduction} = 1 - \frac{\sum_s^{s \in D'} t_s}{\sum_a^{a \in D} t_a} \tag{3.6}$$

t, the average run time of a test.

D, the set of all samples.

D', the set of all selected samples.

A specification from Axis was that the trade off between coverage and time could be determined on a case by case basis. This specification essentially translates to a trade off between recall and time reduction. As such, we created multiple instances of all selection strategies for different values on recall and time reduction.

Trade offs between these two metrics were achieved in different ways depending on the strategy. In the case of the machine learning models, the trade off was tweaked by varying the class weight parameter to favor one class over the other. For the locality strategy, the trade off was achieved by varying the time interval for if a test had previously failed. The recall-time-reduction trade off for the high risk strategy was achieved by varying the amount of tests that were selected by descending order of historical failure rate.

3.5.3 Feature Importance

There are many approaches to evaluate feature importance and they each have their benefits and limitations. We used a combination of different approaches since we wanted the selected features to be as general as possible so that they can be used for other projects. The methods we used to evaluate feature importance were permutation importance, wrapper methods and embedded methods.

Permutation importance

Permutation importance is a widely used method for testing the importance of a feature. The basic intuition is to train a machine learning model using all features, permute a feature in the test set and measure the difference that the model produces in the prediction performance before and after permuting. Permuting a feature simply means to shuffle the order of all the values of that feature. This makes the feature useless in the sense that it does not add anything to the model's prediction capability. If the prediction performance on the hold-out test set decreases after permuting a feature, it is an indication that the permuted feature was important for the prediction capability of the model. It should be noted that different algorithms use features differently, which means that features may exhibit different permutation importance scores across different algorithms. For this reason, we evaluated the permutation importance of all features on multiple machine learning algorithms to see if we could find a pattern or generalize the result.

It is also important to note that permutation importance only takes into account the independent contribution of a feature. It may be the case that a feature is important only in the presence of another feature, therefore it cannot be guaranteed that features with high permutation importance are the only important features. For this reason we use permutation importance in combination with other feature selection methods such as wrapper methods. An advantage of using wrapper methods over permutation importance is that they take feature dependence into account [14].

Wrapper methods

There are three types of wrapper methods; forward selection, backward elimination and bi-directional elimination. Forward selection starts by using each feature individually to train one model each. The feature that resulted in the best prediction is kept and the process is repeated again with the remaining features. Backward elimination works in the same way with the difference that the first stage includes all features and they are eliminated one by one. Bi-directional elimination is a combination of both methods. Due to computation time and framework limitations, we only evaluated forward selection on our best performing model. The results are presented in Section 4.2.

Embedded methods

Embedded feature importance methods come from algorithms that have their own built-in feature selection methods. We used the embedded feature importance methods available from the frameworks for the tree-based algorithms that we used. In the case of Decision Tree and Random Forest the importance method used was Gini Importance, also called Mean

Decrease Impurity (MDI). MDI is measured by summing the number of splits that include the feature, in proportion to the number of samples that it splits [7]. In ensemble methods, the MDI importance for a feature is calculated across all trees. For XGBoost we used the average gain of splits which uses the feature, as this was the most similar to MDI out of the alternatives.

3.5.4 Scalability

To answer **RQ3** about how well the models scale with available data we performed two types of evaluations. Both evaluations were performed on three models, *XGBoost*, *Decision Tree*, and *Random Forest* to investigate any scalability differences between the models.

The first evaluation we performed was to limit the available data points in the data set. We evaluated the data set at 50 different evenly spaced sizes, going from 2% up to 100% of the data. For each such specified size we randomly took a subset of changes matching the size and filtered the data set to only contain those. Then for each such filtered data set we evaluated the model with a 5-fold cross validation as described in Section 3.4.2. This process was repeated five times for each size for a more accurate answer. Between each iteration the split identifier in the data set was reset to new random value for each change.

Next, we evaluated how the amount of tests in the test suite affected the model. We evaluated all possible sizes of test suites, from only one test up to all tests being included. For each size of test suite a random subset of the tests were selected and after filtering the data set to only contain those tests the model was evaluated. This process was repeated 20 times for each size, each time picking a new random subset of tests to include in the filtered data set.

Chapter 4
Results

This chapter includes all the results and findings from the thesis work.

4.1 Selection Performance

To measure selection performance, we evaluate each strategy using the metrics described in Section 3.5.2. Reduction or time reduction is plotted as a function of recall. The strategy with the best selection performance is the one that has the majority of its points above the other curves, as this indicates that it has the highest values achievable in regards to both metrics.

The results of evaluating each heuristic on the data set is presented in Figure 4.1. The departments current strategy does not include any levels of granularity, thus it is represented as a single point. The random selection strategy was not simulated. Instead it is theoretical recall and reduction when using p as a probability to select a test, where p is selected from the range $p \in [0, 1]$. Because we have a probability p of selecting a test, that means p of the test failures will be selected, giving us a recall of p. Conversely since p tests are selected, $1 - p$ are ignored, meaning reduction will be $1 - p$. This gives us the line equation 4.1.

$$reduction_{random} = 1 - recall_{random} \tag{4.1}$$

The locality strategy uses a different time window for the previous fail for each point, going from as low as possible to the highest possible. Finally the high risk test strategy is evaluated at each point by adding a test case to the selection strategy, starting with the test case that has the highest failure rate and continuing in order of decreasing failure rate.

From the Figure 4.1 we can clearly see that the *Locality Strategy* outperforms the other strategies by a large margin when reduction is more than 60%. When reduction is less than 60% the *Locality Strategy* and the *High Risk Test Strategy* have similar results.

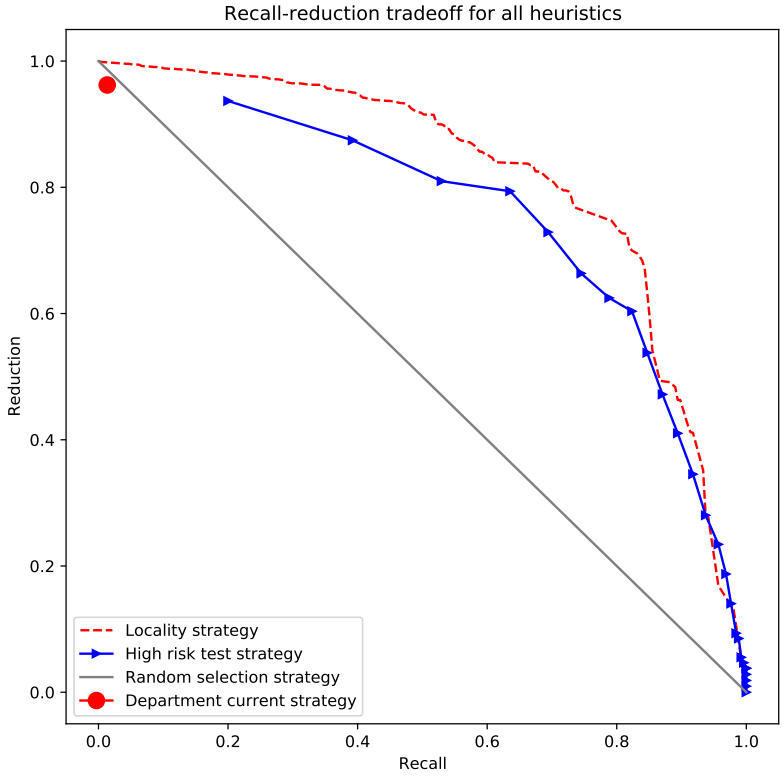


Figure 4.1: Trade off between recall and reduction for heuristics.

4.1.1 Machine Learning and Baselines

The best performing heuristic from Figure 4.1 and *department current strategy* were selected as baselines to be compared with each of the machine learning algorithms. Every algorithm is trained with 5-fold cross validation using the best hyperparameters that were found from Table A.1, and the resulting scores are averaged over each fold. To obtain different trade offs between recall and time reduction, each model is retrained using varying amounts of class weights. Weighting towards a class means that the algorithm will favor that class during training. The results are then plotted against each other as seen in Figure 4.2.

Method	Recall	Time reduction	Changes without tests	Overhead (415 changes)
XGBoost	52.9%	96.9%	71.2%	6 h
Current	2.93%	94.7%	0%	21 h

Table 4.1: Comparison between XGBoost and department current strategy for similar time reductions (similar testing time). XGBoost recommends to run 0 tests in 71% of changes, which reduces the total overhead time of 415 changes to 6h instead of 21h when always running tests.

The overhead is the time that is required to run GUI tests, excluding the run time of the individual test cases. Through experimental results obtained from a test engineer at Axis, the overhead for an individual change was 3 minutes when running the GUI test on a physical device, and 6 minutes when using an emulator. In Table 4.1 we compare the overhead using

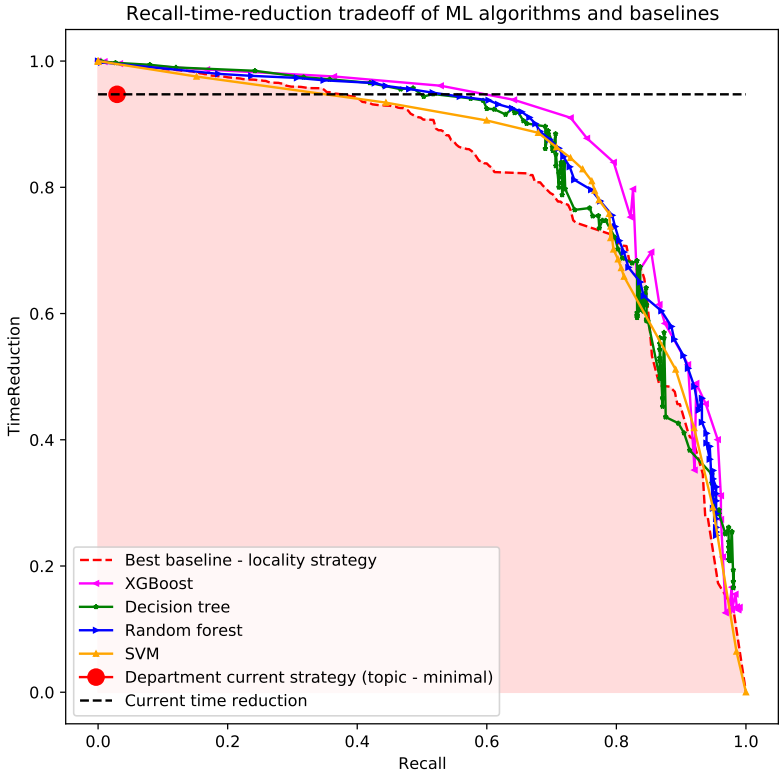


Figure 4.2: Trade off between recall and time reduction for all ML strategies against the best heuristic. The filled area under the locality strategy indicates situations where it outperforms other strategies.

