DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

EALRTS: A predictive regression test selection tool

ERIK LUNDSTEN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

EALRTS: A predictive regression test selection tool

ERIK LUNDSTEN

Master in Computer Science
Date: August 25, 2019
Supervisor: Martin Monperrus
Examiner: Benoit Baudry
School of Electrical Engineering and Computer Science
Host company: Accedo

Abstract

Regression testing is the process of confirming that a code change did not introduce any test failure into the current build. One Regression testing technique commonly used is Regression test selection or RTS. It is the process of identifying all tests affected by a code change, which are identified by creating a dependency graph of the project. The selected tests are then executed. The purpose of RTS is to reduce the development time by lowering the time for testing. Machine learning has been used as a test selection tool in recent studies and have shown promising results. Machine learning were used with a RTS tool to further reduce the number of tests selected. The features are primarily extracted from the dependency graph from the RTS tool. The machine learning is then used to estimate the probability of a test failure, and the tests are selected based on the probability of test failure. However, in order to train a machine learning model, it is essential to have a lot of data, and faulty code changes are required. Code defects need to be tested with the RTS tool while extracting data from running the tests. However, for open source projects, obtaining a large number of historical code defects is challenging. This paper presents EALRTS, a predictive regression test selection tool. EALRTS uses mutation generation instead of historical code defects. The data for the machine learning model is obtained with the help of STARTS, which is a static RTS tool. The data extracted comes mainly from two sources: **(1)** from the dependency graph that STARTS creates. **(2)** And from the test result reports. The data extracted is then used to train a Random Forest algorithm, whose goal is to predict what test to select. EALRTS managed to reduce the number of tests selected by 60.3% while finding 95% of all failed tests. The recall rate is interpreted as the amount of individual test failure found in a test class. The results show a trade-off between the number of individual test failures found and the number of tests selected. The trade-off suggests that a machine learning model can drastically lower the amount of test selected by a slight reduction in recall rate. The results for EALRTS are based on one case study, 725 test runs with a project consisting of 808 Java-files.

Sammanfattning

Regressionstestning är processen för att bekräfta att en kodändring inte införde något testfel för projektet. En regressionstestningsteknik som vanligtvis används är Regression Test Selection eller RTS. Det är processen att identifiera alla tester som påverkas av en kodändring. Syftet med RTS är att minska utveck- lingstiden genom att sänka tiden för testning. Maskininlärning har använts som ett verktyg för testval i nyligen genomförda studier och har visat lovande resultat. Maskininlärning användes med ett RTS-verktyg för att ytterligare minska anta- let utvalda tester. För att träna en maskininlärningsmodell är det dock viktigt att ha mycket data och kodändringar som har introducerat testfel. För open- source projekt är det emellertid utmanande att hitta stort antal kodändringar som ger testfel. Den här studien presenterar EALRTS, ett testverktyg som kan förutspå vilka tester som behöver köras. EALRTS använder mutation gene- ration istället för befintliga felaktiga kodändringar. EALRTS lyckades mins- ka antalet utvalda tester med 60,3 misslyckade test. Resultatet antyder att en maskininlärningsmodell kan sänka mängden test som valts genom en liten minskning av felaktiga test som hittas. Resultaten för EALRTS är baserade på en fallstudie, 725 testkörningar med ett projekt som består av 808 Java-filer.

Contents

Chapter 1

Introduction

Regression testing is an approach to ensure that a code change has not introduced any regressions. After a code change, A Regression Test Selection, RTS, tool finds the tests affected by the code change. The tests not affected should have the same result as in the prior run. A typical regression testing requires two sources of information – The dependency information, i.e., necessary elements of a program needed to execute a test, and the changed elements of the program [1–4]. – This approach can be useful to help allocate limited machine resources, e.g., Google has over 150 million tests running every day [5]. Hence, reducing the number of tests can lower the lag time between check-ins and feedback and as a result, reduce the time to production. Öqvist et al. [1] suggested that safe RTS tools can be combined with other unsafe RTS tools to reduce the time of production. An RTS tool is said to be safe if it runs all tests that have been changed as a result of a code change.

In this thesis, we introduce EALRTS, which is a static, and file-level Predictive Regression Test Selection, PRTS. EALRTS does not require a historical faulty code change. Instead, EALRTS uses mutation generation for code defects. PRTS, similar to RTS, identifies the affected test files given a code change. A PRTS tool estimates the probability of a test failure and tests are selected based on the probability of test failure, which is calculated by a machine learning model. EALRTS can be viewed as an unsafe RTS tool. The reduction of tests comes at a price of safety but hopefully, with a reduction in the end-to-end the testing time. A safe RTS tool selects all tests that are affected by a code change.

EALRTS consists of two sub-methods: (1) the data extraction process, (2) and machine learning as a test selection tool –. The goal with the data extraction process is to extract data from multiple test runs given that code defects are inserted into the project. The code defects were, in this case, created with mutation generation. Multiple mutants are then selected and inserted into a maven-based java project. The number of mutants is supposed to represent the changes made by developers. EALRTS then uses STAtic Regression Test Selection tool, STARTS [3] to run tests and extract data from STARTS's dependency graph and the test result reports. The extracted data was then used as input to a machine learning model to predict the probability of a test failure and then selects what test to run.

The evaluation of EALRTS has been on one maven-based java project, that is commons-math. When mutants inserted to the project, EALRTS reduced the selected test by 60.3% with a 95% recall rate, compared to STARTS. The source code and dataset extracted are available on Github.[1]

1.1 Problem statement

There is a lot of existing research in RTS [1–4, 6–8]. However, there is less research in Predictive Regression Test Selection, PRTS [5, 9]. Previous researches in PRTS used historical faulty code changes made by developers [5, 9]. Memon et al.[5] proposed a set of features that were then used by Machalica et al.[9] which extracted data from a basic regression test selection tool that found every test affected by a code change through static dependencies. The extracted data was then used as an input to a machine learning model to predict test failures. However, no approach was presented on how to train a machine learning model without historical code defects. Also, open source projects with an appropriate size and with historical code defects are hard to find. Even when code defects are available, there are usually too few to achieve statistical significance [8]. This becomes a problem in supervised machine learning since a lot of training data is required.