XGBoost, the strategy with the best selection performance, with *department current strategy*. We tested the XGBoost algorithm on 415 changes and computed total overhead on these changes for both strategies. In 71% of changes XGBoost recommends to not select any tests, therefore there is no overhead for these changes. *Department current strategy* will always run a test for each change, which means that there will be an overhead associated with every change.

4.2 Feature Importance

This section presents the results from the different methods of evaluating feature importance, as described in Section 3.5.3. The models used here were trained using the same data and hyperparameters as used in Section 4.1, with balanced class weights.

The embedded feature importance results presented here are based on Decision Tree, Random Forest and XGBoost respectively. Gini importance was used as feature importance criteria for Decision Tree and Random Forest, while gain was used for XGBoost. The 20 highest scoring features using gini importance for Decision Tree and Random Forest are plotted in Figures 4.4 and 4.5 respectively. The gain importance score for XGBoost for the 20 highest scoring features are plotted in Figure 4.6.

Results of the permutation importance method using Decision Tree, Random Forest and XGBoost are presented in Figures 4.7, 4.8 and 4.9 respectively. Every feature in the test set was

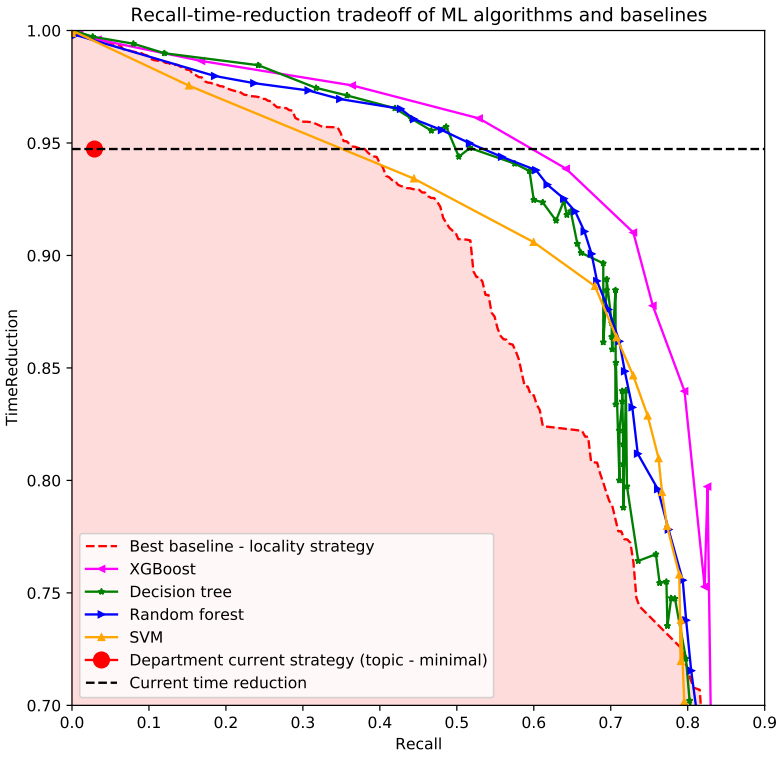


Figure 4.3: Magnified image over trade off between recall and time reduction for all ML strategies against the best heuristic. The filled area under the locality strategy indicates situations where it outperforms other strategies.

permuted 5 times between predictions. The mean importance for each feature was calculated as the drop in balanced accuracy score averaged over the 5 iterations. The mean importance for the 20 highest scoring features are plotted in the figures with the standard error between the 5 iterations.

Using forward selection as wrapper method with the XGBoost model on all features from Table A.1, starting with 1 feature selected and ending at 35 features selected, the highest scoring feature combination was found when 17 features were used. They can be summed up to the following:

- 12 author features (out of 29 total)
- `Ext_png`: Files with PNG file format extension.
- `File_authors_604800_var`: Variance of number of authors per file in the last 7 days.
- `File_authors_1209600_var` : Variance of number of authors per file in the last 14 days.
- `File_authors_7776000_var`: Variance of number of authors per file in the last 90 days.
- `Path_44`: Change ratio of files in one of the directories.

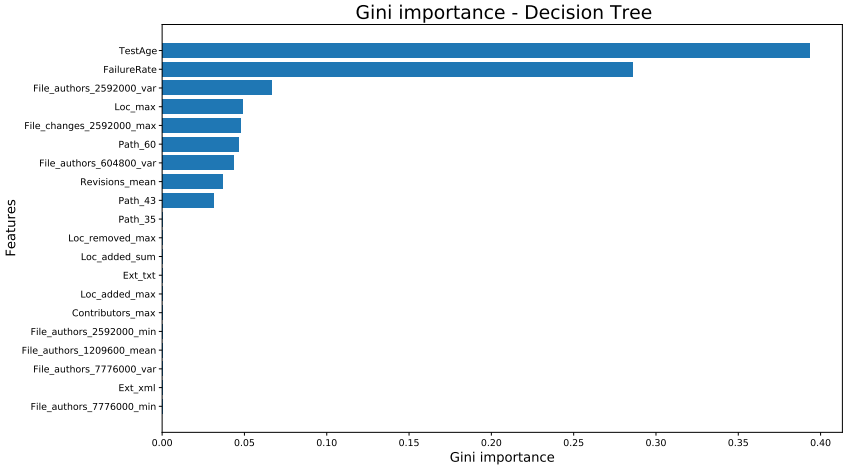


Figure 4.4: Decision tree gini feature importance.

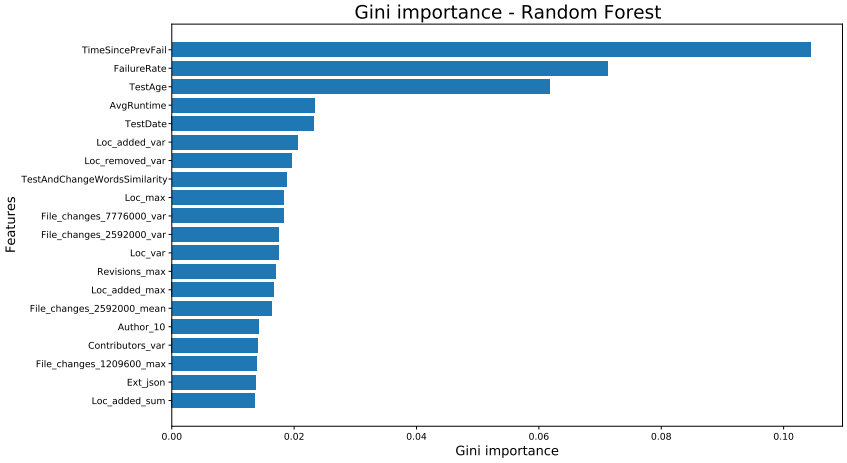


Figure 4.5: Random forest gini feature importance.

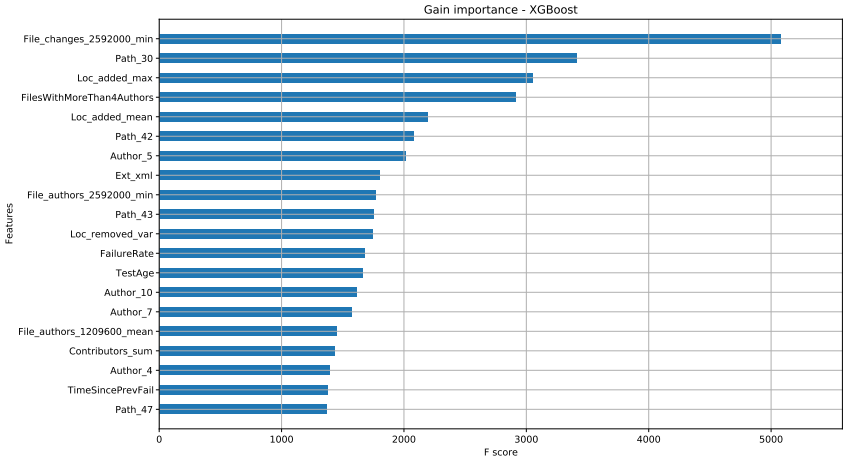


Figure 4.6: XGBoost feature importance based on gain.

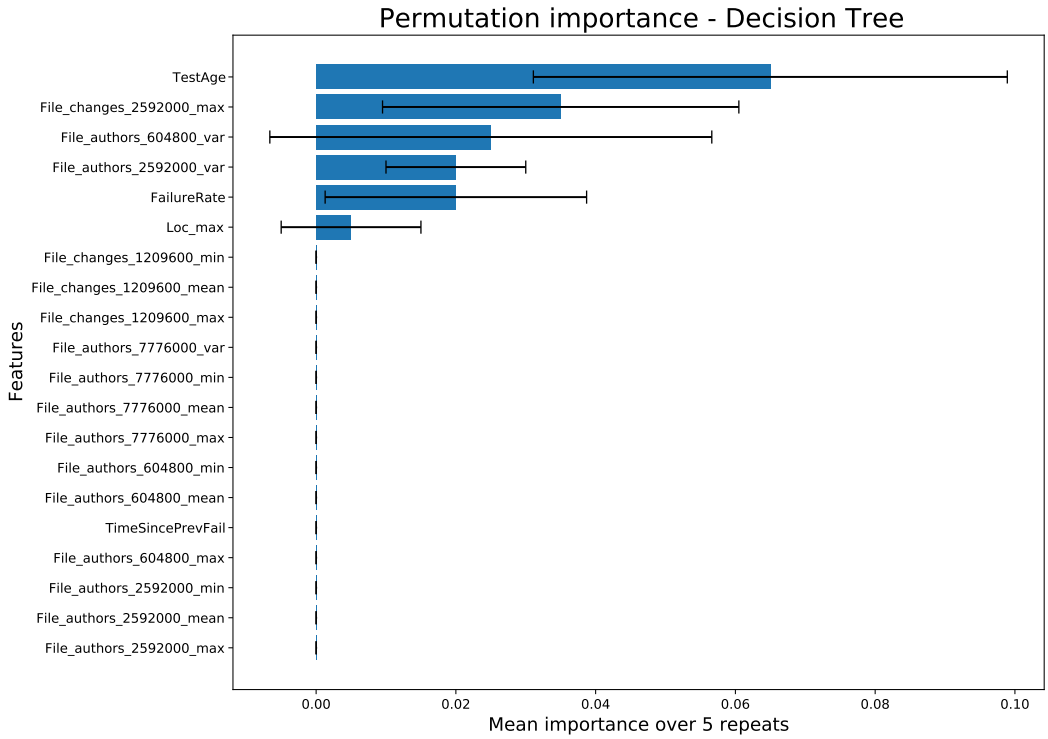


Figure 4.7: Permutation importance for Decision Tree averaged over 5 attempts. The error bar represents standard error.

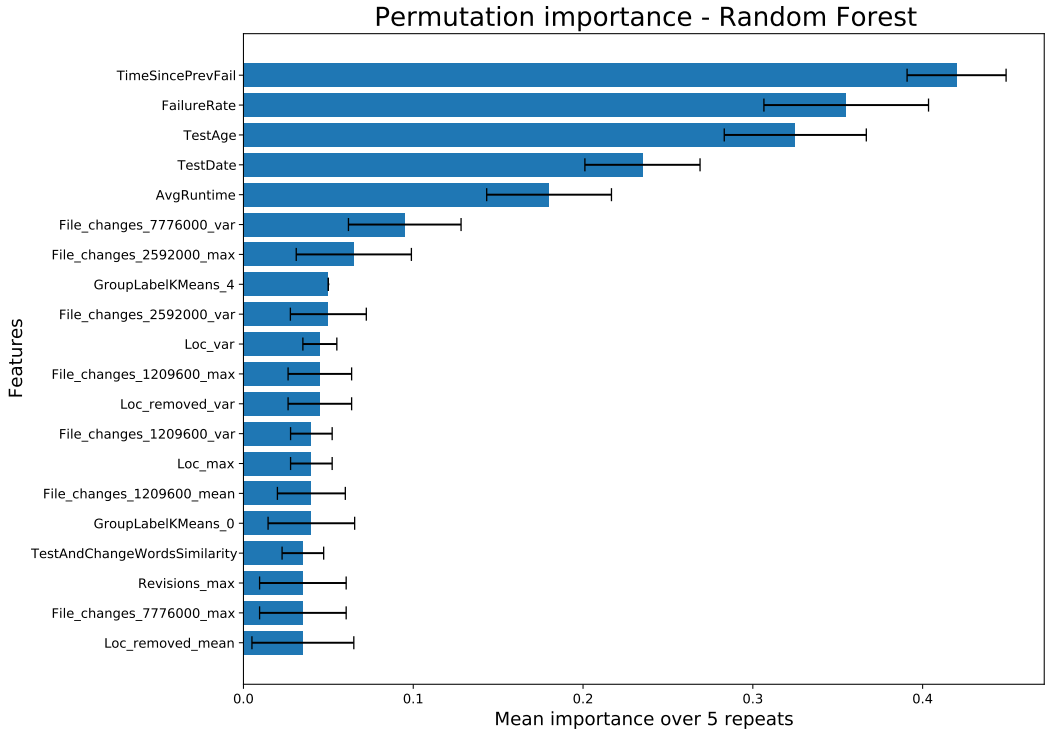


Figure 4.8: Permutation importance for Random Forest averaged over 5 attempts. The error bar represents standard error.

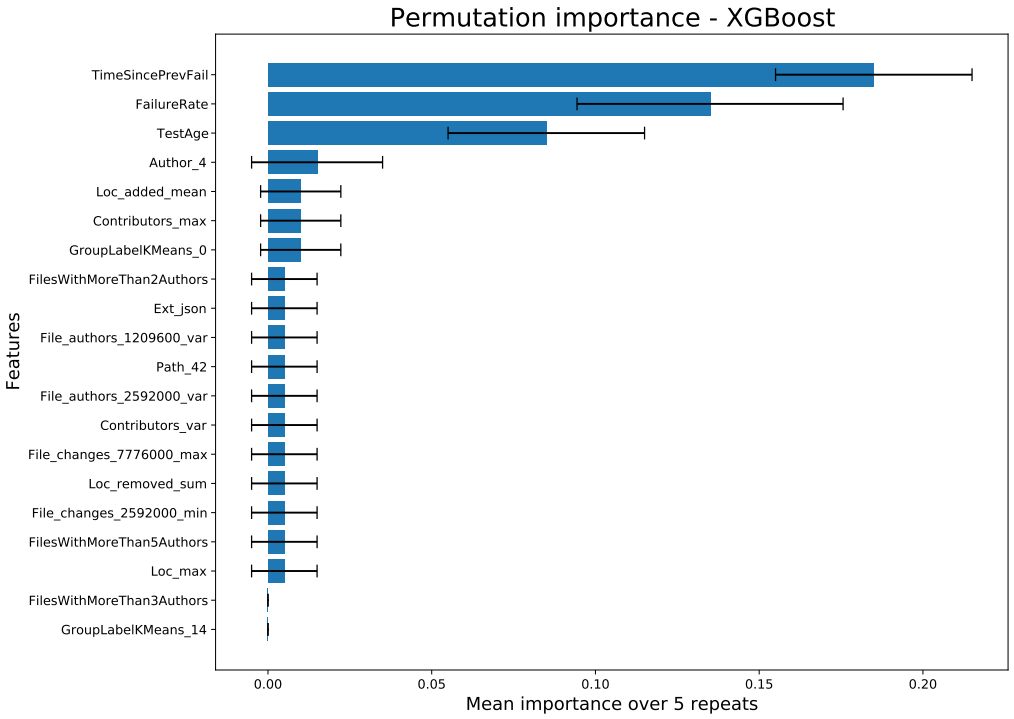


Figure 4.9: Permutation importance for XGBoost averaged over 5 attempts. The error bar represents standard error.

4.3 Scalability

The results from the scalability evaluations are presented here. For each model, described in Section 2.4, it was evaluated how the amount of changes and how the amount of test cases in the training and validation set affect the performance of the model. For each evaluation creating training and validation subsets and retraining a new model. The results from varying the amount of changes can be found in Figures 4.10, 4.11 and 4.12, while the results from varying the amount of test cases can be found in Figures 4.13, 4.14, and 4.15. The figures are all structured in the same way, presenting two graphs, one with recall and one with reduction. The graphs show the median, mean with a standard deviation, and a rolling mean.

Additionally there is a graph that shows how recall, selectivity, and balanced accuracy are affected by the number of changes available in the data set. The evaluations were performed on the XGBoost model and the results are presented in Figure 4.16.

Lastly in Figure 4.17 we see how long it takes in seconds to train each model. In this figure we have included all the models we have evaluated.

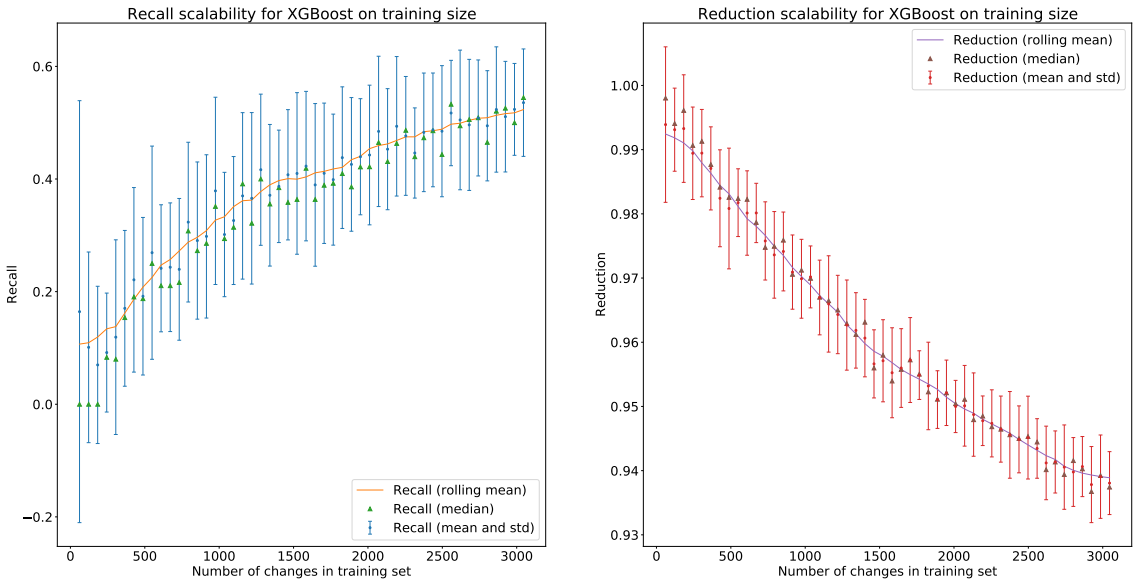


Figure 4.10: The graph on the left shows how the recall of an XGBoost model is affected when increasing the amount of changes available in the data set. On the right we have a similar graph for reduction. The error bar represents standard error.