In this thesis, the problem we address is two fold. (1) How to address the issue of missing historical code defects. (2) How to predict what tests to select given generated data. This resulted in the creation of EALRTS (Erik Alexander Lundsten Regression Test Selection).

1.2 Novelty

First, there are three aspects of novelty in this thesis:

[1]https://github.com/kth-tcs/kth-test-selection/tree/master/eal%20predictive%20rts

This is the first to report an approach of how multiple code defects can be generated and inserted into a maven-based java project. Previous research has only considered using historical code defects created by developers[9]. This thesis used mutation generation to create code defects and select multiple code defects to insert into a maven-based java project.

This thesis also presents an approach of how to extract the features, for the machine learning model, with the help of STARTS. Previous research used a set of features in different conditions: without using an open source RTS, used historical code defects, and little information about how to extract the features [9]. This thesis presents the process of how to obtain the features from STARTS. More specific, an approach of how the data can be extracted from multiple tests runs when mutants are inserted into a maven-based java project.

The thesis is also the first to evaluate the predictive performance of two machine learning models on the generated dataset. The models are evaluated in terms of the number of tests selected, AUC roc-curve, and recall. Where recall explains the number of test failures found compared to STARTS. Then a comparison between the machine learning models are made and how it compares to STARTS in regards to these metrics.

1.3 Research questions

The research questions are divided into two main areas, which are explored in this thesis. The areas are data generation and machine learning as a test selection tool. The distinction is made since machine learning as a test selection tool is a result of the data generation process.

Data generation

In order to evaluate how to determine how many mutants can be inserted into the project, the following research question will be answered:

RQ1: What is a good number of mutants to be used for data generation in EALRTS?

The features explored in this thesis require data such as minimum distance, file cardinality, connected file cardinality, and target cardinality. (1) The minimum distance is how far away a change and a test target are from each other in terms of dependencies. (2) File cardinality is how many files changed (3), And connected file cardinality is how many files given a change transitively depend on a test target. (4) The test target is how many tests are affected by a given code change. Moreover, in the context of training a machine learning model, it is required to have data from multiple test runs, where a test run is the executed test suite given a code change.

RQ2: How can minimum distance, file cardinality, connected file cardinality, and target cardinality be extracted from the STARTS to represent the code changes made?

Machine learning as a test selection tool

The metrics used to explore machine learning as a test selection tool are recall, selection rate, time to select tests, AUC roc-curve. The recall says how many of the individual tests targets were found. A test target consists of one or more test cases. Selection rate is how many tests were selected by the PRTS tool compared to an RTS, which is considered the ground truth. Finally, AUC roc-curve is a metric to distinguish two metrics, which in this case is recall and selection rate.

RQ3: When predicting what test to select with machine learning, how does Random Forest and XGBoost compare to each other in terms of recall, selection rate, time to select tests, and AUC-roc curve?

As a result of the research question 3, the best performing machine learning model will be chosen for EALRTS and compare against STARTS. The following research question will be answered to evaluate EALRTS:

RQ4: How well does EALRTS improve STARTS in terms of selection rate, recall, change recall, and time to select tests?

1.4 Structure of the thesis

The thesis explores two different areas, a data generation process and machine learning for test selection. Chapter 3 presents the theoretical background. Chapter 4 is an analysis of the state-of-the-art and motivations for the selected tools used in this thesis. Chapter 5 describes the methodology for EALRTS and the evaluation methodology. Chapter 6 presents the methodology for machine learning for the test selection. Chapter 7 presents the results and answers to the research questions. Finally, Chapter 8 presents the discussion and conclusion. Which brings up limitations with this approach and what needs to be further researched.

Chapter 2

Background

2.1 generation

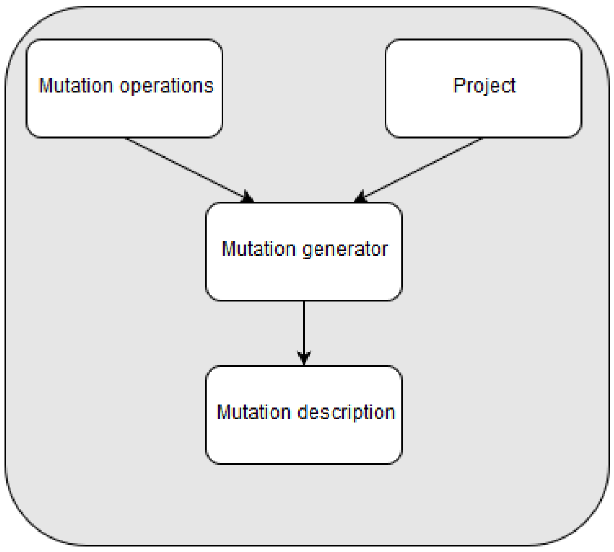

Figure 2.1: Overview of the mutation generation

Researches in regression testing often require changes to the program with code defects. However, one issue is that open source projects rarely have this data available or it does not exist. An approach is to generate data with a technique called mutation generation, which changes a file to be a defect. The code defect can represent faults similar to the faults introduced by developers. Thus, it is useful for reproducible experiments. [8]

Mutation generation is the process of manipulating code in order to introduce defects in the form of small code changes. The change introduced should result in unexpected behavior which should raise test failures. Mutant generation is created by applying mutation operation to the original program; these mutations are syntactic changes. The operators changed often includes[8, 10]:

- Changing constant C to 0,1, -1, (C+1), (C-1)
- Change arithmetic, relational, logical, bitwise logical, increment, decrement, arithmetic-assignment operators by another
- within the same operator-class.
- Negate the decision in "if" or "while" statements
- Delete statement

The mutation occurs at either a source code-level or bytecode-level. Coles et al.[10] concluded that bytecode-level seems to generate mutants quicker compared to source code-level for java. One way to generate mutations efficiently is to divide it into two stages. First, a scan of the project is initialized, then all possible mutants are identified to store a description of the mutant. The description can then be used to recreate the mutants [10, 11].