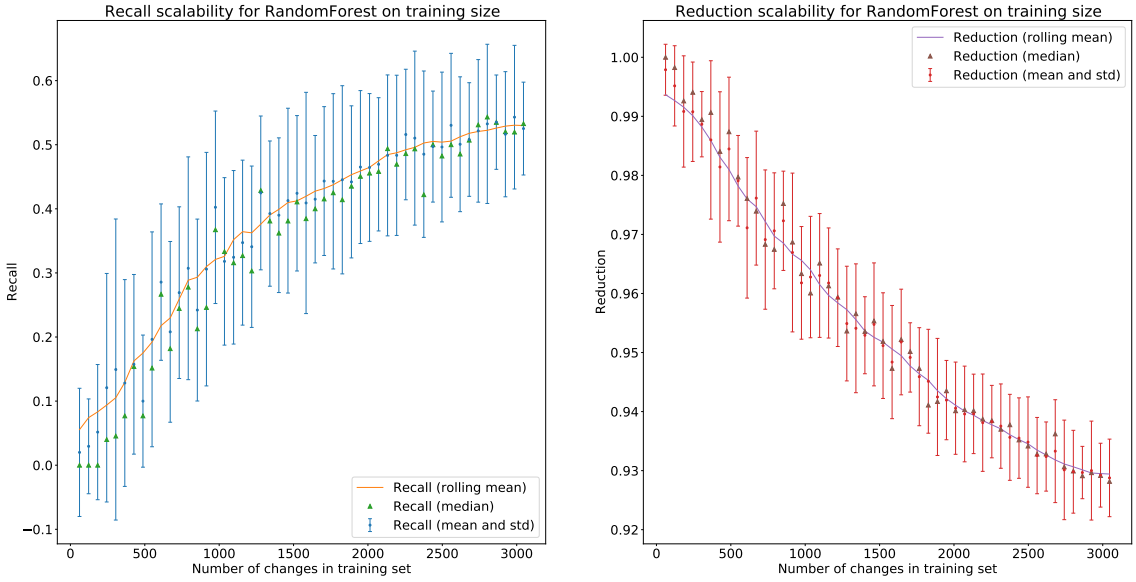


Figure 4.11: The graph on the left shows how the recall of a Random Forest model is affected when increasing the amount of changes available in the data set. On the right we have a similar graph for reduction. The error bar represents standard error.

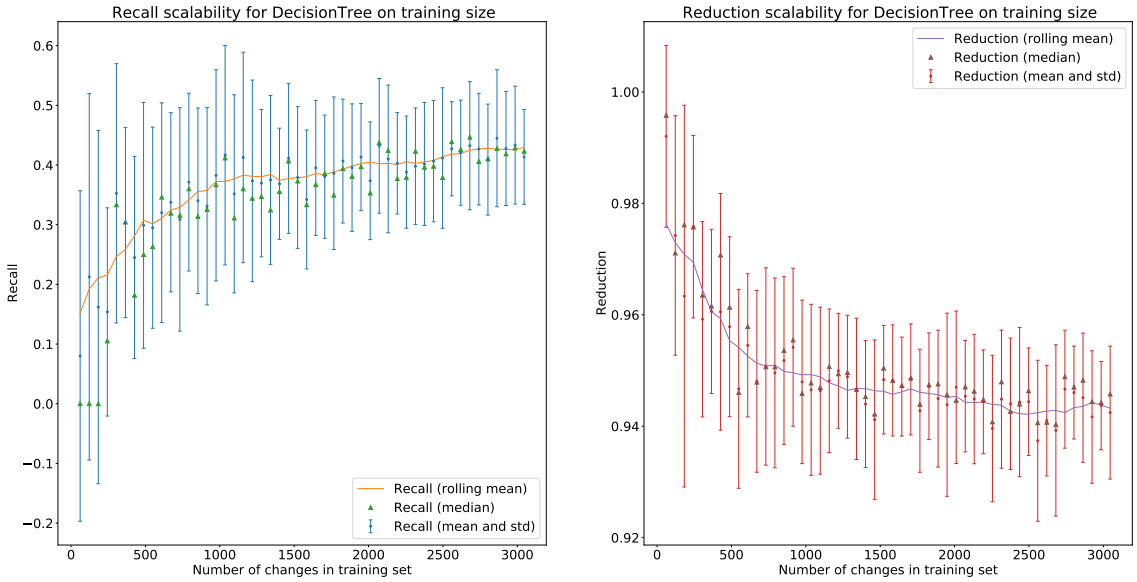


Figure 4.12: The graph on the left shows how the recall of a Decision Tree model is affected when increasing the amount of changes available in the data set. On the right we have a similar graph for reduction. The error bar represents standard error.

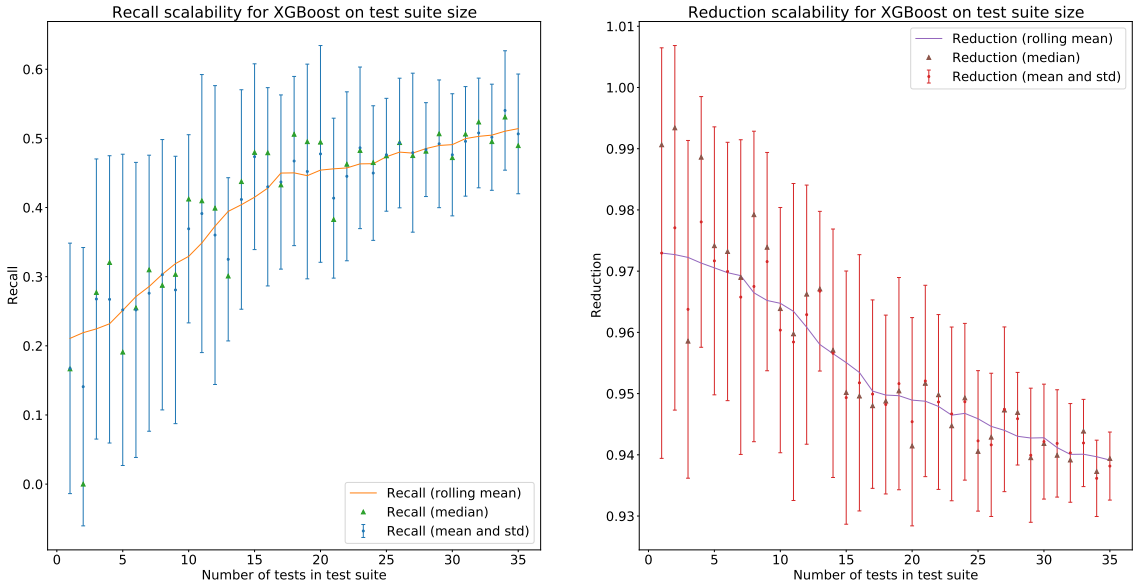


Figure 4.13: The graph on the left shows how the recall of an XGBoost model is affected when increasing the amount of tests available in the test suite. On the right we have a similar graph for reduction. The error bar represents standard error.

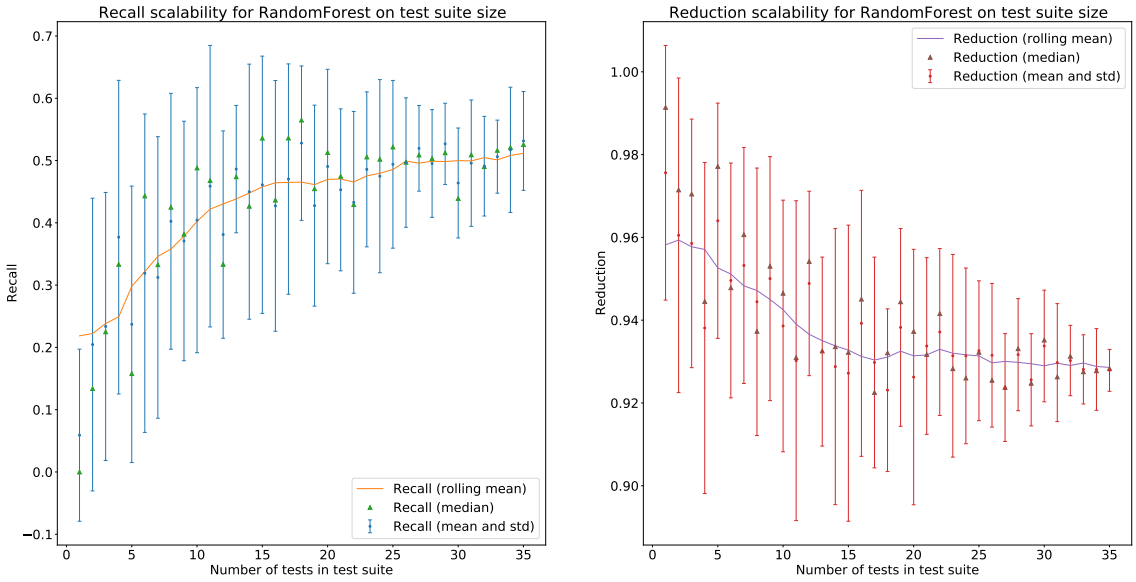


Figure 4.14: The graph on the left shows how the recall of a Random Forest model is affected when increasing the amount of tests available in the test suite. On the right we have a similar graph for reduction. The error bar represents standard error.