Coles et al. [10] developed Pitest that provides a state-of-the-art faulty test detection which utilizes mutants to see whether a test finds the anomalies that Pitest creates. However, it can also be used only to generate code defects or mutants. The created mutants are at a bytecode-level, which resulted in the mutation generation process is computationally inexpensive. The mutation generation consists of two parts: (1) all classes are examined within the system under test, and stores the identified mutation points to memory. (2) At the same time, the mutated bytecode-file are generated during this process and immediately discarded. This information about the mutation is enough to recreate all the mutants previously discarded. It also enables Pitest to store possibly millions of mutants without overloading memory. However, it is possible to persist mutants to disk via command line options. [10]

Furthermore, the tool was used in a similar work done by Zhang et al. [12], which used Pitest in order to predict mutation result. Moreover, EALRTS uses Pitest because of the possibility to generate mutant quickly, persisting mutants to disk and the active development of the tool. The Pitest will be the mutation generation tool in this thesis.

2.2 Regression test selection

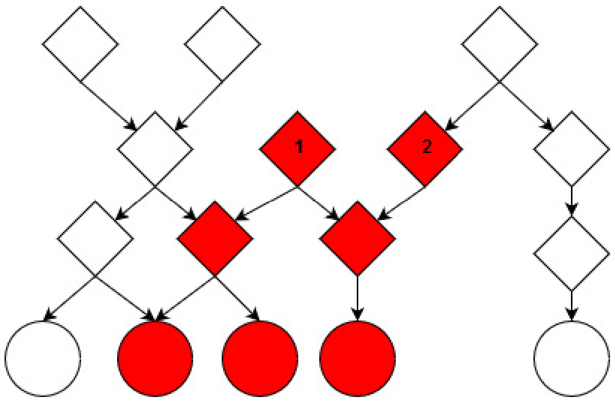

Figure 2.2: Diamonds represent source files and circles represent the test target. An arrow is connected from X –> Y if Y depends on X, or X impacts Y. Two source files are changed in this picture and are marked red with the number 1 and 2. All files that transitively depend on the given change are red. Finally, the selected tests are the red circles and are selected given the code change, i.e., the test transitively depend on the code change.

To retest all tests given a code change is an expensive process which is why researches have been made to reduce the number of tests that will not fail [1–4, 6–8]. Regression test selection, RTS, is the process of detecting regressions for a code change. When developers change the code, it is crucial that the code changed did not introduce any new regressions into the working build. A typical technique within regression test selection requires two sources of information [2–4]. First, it needs to compute the dependency information of the given system under test, these dependencies states what files are affected given the code changed. Second, it needs to identify what files have changed, which is done by comparing checksums of the files between two versions of the program.

Figure 2.2 presents a basic file-level RTS tool, which pictures a project with dependencies, two changed files, and what test files are affected, i.e., the red circles. The figure represents a safe regression test selection technique, i.e., the tool selects every test affected by a code change [1]. RTS can reduce the time spent by executing only a portion of the entire test suite [6, 13].

Test dependencies can be collected either as static or dynamic. Legunsen et al. [3] used a static approach to select tests, and Gligoric et al. [2] used a dynamic approach to select tests. The static approach uses static analysis to approximate what test to run. In contrast to the dynamic approach that uses run-time information in order to determine what tests to run. Further, the gathered information for both static and dynamic RTS techniques is at different granularities. Research made have shown that file-level have a less end-to-end time than a method-level technique even though it can select fewer tests. [2–4].

2.2.1 Static test selection approach

Legunsen et al. [14] compared two static approaches to test selection, one at a class-level and another at a method-level. They concluded that class-level outperformed the method-level and were comparable to the state-of-the-art dynamic RTS, Ekstazi [2]. Legunsen et al. came up with a class-level Static RTS or STARTS. It is a tool which uses compile-time information to build a dependency graph to find the impacted tests. The impacted tests are the ones that transitively depend on the changed code. They further described a static approach in five steps: (1) Find type dependencies (2) Construct the type dependency graph (3) find the code change between two revisions (4) store checksums for the current revision (5) select tests impacted by the code change and optionally run the tests [3].

Extraction-based is another static RTS tool that was developed to reduce overhead costs compared to running all tests. Extraction-based RTS is a test selection technique that uses static analysis to select tests. This method uses a dependency graph to select tests and does a minimal extraction of the test program, i.e., a subset of files that is enough for running the test. Further, it is also a coarse-grained and incremental method, meaning that the dependency graph incrementally changes after update to the program is made. Further, a discussion was made in the trade-off between a safe RTS and reducing the safety of the RTS tool to select fewer tests. That is, RTS tools can be combined with RTS methods to reduce the number of tests further[1]. However, the documentation of the tool is lacking, and already implemented functionality was limited.

2.2.2 Dynamic test selection approach

Ekstazi is regression test selection tool which is considered state-of-the-art [1–4, 6, 7]. Ekstazi was developed to be more easily adopted into existing testing frameworks and build systems. The purpose of the developed tool was to increase the adoption rate. Ekstazi tracks dynamic dependencies and only select tests that are affected by these dynamic dependencies for each test class.

Ekstazi then stores the name and checksums of these classes. The information stored dependencies for each test class in a separate file. Ekstazi then analyzes if the checksums remain the same. A test is executed if any file has changed for a given test class. On average, Ekstazi reduced the end-to-end testing time by 32% compared to running all tests [2]. While it is considered state-of-the-art, this project was not open source for the more significant part of the time of this thesis.

2.3 Machine learning

2.3.1 Data preprocessing

Data preprocessing is the technique used to transform data into an understandable format for a machine learning model. The technique can manage data that is incomplete, inconsistent or contains errors. Different techniques exist to solve such issues [15–18].