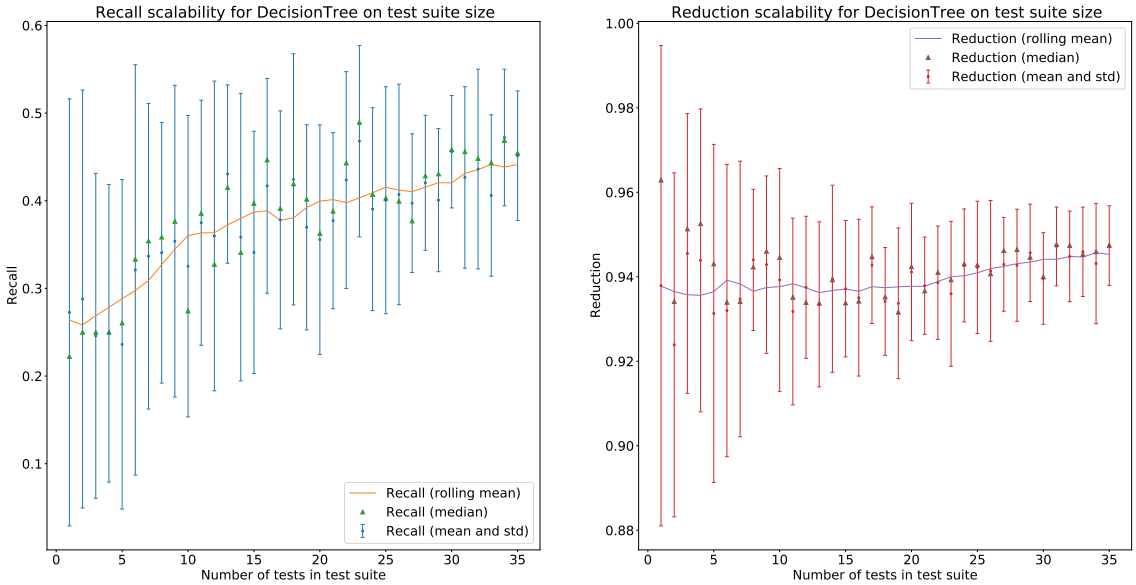


Figure 4.15: The graph on the left shows how the recall of a Decision Tree model is affected when increasing the amount of tests available in the test suite. On the right we have a similar graph for reduction. The error bar represents standard error.

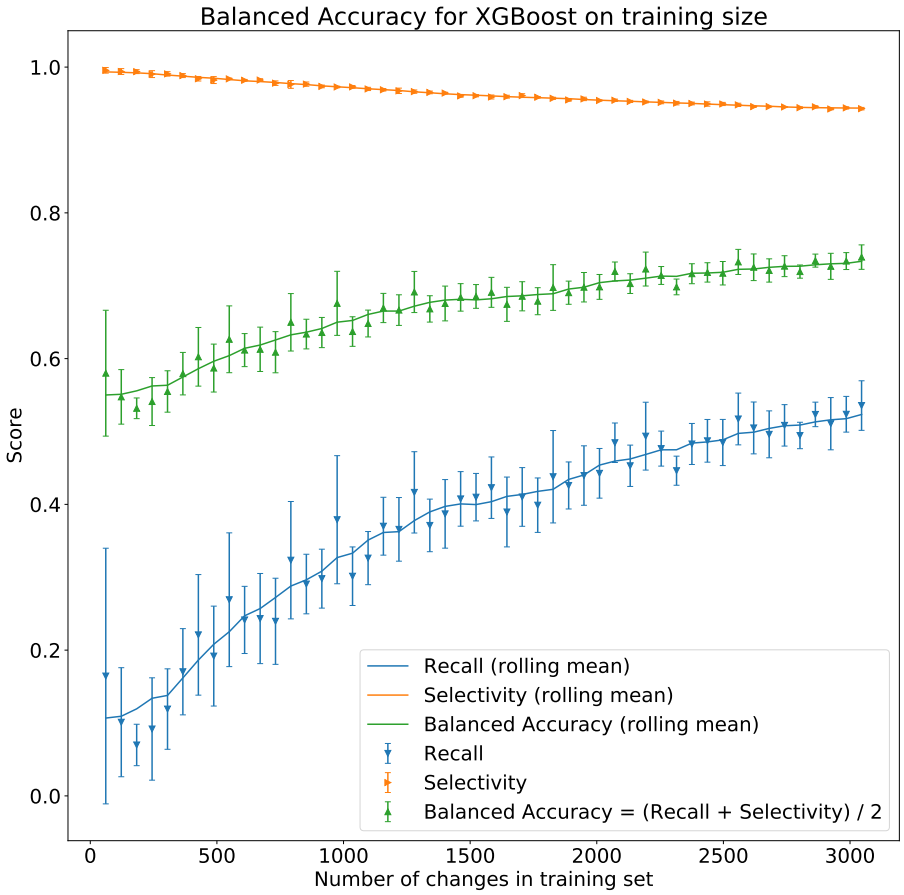


Figure 4.16: The graph shows how the *Balanced Accuracy*, based on *Recall* and *Selectivity*, of an XGBoost model is affected when varying the amount of changes available in the data set. The mean is plotted with an error bar representing a standard error. The equations can be found in Section 3.5.

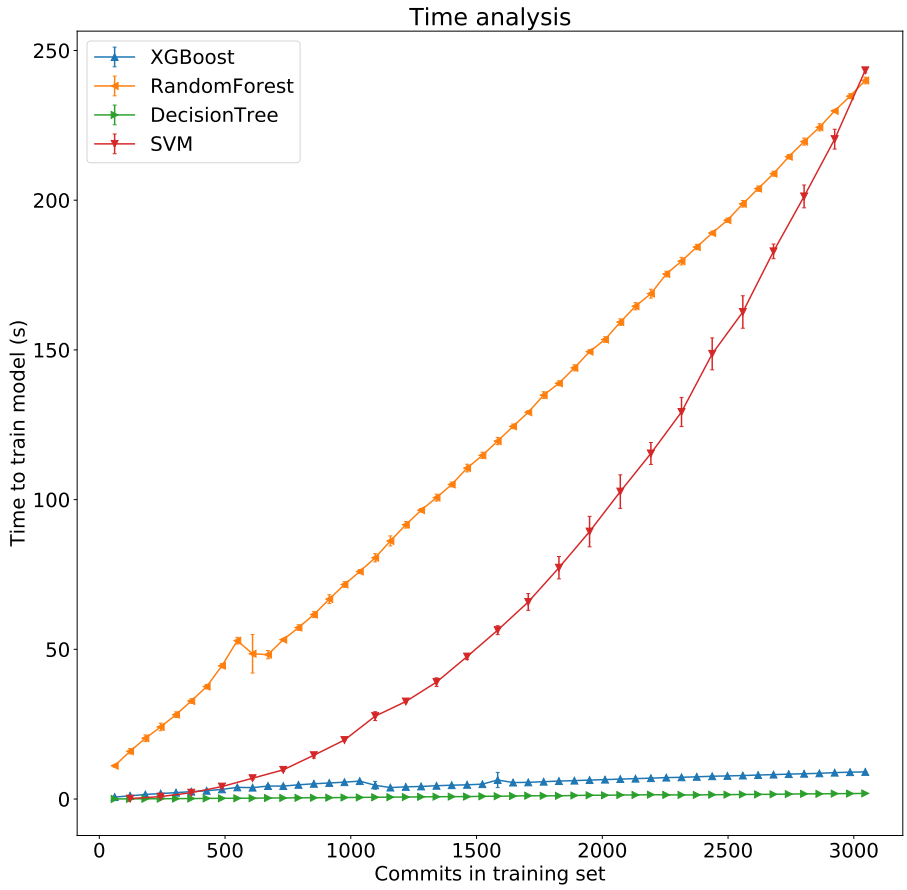


Figure 4.17: Time required to train the models based on size of training data set. It was measured by taking the difference in time from before and after training.

Chapter 5
Discussion

This chapter discusses the result from the previous chapter with the aim of answering the research questions of the thesis.

5.1 Selection Performance

In Figure 4.2 we see that the ML algorithms perform significantly better than the baseline in the more balanced trade off regions. The differences seem to diminish between the strategies the more unbalanced the trade off between recall and time reduction becomes. The best performing algorithm was XGBoost, which had most of its points above the other curves in Figure 4.2, closely followed by Random Forest, Decision Tree, Support Vector Machine and finally, the Locality Strategy. However, other aspects that are worth taking into consideration when choosing one of these algorithms are training time and complexity.

Training time could be affected by the choice of hyperparameters. For example, one hyperparameter was the number of trees to build in the forest, and we performed a randomized grid search over 100 to 5000 trees which is shown in Figure A.1 as `n_estimators`. Over the 100 iterations that the grid search performed, we can see in Figure A.2 that it found the most optimal number of trees to be 4511. A choice of a significantly lower number of trees might have reduced performance slightly but improved the training time by a large amount. However, for our purposes the training time was of less importance since we, as stated in **RQ1**, were interested the selection performance of machine learning algorithms.

An advantage of the tree-based algorithms over SVM, as mentioned in Section 2.4.2, was that they did not require normalization of features. In addition, they do not require one-hot encoding of categorical features. The reduced implementation complexity of tree-based algorithms, in combination with their better performance and linear training times shown in Figure 4.17, indicate that they are better suited for this type of problem than SVM.

Further, we see in Figure 4.3 that while retaining the time reduction of *department current strategy* (94.7%), the recall can be increased from 3% to 40% using *locality strategy* or 60% using

XGBoost. In other words, without increasing the current time of GUI testing, a much higher recall can be achieved using either of the two new strategies.