Data cleansing is the process of cleaning the data by dealing with missing values, removing outliers, smoothing noisy data. Missing data can occur because of collection error or corrupted data. A dataset with relatively low missing values, the inputs can be removed. If the missing values are high, it is possible to implement imputation techniques as described by Donders et al. [15]

Binning is the process to reduce the number of possible values in the data. An example of a reduction is to introduce categories, or an interval, where each interval is a category. The reduction will help with problems such as outliers, minor data errors, and preparation of the data. [18]

Data reduction strategies are used to reduce the number of features without losing its originality. An approach for reducing a smaller set of features is through a brute-force wrapper method. Traditionally described, a wrapper method evaluates the predictive performance for a subset of features. In practice, the following needs to be defined: (1) How to search for the possible subset of features (2) How to assess the predictive performance of the machine learning model (3) The predictor to be evaluated. An exhaustive feature selection approach uses a brute-force search which is suitable if not the number of features is too large.[17]

Data transformation is the process of transforming data into another format. The purpose of data transformation is to enhance the data in such a way that it increases the likelihood that the machine learning model to find meaningful patterns. Normalization is one such technique where values convert into a normalized range. Another approach is one hot encoding where the categories transform into features with the value one or zero, i.e., true or false.

2.3.2 Gradient boosted decision tree

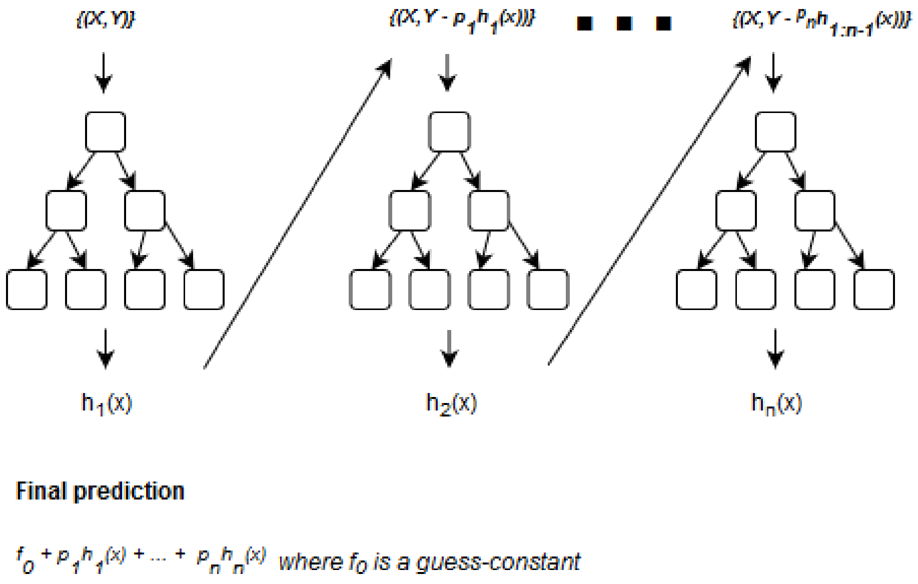

Figure 2.3: Illustration of the gradient boosted decision tree prediction process

The main idea behind boosting is to sequentially add new weak models, decision trees, to an ensemble that can produce accurate predictions. At each iteration, a newly trained model tries to minimize the error of the whole ensemble learned to this iteration. Figure 2.3 is an illustration of the prediction process of a gradient boosted decision tree.

The goal is to map input features $x = \{x_1, ..., x_d\}$ to an output y and to reconstruct unknown functional dependence $x \xrightarrow{f} y$ with a functional estimate $\hat{f}(x)$ that minimizes some loss function $J(\theta) = \Psi(y_i, f(x_i, \hat{\theta}))$. For gradient boosting, each functional estimate is a weak learner that is sequentially fitted on the previous ensemble's residuals. The minimization of the residuals is done with gradient descent of the loss function $\nabla J(\theta)$. This process of adding weak learners and fitting on the previous ensemble's residuals is repeated until the numbers of estimators are reached. [19]

Gradient boosting algorithm

```
Initialization
1: Import the data $x = \{x, y\}_{n=1}^{N}$
2: Define the number of iterations
3: Choose a loss-function $\Psi$
4: Choose a base learner $f$
Algorithm
5: Make the initial guess $\hat{f}_0$ as a constant
6: For t=1 in range M
7:          Calculate the gradient the negative gradient $g_t(x)$
8:          train a new decision tree $f(x, \theta_t)$
9:          Calculate the gradient descent step-size $\rho_t$

                $\rho_t = \arg\min \sum_{n=1}^{N} \Psi[y_i, \hat{f}_{t-1} + \rho f(x_i, \theta_t])]$

10:          iterate the function estimate:

                $\hat{f}_{t-1} + \rho_t f(x, \theta_t) \rightarrow \hat{f}_t$

11: end
Note that $f(x, \theta_t)$is parallel to the negative gradient $g(x)$ [19]
```

2.3.3 Random Forest

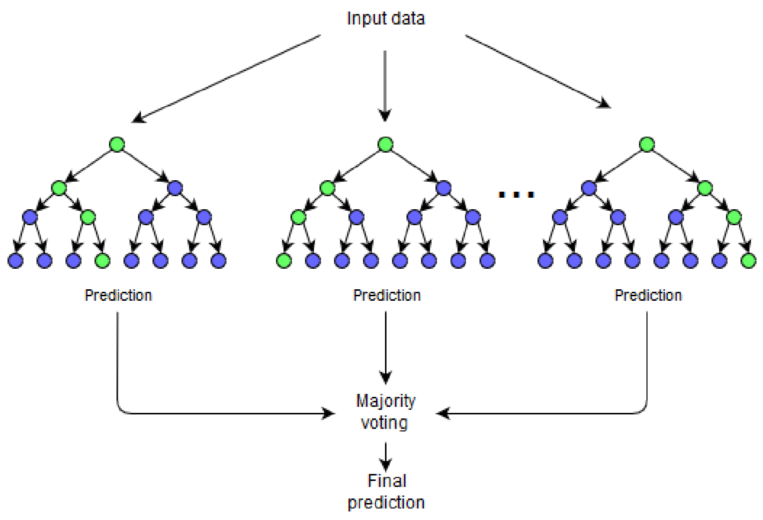

Figure 2.4: Shows the predictions process of random forest, the green circles represent the predictions made by each decision tree in a random forest. The final prediction is the result of the majority voting

The idea of training multiple decision trees in parallel for classification was first introduced by Ho et al. [20]. Picture in figure 2.4 shows a simple version of Random Forest, which is an ensemble technique. The ensemble consists of many decision trees which are trained in parallel and outputs a prediction. The predictions result in a majority vote, and the class with most predictions is the final prediction. Each decision tree utilizes something called bootstrap aggregation, commonly known as bagging. Bagging is the process of selecting a random subsample of the features with replacement, i.e., $m \in M$ where M is the entire dataset. "With replacement" means that that the other decisions tree can select the same samples [21].

```
1: for i=1 to B
2:         draw m samples with replacement
3:         Grow a tree and recursively repeat the following steps until
all terminal nodes are reached:
4:             i) randomly select $m_{try}$ of the predictors
5:             ii) calculate the best split among the $m_{try}$ variables
6:             iii) split the node into two daughter nodes
7: Output: for classification, perform majority vote for all the decision
trees. [21]
```

Chapter 3

State-of-the-art

This section presents an analysis of the state-of-the-art and the motivation of the tools and models used in this thesis.

3.1 HyRTS: Hybrid Regression Test Selection

Zhang, L. [4] studied the strengths of dynamic and static RTS and at different granularity levels. They found a way to combine the strengths of the different level of granularities in order to create a Hybrid Regression Test Selection, HyRTS. The basic technique behind HyRTS is to perform finer-grained method-level analysis on code changes while performing file-level analysis on additions, deletions, or class file header changes. It selects tests at a method- and file-level analysis of the test dependencies. The selected tests are then executed, and method-level dependencies are gathered for future revisions. Also, the file-level dependencies can be derived from the method-level analysis.

HyRTS currently supports maven-based java projects at a test class level. The three components in HyRTS are (1) *change computation*, which finds the changed file by getting the checksums for the bytecode files. Also, HyRTS tracks method-level changes. (2) *dependency collection*, the method dependencies are gathered with a java agent. The file-level dependencies can then be derived from the method-level dependencies. The dependencies are gathered from a test-class level since test methods are hard to find in practice. (3) *Application modes* the selected tests can then be derived from the code changes identified and the collected dependencies.

However, the code of this experiment is not open source and can, therefore, not be used in this experiment.

3.2 STARTS: STAtic Regression Test Selection

In a previous study by Legunsen et al.[14] found that Static RTS was comparable to the dynamic state-of-the-art RTS tool, Ekstazi. However, at this time, static RTS were less precise and could be unsafe. Precise is if it only runs the failed tests and safe if it contains all the failed tests. The paper also found that class-level static RTS outperformed method-level static RTS. The conclusion sparked the idea of STARTS. Legunsen et al.[3] then created STAtic regression test selection tool, or STARTS, which is a state-of-the-art static RTS tool for a maven-based java project. STARTS use compile-time information to select tests, in more depth STARTS builds a dependency graph relating all types (Classes, interfaces, and enums) and finds impacted tests that depend on the given change. It builds this dependency graph with the graph tool yasgl[1], STARTS computes the transitive closure for each test and finds all its dependencies. STARTS can determine what files have changed by computing the checksums for each file and compares it to the checksums of the previous run. However, STARTS might be unsafe if the paths between tests and changed files only reach via reflection.[3]

STARTS also brings a variety of options, which makes it easy to work with, such as only finding what impacted tests that changed since the last run. STARTS also brings the option to find and run impacted tests. Another critical option that STARTS brings is the option not to update checksums after a test run. The test tool can be used not to store the checksums for a version of the program with inserted mutants. If the original bug-free file then replaces the mutants, the test tool will notice a change even though this piece of code is working. The option not to update checksums is, therefore, an essential option for this thesis. [2]

3.3 Features for predictive regression test selection

Memon et al.[5] proposed features that indicated if a file were prone to breakage. They found that very few tests ever fail, but those that fail are closer to the code change. With a code based represented as a dependency graph, Memon

[1]https://github.com/TestingResearchIllinois/yasgl
[2]https://github.com/TestingResearchIllinois/starts

et al. found that the test target is within a distance of 10 in terms of edges. Furthermore, other features have frequently modified the code, individual users and tools cause more breakage and lastly, code recently modified by more than three developers could indicate that a file is more prone to breakage.

The following features were used in this thesis based on Machalica et al.[9], and Memon et al.[5] and are further divided into three categories. The change-level feature consists of features given the code change:

- *Change history* for files, how many times have a file changed. In order to identify areas of development which are more prone to breakages.
- *File cardinality* is the number of files touched by a change. Large changes are harder to review, and smaller changes are less likely to fail.
- *Target cardinality*, number of files (test targets) touched by a change. A widely used particular file might trigger unexpected behavior.

Target level features are the single test target that perhaps needs to be further tested. These features consist of:

- *Failure rates*. A measurement of how good the test target performs.
- The *number of tests*, which is an indication of the code area coverage.

Cross features are features depending on test target and the code change:

- *Connected file cardinality*, how many files are having a transitively depend on the given a test target.
- *Minimum distance*, the distance between the changed files, and the selected testing target. A closer distance is more prone to breakage given a change.

3.4 Model selection for predictive regression test selection

Model selection for predictive regression test selection Machalica et al.[9] conducted their experiment at Facebook and based their features on the paper by Memon et al.[5]. The suggested features were input for a gradient boosted decision tree model. Gradient boosted decision tree is a favorite machine learning model that works out of the box, requires no normalization, and can deal with imbalanced data. Therefore, a gradient boosted decision tree is one model evaluated in EALRTS.

Zhang et al. [12] researched predictive mutation testing, PMT, which aims to predict mutation testing results without executing the mutants. The model used in this research with the reason is that it is a robust model that it is practical to deal with imbalanced data. Logistic regression is another model used for classifications tasks. However, Muchlinski et al.[22] suggested that Random Forest may perform better than logistic regression for imbalanced datasets. Therefore, Random Forest will be the second model to be compared for this thesis.