5.2 Feature Importance

The top 3 features found using permutation importance with XGBoost, which was the best performing algorithm, are features solely based on test history data. This is also the case using the embedded method and permutation importance methods with Random Forest shown in Figure 4.5 and Figure 4.8 respectively. In the permutation importance graphs, most of the remaining permuted features either cause a standard deviation that overpowers the performance score or cause an insignificant performance score to be considered as independently important. Therefore it can be said that test related features are independently good at predicting test failures.

The embedded methods for decision tree, shown in Figure 4.4, and random forest, shown in Figure 4.5 are both based on gini importance. A drawback of gini importance, also known as Mean Decrease in Impurity (MDI), is that it favors variables that have many unique values [7]. We see in both graphs that TimeSincePrevFail, FailureRate and TestAge are the features that stand out the most. FailureRate can at most have 35 unique values, since there are 35 different test cases. Therefore the gini importance for FailureRate is not due to a high number of unique values. As TestAge is the relative time between a change and a modified test, this feature has many unique values due to most changes having unique timestamps. This is also reflected in Table A.1 showing unique values for all features, where TestAge has 40780 unique values. TimeSincePrevFail also has a high number of unique values at 12017. The high number of unique values could be a reason why TestAge and TimeSincePrevFail receive high gini importance scores. However, they are also one of the most important features using the permutation importance methods for tree-based algorithms, which can be verified with Figures 4.8 and 4.9. Therefore we can conclude that TestAge and TimeSincePrevFail are not only favored due to having many unique values, they also affect prediction performance negatively when they are permuted, and thus are important features. Other features with high gini or gain importance were number of authors, number of revisions, lines of code and project structure (path).

We also saw in Section 4.2 that forward selection found individual authors to be important features. The author feature was not very prominent when using the embedded and permutation importance methods. This implies that authors are not important features on their own, but are dependent features that are important together, which is exactly the relationship that wrapper methods are known to capture [14].

Machalica et al. found that their best performing model used file extensions, number of changes made to modified files, historical failure rates for tests, project name and minimal distance as features [9], of which we included the first three. Using permutation importance and embedded feature performance methods of tree-based algorithms we have also found that these features are important for predicting test failures. In addition, we also found that authors, test age, lines of code and project structure showed high potential as being important features for test selection.

5.3 Scalability

From Figures 4.10, 4.11, and 4.12, we can clearly see an improvement in recall as the amount of training data is increased. However, reduction is simultaneously decreasing. Notably in Figure 4.10, the reduction is reduced from about 99% to 94%. From Section 3.3.1 we know that the amount of failures in the complete data set are approximately 1%, hence a change in recall can at most affect the reduction as much. This means that the 5% decrease in reduction is not only caused by the increase in recall, and the model is not improving in both aspects. To see if the model is improving overall we must look at another measure. In Figure 4.16 we see how the balanced accuracy varies with the amount of training data. Balanced Accuracy is a good measure of overall performance since it depends on both Recall and Selectivity, which refer to how well the model is correctly labeling data for the classes select and ignore respectively. Importantly, recall and selectivity are relative to the class sizes rather than absolute, which makes balanced accuracy unbiased towards either. In the figure we can see that the balanced accuracy is indeed improving, because recall is increasing much faster than the selectivity is decreasing.

In Figures 4.13, 4.14, and 4.15 we evaluated how performance varies with the amount of test cases available in the data set. Here all figures show similar increase in recall. Interestingly, for the Decision Tree model, there seems to be an increase in reduction as well. We are uncertain as to why this is the case, however given this result, the Decision Tree model scales best with additional test cases.

One interesting scenario is when a new test case is added. Since all the ML-strategies and most heuristics rely on historical data of the individual test cases, their selection performance will most likely vary when a new test is introduced. This could be evaluated by introducing a test case to the test set but not the training set, and then evaluate the model's performance. Additionally, the test case could be introduced gradually to the training set to see how the performance evolves as the data for the test case increases. Unfortunately there was no time to perform such an evaluation. One advantage the machine learning models have over the heuristics are the software change features, which could be used to predict failures for new test cases.

In Figure 4.17 we find an estimate for how long it takes to build a model given the amount of changes in the training set. Here we can see the advantage XGBoost and Decision Tree have over Random Forest and SVM. However it is important to note that the time complexity of Random Forest could be due to the number of estimators it depends on, as was explained in Section 5.1.

We found that there were diminishing returns when introducing additional data, for both new changes and new test cases, shown in Figures 4.10 and 4.13. One could possibly achieve further improvements to the rate of which the model learns by instead introducing data from new environments. In our context, a new environment could be using another programming language or another project. A reason as to why this could help is because it would expand the domain of data points. However, it could also be the case that there is no transfer learning between projects, meaning that the learning from one project does not apply to another. Therefore we think it could be interesting to further research the affect of using several projects, and the transfer-learning potential.

5.4 Alternative Solutions

As mentioned in Section 3.1.2 we discussed other solutions with an expert engineer at Axis who had been involved in another internal project where machine learning was used for RTS. However, the machine learning solution they designed was outperformed by another strategy that simply involved running tests that had failed in the past couple of months. We thought it would be interesting to see how the latter strategy of the two would perform against our solution, hence we designed the *Locality strategy*. However, instead of only selecting tests that had failed in the past couple of months, we experimented with different time intervals. As can be seen in Figure 4.1, the *Locality Strategy* has been the best performing heuristic in our evaluation. Using only test history data it is almost able to keep up with the ML algorithms in terms of selection performance.

The second best performing heuristic was the high risk test strategy. From Figure 4.1 we can see that its performance is similar to *Locality strategy* when recall is favored, however it performs worse when reduction is favored. Complexity wise, the *High risk test strategy* is easier to implement since it only requires the failure rate of all test cases to make its selection.

The benefits of the *Random selection strategy* are that it is very easy to implement and requires no historical data. The drawback is however that its selection performance is very poor. Even so, it manages to outperform *department current strategy* based on time reduction and recall.

An alternative method that we did not evaluate is static analysis as a method for RTS. Static analysis of the program code can be performed to find and select test cases that are relevant to the changed source code. Due to how GUIs are programmed in visual environments, they are not well-typed and are subject to frequent changes. This makes it difficult to use them when performing static analysis [22].

As mentioned in Section 3.5.2, a requirement from Axis was that the trade off between coverage and testing time should be easily adjustable depending on the use case. If testing times are hindering the development, they want to be able to sacrifice some amount of test coverage for time, and vice versa. This is another reason why traditional static test selection strategies are not as suitable. Strategies such as these can only be performed at different granularity levels broadly divided into file or function level analysis [6]. In contrast, we have seen that the binary machine learning classifiers that we have used are able to adjust continuous weights to favor a certain class over another, ultimately allowing for finer adjustment of the trade off between coverage and time. In the case of the classes being defined as *select test* and *ignore test*, higher recall or higher reduction can be achieved by weighing the algorithm towards one of these classes. A trade off between coverage and time is therefore adjustable to a higher degree with a binary classifier than a static code analysis.

Despite the difficulty of performing static analysis of GUIs, it has been used to create high level graphical representations of code through reverse engineering. This enables reasoning between user interaction and the system at a higher level of abstraction, and makes it possible to define new GUI test cases from the graphical representations. [18] However, in the case of Axis where several GUI test cases have already been defined, they would have to be manually mapped to such graphical representations in order to bridge the gap between source code and test cases, which would require manual intervention every time a new test case is introduced.

5.5 Limitations

The machine learning implementations in this thesis are limited to the ones provided by the scikit-learn and XGBoost frameworks for Python, due to their wide support and ease of use. All of the code except for the data collection, written in the Bash command language, was written using Python version 3.7. Python was chosen since both of us had experience using it and since it has wide support within the machine learning and data science community.

One of the early problems we faced was to collect and process the historical data. Since there was no available data prior to the thesis work, we had to start from the beginning. The process of collecting data was time consuming and limited us in exploring other aspects that could improve the system.

5.6 Validity Threats

As explained in Section 3.4.2 we imposed several conditions on the training and tests sets to simulate a scenario where the models would be used. This resulted in varying ratios between their sizes. If the test set size is too small, the variance in the result will be high. If the training set size is too small, the model may not learn enough.

Since a requirement was that a change could only be present in either the training or test set, the variance in failures between changes could affect the distribution of the classes in the two sets. The variation in amount of failures per change was quite large, this was due to almost all changes having no failures and a small amount of changes containing almost all failures. Depending on how the split is made on changes, the result will be affected. This is reflected by the variance in the Figures 4.10, 4.11, and 4.12. However, the same split on changes is used for all strategies so that they can be compared. This means that the lines in Figure 4.2 may be shifted either in x-direction or y-direction if a new split was decided, but they will remain in proportion to one another. In other words, the variance in the result from splitting the changes differently should affect all methods equally.

5.7 Ethical Considerations

While the data that we collected comes from sources that are available to access for all Axis employees, there are cases where it can be used for unethical purposes. For instance, we gather data on which author who has made a certain change. In conjunction with other metrics that we collected, such as lines of code and test failure data, the data can theoretically be used to analyze authors' working patterns. Therefore we made the authors in this report anonymous by changing their names to numbers in the data. In future works we also recommend that if authors are used as features, names should be made anonymous, so that the data cannot be used to make claims about specific authors.

5.8 Future Work

We originally collected data for differences between changes down to source code level. Our intention was to use this data for string comparison with the source code for individual test cases. Similarly to how we did with our cross feature described in Section 3.2.2, one could compare similar words in the source code with the test case code to create additional cross features that could potentially further improve the learning of the machine learning algorithms.