In this thesis, the models used are Random Forest, and gradient boosted decision tree. These models both have support to handle imbalanced data, which is why the models are chosen over logistic regression.

Chapter 4

Methodology

This chapter consists of two aspects of the methodology, an approach- and an evaluation- methodology. The approach methodology explains how EALRTS works. It consists of two main areas, generating data and selecting tests with machine learning. The data generated is used as input for a machine learning model to predict what tests to run. First, code defects, or mutants, were created with Pitest. Then multiple mutants were inserted into the project and STARTS found the changed files by comparing checksums with the original project, i.e., commons-math. STARTS then finds the affected tests by searching the dependency graph that it creates. At this time, a modified STARTS outputted data from the dependency graph. Once finished the test run, the project was reset back to normal, and a new set of mutants was inserted. This process was then repeated and resulted in a dataset. Each entry in the dataset was a test target, and the class was whether or not the test failed. The dataset was the input for a machine learning model. The goal of the machine learning model was to predict if a test needs to be tested or not.

The evaluation methodology presents the evaluation process of EALRTS. It consists of the two main areas, the data generation and selecting the machine learning model. The data generation describes what metrics were extracted to evaluate the data generation process. The selecting machine learning model presents the process of how both the models were optimized, and how the models were evaluated.

4.1 Approach methodology for EALRTS

EALRTS uses generated mutants to extract data from STARTS and then use the data as input for a Random Forest classifier to predict what test select. Figure 4.1 presents the process of the entire data generation process. The data to be extracted is presented in section 4.1.1. The process of generating mutants is presented in section 4.1.2. Then the process of using mutants to extract data is presented in section 4.1.3 and finally, the test selection with machine learning is presented in section 4.1.4.

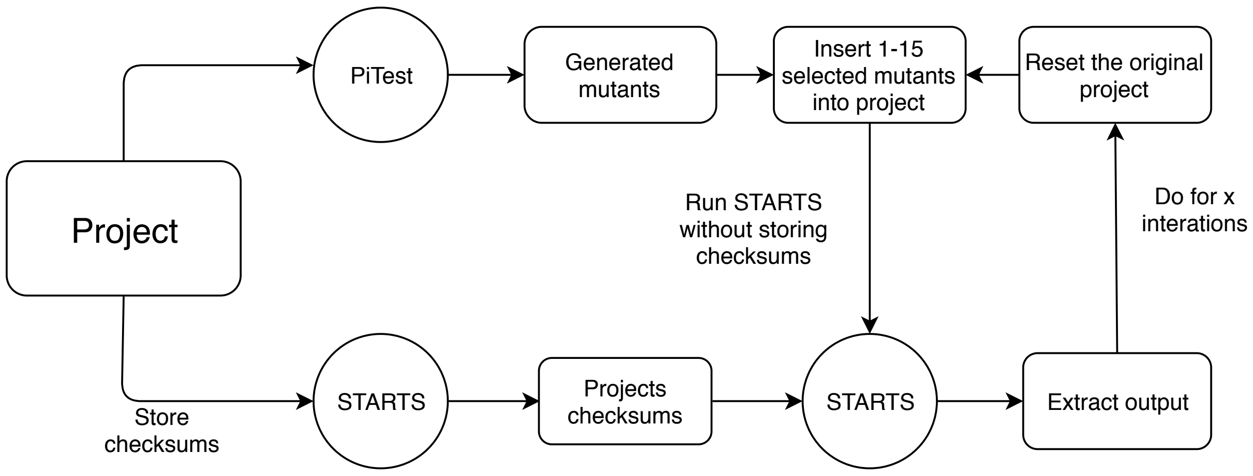

Figure 4.1: Data extraction process of EALRTS

To go into more detail, pseudo-code of the EALRTS is also provided in algorithm 1. The pseudo-code explains the individual steps of figure 4.1 and it also includes test selection with machine learning.

Algorithm 1: pseudo-code for EALRTS

```
Result: A selection of tests with machine learning
#Data generation;
Generate mutants with Pitest;
Run STARTS and store checksums for the original project;
int num_runs = number of test runs;
for i in num_runs do
    int num_files = random number between 1 and 15;
    for j in (num_files) do
        Select a random file from original project;
        Select a random mutant for the file;
        Replace the file with the mutant;
    end
    Run & extract data from STARTS;
    Replace mutants with the original files;
end
#Test selection with machine learning;
Read the extracted data;
split data into evaluation set and a training set;
train a Random Forest model on the training set;
List<float> probabilities = predictions on evaluation set;
for j in (num_files) do
    if (probability> (probability cut off)) then
        test selected to run;
    else
        test not selected to run;
    end
end
```

4.1.1 Extracting features

A machine learning model requires input data in order to find patterns for making predictions. Input data consists of features, and the label is what we want to predict. The features for this experiment was about the code change, test target, and cross-dependent features, which are features depending on both the code change and test target. Moreover, the label is whether a specific test target failed. A test target is a test class containing tests cases. The extracted features used in this thesis are based on previous research [9]:

Code change features:

- *File change history* is how often each of the changed files been changed in the past. File changed more often might be more prone to breakage.
- *File cardinality*, number of files affected by a change. Larger changes can indicate a file is more prone to breakage

Test target features:

- *Failure rates*, how often the tests have failed, given all the test runs.
- *The number of tests* can be an identifier of code coverage.
- *Target cardinality*, number of tests selected given a change.

Cross dependent features:

- *Connected file cardinality*, number of files touched by a code change and have transitive closure to a test target.
- *The minimum distance* is the shortest distance in terms of edges from a change to a test target.

4.1.2 Generating mutants with Pitest

The data was extracted with the help of STARTS. By default, STARTS does not have an option to extract these features, so modification to STARTS was made in order to extract these features. Although File change history was obtained through the project's Github.