As mentioned in the end of Section 5.3, another interesting prospect is to include the historical data of multiple software projects to train the same algorithm, which is called transfer learning. With general features like the ones we used, it should be possible to have a model that can learn from many different projects. We have seen how machine learning prediction performance scales with more data, and if the data is more varied it can also help to reduce the chance of overfitting.

From the results in Section 4.2 we saw that test related features were highly relevant for the prediction performance of the ML-algorithms. Therefore we are interested in investigating other test related features that could improve the prediction performance. For GUI tests one could investigate features explaining what type of GUI elements and navigation types that are used. For instance lists, drawers or buttons, and clicks, scrolls or swipes.

Chapter 6
Conclusion

Throughout this report we have studied how machine learning can be used as an RTS strategy. We have shown how to design a system that uses historical data on code changes and test outcomes to predict if a test should be run or not. We have shown how to prepare the data so that it can be processed by different algorithms, and how the algorithms can be trained and evaluated. We have concluded that although more primitive solutions can achieve promising results, using machine learning yields even better performance. We found that it could significantly reduce waiting time for developers when using regression testing during development, while at the same time ensuring that as many regressions as possible are caught. We found that when favoring test coverage we were able to achieve 91% recall and 52% time reduction. Conversely, when maintaining the same time reduction as the current strategy of the department, the recall was increased from 2% to 53%, with no tests being selected in 71.2% of changes.

We have shown how machine learning can be used to find information that is important for determining whether a test will fail. Using different feature importance methods we showed that test age, authors, revisions, failure rates, time since previous failure, file extensions, lines of code and changed directories are all features that have an effect on test outcomes.

Finally, we conclude that tree-based machine learning models scale well with the type of data used. We found that the initial growth in performance has been passed and saw diminishing returns when introducing additional data. This indicates that the amount of data is sufficient, and for further improvements one should explore additional feature engineering or introducing data from new environments. Furthermore we found that the most efficient algorithm, in terms of performance versus training time was XGBoost.

References

[1] D. Berrar. Cross-validation. In *Encyclopedia of Bioinformatics and Computational Biology*, pages 542–545. Academic Press, 2019.

[2] S. Chacon and B. Straub. *Pro Git*. Apress, USA, 2nd edition, 2014.

[3] P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer, and R. Wirth. Crisp-dm 1.0 step-by-step data mining guide. Technical report, The CRISP-DM consortium, 2000.

[4] E.D. Ekelund. Test selection based on historical test data, 2015. MSc report.

[5] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pages 126–135, 2000.

[6] 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 *FSE 2016 - Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 583–594. Association for Computing Machinery, November 2016.

[7] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts. Understanding variable importances in forests of randomized trees. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[8] E. Lundsten. EALRTS: A predictive regression test selection tool, 2019. MSc report.

[9] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. *IEEE Transactions on Software Engineering*, 2019.

[10] A. Mahmoud Muthanna and L. Hellgren Winblad. Regression test selection för snabb återkoppling när utveckling görs för androidplatformen, 2018. BSc report.

[11] A. Mamat, F. Mohamad, M.A. Mohamed, N. Rawi, and M.I. Awang. Silhouette index for determining optimal k-means clustering on images in different color models. *International Journal of Engineering & Technology*, 7:105, 04 2018.

[12] G. Mastorakis. Human-like machine learning: limitations and suggestions. *CoRR*, abs/1811.06052, 2018.

[13] A. Memon, B. Nguyen, E. Nickell, J. Micco, S. Dhanda, R. Siemborski, and Z. Gao. Taming google-scale continuous testing. In *ICSE '17:Proceedings of the 39th International Conference on Software Engineering*, 2017.

[14] R. Musheer, C.K. Verma, and N. Srivastava. Dimension reduction methods for microarray data: a review. *AIMS Bioengineering*, 4:179–197, 03 2017.

[15] V. Pappu and P. Pardalos. *High Dimensional Data Classification*, page 34. Springer, New York, NY, 12 2013.

[16] L. Phan, H. Liu, and C. Tortora. K-means clustering on multiple correspondence analysis coordinates. *Archives of Data Science, Series B (Online First)*, 1(1):B05, 17 S. online, 2019.

[17] C. Seger. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing, 2018. BSc report.

[18] J. Silva, J. Campos, and J. Saraiva. Gui inspection from source code analysis. In *Electronic Communications of the EASST*, 01 2010.

[19] P. Simon. *Too Big to Ignore: The Business Case for Big Data*. Wiley Publishing, 1st edition, 2013.

[20] M. Steinbach, L. Ertöz, and V. Kumar. The challenges of clustering high-dimensional data. In *In New Vistas in Statistical Physics: Applications in Econophysics, Bioinformatics, and Pattern Recognition*. Springer-Verlag, 2003.

[21] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar. *Introduction to Data Mining (2nd Edition)*. Pearson, 2nd edition, 2018.

[22] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67 – 120, March 2012.

[23] C. Yuan and H. Yang. Research on k-value selection method of k-means clustering algorithm. *J*, 2(2):226–235, 2019.

Appendices

Appendix A
Design

A.1 Features

Table A.1: All features, as they are fed as inputs to the ML-algorithms, including number of distinct values.

#	Feature	Unique values	Description
1	Author_1	2	Author of the change (true/false)
2	Author_10	2	
3	Author_11	2	
4	Author_12	1	
5	Author_13	2	
6	Author_14	2	
7	Author_15	2	
8	Author_16	2	
9	Author_17	2	
10	Author_18	2	
11	Author_19	2	
12	Author_2	2	
13	Author_20	2	
14	Author_21	2	
15	Author_22	2	
16	Author_23	2	
17	Author_24	2	
18	Author_25	2	
19	Author_26	2	
20	Author_27	2	
21	Author_28	2	
22	Author_29	2	
23	Author_3	2	
24	Author_4	2	
25	Author_5	2	

Continued on next page

Table A.1 – continued from previous page

#	Feature	Unique values	Description
26	Author_6	2	
27	Author_7	2	
28	Author_8	2	
29	Author_9	2	
30	AvgRuntime	35	Average run time of the test
31	Contributors_max	21	Max # contributors for a changed file
32	Contributors_mean	422	
33	Contributors_min	17	
34	Contributors_sum	201	
35	Contributors_var	803	
36	Ext_2	1	# changed files with file extension
37	Ext_Gemfile	2	
38	Ext_aar	2	
39	Ext_apk	2	
40	Ext_eap	6	
41	Ext_env	2	
42	Ext_gitignore	2	
43	Ext_gradle	3	
44	Ext_java	37	
45	Ext_json	6	
46	Ext_kt	95	
47	Ext_lock	2	
48	Ext_md	2	
49	Ext_orig	2	
50	Ext_out	1	
51	Ext_png	13	
52	Ext_properties	3	
53	Ext_py	2	
54	Ext_txt	5	
55	Ext_xml	43	
56	FailureRate	24	The test's historical failure rate
57	File_authors_1209600_max	6	Max # authors per file in the last 1209600 sec
58	File_authors_1209600_mean	165	
59	File_authors_1209600_min	5	
60	File_authors_1209600_var	227	
61	File_authors_2592000_max	8	
62	File_authors_2592000_mean	244	
63	File_authors_2592000_min	7	
64	File_authors_2592000_var	410	
65	File_authors_604800_max	6	
66	File_authors_604800_mean	110	
67	File_authors_604800_min	5	
68	File_authors_604800_var	134	
69	File_authors_7776000_max	10	
70	File_authors_7776000_mean	306	
71	File_authors_7776000_min	10	
72	File_authors_7776000_var	580	
73	File_changes_1209600_max	20	Max # changes per file in the last 1209600 sec
74	File_changes_1209600_mean	358	
75	File_changes_1209600_min	12	
76	File_changes_1209600_var	693	
77	File_changes_2592000_max	29	
78	File_changes_2592000_mean	487	

Continued on next page

Table A.1 – continued from previous page

#	Feature	Unique values	Description
79	File_changes_2592000_min	20	
80	File_changes_2592000_var	912	
81	File_changes_604800_max	15	
82	File_changes_604800_mean	258	
83	File_changes_604800_min	12	
84	File_changes_604800_var	462	
85	File_changes_7776000_max	62	
86	File_changes_7776000_mean	647	
87	File_changes_7776000_min	38	
88	File_changes_7776000_var	1210	
89	Files	117	# files in the change
90	FilesWithMoreThan1Authors	66	# files with more than 1 author
91	FilesWithMoreThan2Authors	31	
92	FilesWithMoreThan3Authors	12	
93	FilesWithMoreThan4Authors	5	
94	FilesWithMoreThan5Authors	4	
95	GroupLabelKMeans_0	2	If the test belongs to test group 0
96	GroupLabelKMeans_1	2	
97	GroupLabelKMeans_10	2	
98	GroupLabelKMeans_11	2	
99	GroupLabelKMeans_12	2	
100	GroupLabelKMeans_13	2	
101	GroupLabelKMeans_14	2	
102	GroupLabelKMeans_15	2	
103	GroupLabelKMeans_16	2	
104	GroupLabelKMeans_2	2	
105	GroupLabelKMeans_3	2	
106	GroupLabelKMeans_4	2	
107	GroupLabelKMeans_5	2	
108	GroupLabelKMeans_6	2	
109	GroupLabelKMeans_7	2	
110	GroupLabelKMeans_8	2	
111	GroupLabelKMeans_9	2	
112	Loc_added_max	208	LOC added for file with max LOC added
113	Loc_added_mean	972	
114	Loc_added_min	64	
115	Loc_added_sum	516	
116	Loc_added_var	1555	
117	Loc_max	516	# LOC in changed file with max LOC
118	Loc_mean	1642	
119	Loc_min	271	
120	Loc_removed_max	144	
121	Loc_removed_mean	690	
122	Loc_removed_min	37	
123	Loc_removed_sum	353	
124	Loc_removed_var	1095	
125	Loc_sum	1432	
126	Loc_var	1897	
127	Path_0	2	# changed files in directory
128	Path_1	8	
129	Path_10	2	
130	Path_11	2	
131	Path_12	2	

Continued on next page

Table A.1 – continued from previous page

#	Feature	Unique values	Description
132	Path_13	2	
133	Path_14	2	
134	Path_15	1	
135	Path_16	1	
136	Path_17	1	
137	Path_18	2	
138	Path_19	1	
139	Path_2	2	
140	Path_20	2	
141	Path_21	2	
142	Path_22	2	
143	Path_23	2	
144	Path_24	2	
145	Path_25	2	
146	Path_26	2	
147	Path_27	1	
148	Path_28	2	
149	Path_29	2	
150	Path_3	2	
151	Path_30	2	
152	Path_31	2	
153	Path_32	2	
154	Path_33	2	
155	Path_34	2	
156	Path_35	2	
157	Path_36	2	
158	Path_37	2	
159	Path_38	3	
160	Path_39	2	
161	Path_4	2	
162	Path_40	5	
163	Path_41	2	
164	Path_42	40	
165	Path_43	6	
166	Path_44	13	
167	Path_45	8	
168	Path_46	7	
169	Path_47	9	
170	Path_48	9	
171	Path_49	9	
172	Path_5	2	
173	Path_50	2	
174	Path_51	29	
175	Path_52	11	
176	Path_53	1	
177	Path_54	35	
178	Path_55	19	
179	Path_56	8	
180	Path_57	4	
181	Path_58	2	
182	Path_59	2	
183	Path_6	2	
184	Path_60	14	

Continued on next page

Table A.1 – continued from previous page

#	Feature	Unique values	Description
185	Path_61	5	
186	Path_62	4	
187	Path_63	5	
188	Path_64	5	
189	Path_65	7	
190	Path_66	2	
191	Path_67	21	
192	Path_68	9	
193	Path_69	6	
194	Path_7	1	
195	Path_70	49	
196	Path_71	59	
197	Path_8	2	
198	Path_9	2	
199	Revisions_max	290	# revisions in file with max revisions
200	Revisions_mean	1024	
201	Revisions_min	92	
202	Revisions_sum	626	
203	Revisions_var	1350	
204	TestAge	40780	Test age relative to change
205	TestAndChangeWordsSimilarity	4181	Cosine similarity score
206	TestDate	22	Time of test modification
207	TimeSincePrevFail	12017	Time since test previously failed

A.2 Hyperparameters

```
    decision_tree_parameters = {
        'criterion':['gini','entropy'],
        'max_depth'
    :[2,3,4,5,6,7,8,9,10,11,12,15,20,30,40,50,70,90],
        'min_samples_leaf':[1,2,3,4,5]
    }

    random_forest_parameters = {
        'criterion':['gini','entropy'],
        'n_estimators': list(range(100, 5000)),
        'min_samples_split': list(range(1, 11)),
        'min_samples_leaf': list(range(1, 11)),
        'bootstrap': [True, False],
        'max_depth': list(range(2,15))
    }

    xgboost_parameters = {
        "learning_rate" : stats.uniform(0.05, 0.30),
        'min_child_weight': stats.uniform(1,10),
        'gamma': stats.uniform(0.5, 5),
        'subsample': stats.uniform(0.6, 1.0),
        'colsample_bytree': stats.uniform(0.3, 0.7),
        'max_depth': list(range(3, 20))
    }

    svm_parameters = {
        'C': stats.loguniform(1, 1000),
        'gamma': stats.loguniform(0.0001, 1),
        'kernel': ['rbf']
    }


```

Figure A.1: Specified hyperparameters for the grid search.

```
decision_tree_parameters = {
    'criterion': 'gini',
    'max_depth': 9,
    'min_samples_leaf': 5
}

random_forest_parameters = {
    'n_estimators': 4511,
    'min_samples_split': 4,
    'min_samples_leaf': 5,
    'max_depth': 7,
    'criterion': 'entropy',
    'bootstrap': False
}

xgboost_parameters = {
    'colsample_bytree': 0.35388593688015507,
    'gamma': 1.9487572645688402,
    'learning_rate': 0.09836638617620133,
    'max_depth': 4,
    'min_child_weight': 7.2435404813379325,
    'subsample': 0.895633685837714
}

svm_parameters = {
    'C': 3.9721107273819114,
    'gamma': 0.011400863701127315,
    'kernel': 'rbf'
}
```

Figure A.2: Selected hyperparameters by the grid search.

Appendix B
Test Case History

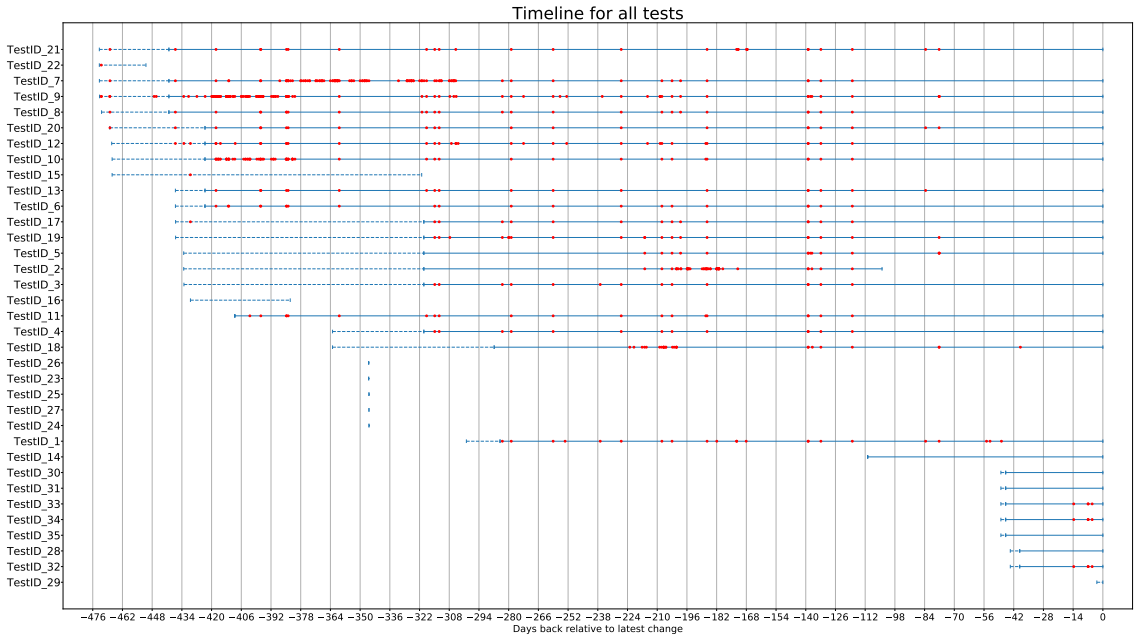


Figure B.1: Timeline graph showing when tests have failed. On the y-axis we have different test IDs and on the x-axis we have how many days ago the change was published. The dots represent when a break happened. The lines show when that particular test have been active. Where the dashed line is before the test was in production, and the solid line afterwards.

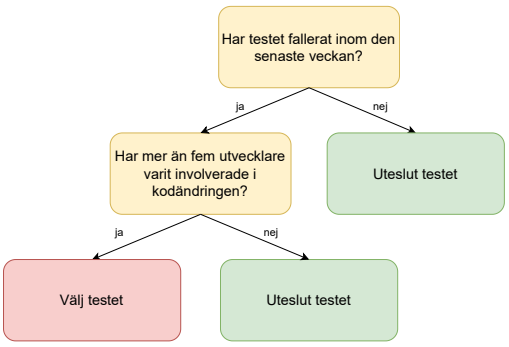
EXAMENSARBETE Selective Testing Using Machine Learning
STUDENTER Saam Mirghorbani, Viktor Claesson
HANDLEDARE Emelie Engström (LTH)
EXAMINATOR Ulf Asklund (LTH)

Maskininlärning för testurval

POPULÄRVETENSKAPLIG SAMMANFATTNING **Saam Mirghorbani, Viktor Claesson**

Intelligenta urval av tester kan göras med hjälp av maskininlärning. För projekt där historisk data för kodändringar och testresultat finns att tillgå, finns det potential för att använda precisa metoder för testurval. Dessa kan justeras till att föredra mindre eller större urval beroende på hur snabb återkoppling som önskas av utvecklaren.

I en värld där allt fler komplexa system utvecklas sätts stora krav på kvalitetssäkring. När allt fler ändringar sker i stora system växer behovet av snabbare och mer träffsäker testning. För att möta behovet krävs automatiska och allt mer intelligenta metoder för testurval som kan anpassas efter både tids- och säkerhetskrav.



Figur 1: Exempel på ett beslutsträd för testurval.

Trädbaserade algoritmer visar sig vara effektiva på att göra testurval. I sin enklaste form består dessa av olika påståenden som tillsammans kokar ner till ett beslut om att antingen välja eller utesluta ett test från körningen.

Mer sofistikerade algoritmer består av flera enkla träd, likt exemplet i figuren, som tillsammans stärker varandra och ger upphov till mer korrekta beslut. Genom att applicera dessa algoritmer med noga utvald information från datan kan man göra en avvägning mellan att fånga fler fel och att korta testtiden utefter behov. I ett fall där träffsäkerhet prioriterades valdes 91% fallerande tester, samtidigt som den totala testtiden kunde halveras.

Genom att applicera vårt system kan man introducera regressionstestning för varje ändring även i projekt där testningsprocessen tar lång tid. En fördel med detta är att man då kan hitta fler fel tidigare under utvecklingsprocessen. I slutändan leder detta till att längre felsökningar blir allt färre, då problem kan handskas med så fort de uppstår.