For this experiment, the machine learning model requires input from three sources of information: (1) A code change with defects and (2) test failure caused by the code change. (3) cross-dependent features between the code change and the test target that failed[9]. In other words, code defects are required in order to train a machine learning model. However, finding open source projects with an appropriate size and with historical code defects are hard to find. Even when code defects are available, there are usually too few to achieve statistical significance. When lacking historical code defects, two approaches are available. Faults can either be introduced by hand or automatically with a mutation generation tool [8]. Also, when a developer makes a code change, it can consist of multiple files. This experiment recreated this by inserting multiple mutants into the code.

Generated mutants

Pitest is a state-of-the-art tool to generate mutations that are seeded into the project and then run the tests. For this experiment, the mutations were extracted, and the execution of the tests by Pitest was skipped. The mutants generated by Pitest are at a bytecode-level, which is more time efficient than mutants generated at a source-code level[10]. The name of the mutations is equal to the classpath, which makes it easy to find the changed file. Alongside the mutants, comes details of what method have changed, what line, and the operation that changed.

Mutants inserted into the project

EALRTS replaced a random number of files with mutants in the original project. The random number of files to be replaced by mutants have an upper limit, which is found by analyzing the *buildSuccessRate* when a certain number of mutants are inserted. For this experiment, the maximum number of files to be replaced by mutants was 15, which is further described in section 4.2.2. EALRTS then selected the mutants generated by Pitest and inserted them into the project. To reduce the sample bias[23]; A file cannot be replaced again until EALRTS have replaced all files in the project. Once STARTS has finished its test run, EALRTS replaces the mutants with the original files. Moreover, if the project failed to compile with the inserted mutants, then the original version of the project is restored. Also, a new number of mutants are selected instead in order to reduce the risk of a sample biased dataset.

Another approach to reset the project could be to recompile the project. Also, mutants could be created generated at a source code-level instead of a bytecode-level.

However, recompiling the project takes significantly longer than simply inserting the compiled file again. Also, research suggests that bytecode-level mutants are more time-efficient than those generated at a source code-level[10].

4.1.3 Extracting data from STARTS

Stored checksums

To identify a code change, STARTS compares two versions of the program an old and a new one by storing the checksums of the two versions. If a checksum for a file is different between an old version and a new version of the program, the file is considered as a changed file. The checksums in this experiment are only stored once; in this case, this would be the original project checksums. The checksums are only stored once since the removed mutants from the last iteration, or an old version, caused a difference in checksums. Also, EALRTS configured STARTS not to store the checksums by default.

Using the graph library Yasgl to find data

By default, STARTS builds a dependency graph, and the test selected is on a class-level. The dependency graph that STARTS builds is created the help of Yasgl, yet another simple graph library, where each node is a type.[1] Yasgl is used in STARTS to find the affected test through the changed files.

EALRTS modified STARTS to extract data from the dependency graph with the utilities that Yasgl brings. For this thesis, Yasgl was a convenient way to find the data needed, such as minimum distance between changed files and a test target, and connected file cardinality. A breadth-first search was conducted on the dependency graph to find the minimum distance and the connected file cardinality. Also, the file cardinality and target cardinality was already identified by STARTS and required minor changes to be extracted.

Extracting data

The modified STARTS tool extracted data at run-time, i.e., connected file cardinality, file cardinality, target cardinality, minimum distance, and the name of the tests that failed. Once STARTS finished executing the tests, it outputted the test result reports with data such as the number of tests and test failures. EALRTS then combined the test result reports with the data that STARTS extracted at run-time and saved to the /erik-files directory. Each entry in the dataset was the result of a single test target,t, and is labeled positive if and only if $t \in FailedTests(c)$ where c is a changed file. If the test target has failed, it is labeled true; otherwise, it is labeled false. Once the data is com-

[1]https://github.com/TestingResearchIllinois/yasgl

bined, EALRTS resets the project to its original version and deletes the test result reports in the surefire-reports.

There are two benefits of deleting the surefire-reports that modified STARTS outputs. (1) The 725 test runs outputted 82437 individual data points or 82437 test result reports, which takes up much memory. (2) The reports output the same name, by deleting the previous test reports makes it easier to track what test result report belongs to a specific test run. Note that these values are for the commons-math project.

4.1.4 Selecting tests with machine learning

A Random Forest model was then trained on the data extracted from the modified STARTS. The model was trained on all features described in section 4.1.1 except the file cardinality. The features were selected with a method called a brute force wrapper-method[17], which is further described in section 4.2.5. The trained model then outputted a probability of a test failing. A test is said to be failing if the probability of failing is above a probability threshold $probability > probabilityCutOff$, where $probabilityCutOff$ is the probability threshold. The tests that had a probability over the threshold are then selected to be tested. The model could then be compared against STARTS in terms of selection rate and recall.

4.2 Evaluation methodology

Figure 4.2 presents an overview of the elements that were required to get evaluation results from the two machine learning models. First, descriptive evaluation statistics about the inserted mutants and data extraction are presented in section 4.2.2 and section 4.2.3. Then the data had to be preprocessed in order to be used as input for the machine learning models. Hyperparameters were then tuned with Bayesian optimization, which is described in section 4.2.4. Optimized hyperparameters increase the model's predictive performance and are required to make a fair comparison between the models. Then a feature selection was conducted to remove noisy features, presented in section 4.2.5. Finally, the model is trained, and the evaluation metrics can be obtained, which is explained in section 4.2.6

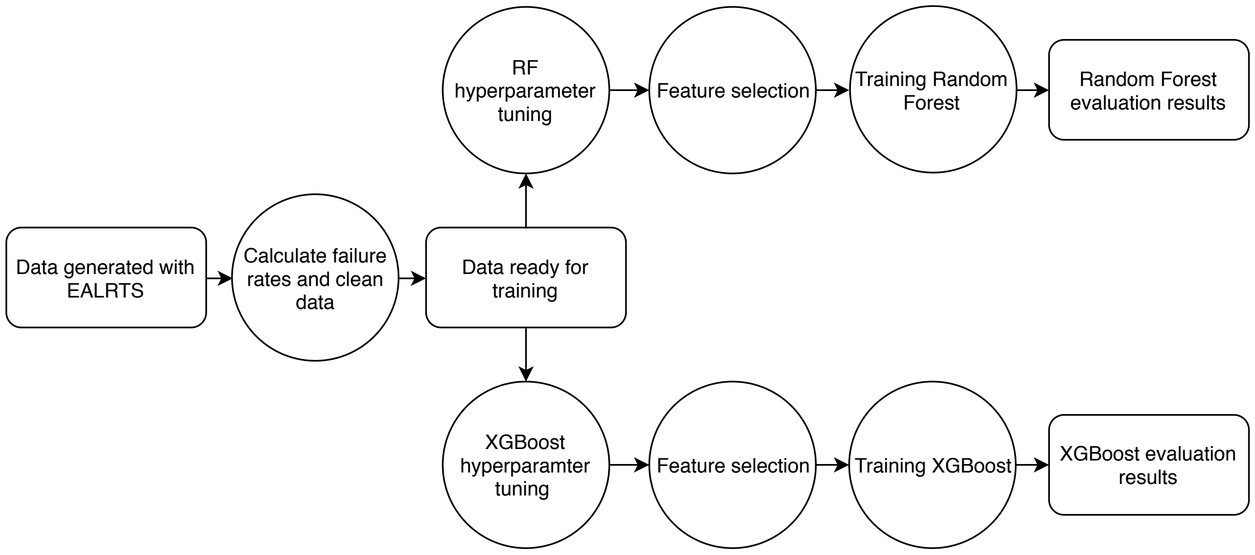
Data generated with EALRTS
Calculate failure rates and clean data
Data ready for training
RF hyperparameter tuning
Feature selection
Training Random Forest
Random Forest evaluation results
XGBoost hyperparamter tuning
Feature selection
Training XGBoost
XGBoost evaluation results

Figure 4.2: Overview of the machine learning process

4.2.1 Evaluation metrics

Build success rate

$$buildSuccessRate = \frac{SuccessfulBuilds}{SuccessfulBuilds + FailedBuilds}$$

Where $SuccessfulBuilds$ is the number of test runs successfully compiled with inserted mutants, and if the compiling failed, it is considered as $FailedBuilds$. This metrics tries to capture success rate of compiling a project, where a higher success rate is generally better.

Recall, change recall and selection rate

Evaluation metrics for predictive regression test selection have been proposed by previous research[9]. The suggested metrics are used in this paper to evaluate a machine learning model are *recall*, *change recall*, and *selection rate*. These metrics are defined as:

$$Recall = \frac{selectedtests \cap failedtests}{failedtests}$$

The selected tests are the test selected by the PRTS tool and failed tests are all the failed test found by the RTS tool, i.e., STARTS. Generally, a higher recall is better, and it means that the model finds more individual test failures. That is it finds test failures in a test target.

$$SelectionRate = \frac{selectedtests \cap nonFailedtests}{nonFailedtests}$$

The selected tests are the test selected by the PRTS tool and non-failed tests from the RTS tool. The selection rate is the comparable metric that says how well the PRTS model performs in comparison to a test selection tool. A lower selection rate is better, which means that the model select fewer tests that pass.

$$Changerecall = \frac{selectedtests \cap Failedtests \neq \emptyset}{Failedtests \neq \emptyset}$$

Change recall describes if at least one test failure was found in a test suite by the PRTS. It describes a models performance to find at least one test failure per test run. A higher change recall is better since it means that the model can find at least one test failure per test run. Therefore, a higher probability of capturing faulty changes.

AUC-ROC curve is another technique which uses a two-dimensional depiction of a classifier's performance at various thresholds. Traditionally the two dimensions consist of recall and precision. The area under the ROC curve then indicates how well the two-performance metrics separate from each other[24]. The two dimensions can be replaced, and for this experiment, recall and selection rate was used as the evaluation metric for the AUC ROC-curve.

4.2.2 Evaluating the inserted mutants

Since mutants are traditionally inserted one at a time. Inserting multiple mutants into a project meant a risk of the project not compiling successfully. Therefore, an analysis of the $buildSuccessRate$ had to be made. For this thesis, 30 iterations were made at various thresholds. The results were then presented in a table. The analysis was the trade-off between the number of mutants and $buildSuccessRate$.

4.2.3 Evaluating data extraction process

When the modified STARTS detects code changes, it finds the affected tests through a dependency graph. The code change was multiple mutants inserted into the project, and a set of mutants cannot be guaranteed to compile successfully. Therefore, the average file- and target cardinality was extracted from STARTS to see how inserting multiple mutants affects the data extraction process. Then the average failure rate was extracted from STARTS. The average time for build failure, i.e., time for a set of mutants not to compile was also extracted from STARTS.

4.2.4 Tuning hyperparamters

The hyperparameter tuning used in this thesis is Bayesian optimization which is an efficient way to find the hyperparameters.[2] In contrast to random or grid-searches, keeps tracks of historical evaluations of the model and creates a posterior distribution of a function that is will be optimized. The function to be maximized is the mean AUC-score of 5-fold cross-validation. The cross-validation is used with early stopping to reduce the risk of overfitting. Once finished, the best score will be that the parameter will be the parameters for the model. The final hyperparameters will be presented in tables for Random Forest and XGBoost.

The model was then further explored by removing the feature and see how it affects the relative AUC-score which is defined as $\frac{AUC\text{–}score\ (all\ features)}{AUC\text{–}score\ (one\ removed\ feature)}$.

4.2.5 Measuring feature importance

A brute force wrapper-method is an approach for reducing a small set of features. Traditionally described, a wrapper method evaluates the predictive performance for a subset of features. In practice, the following needs to be defined: (1) How to search for the possible subset of features (2) How to assess the predictive performance of the machine learning model (3) The predictor to be evaluated[17].

A brute-force search was used in this thesis since the feature set consisted of seven features. The predictive performance measured was the AUC-score and the logarithmic loss. The logarithmic loss penalizes classifiers that are too confident about a prediction. Predictor used was presented in section 4.1.1.

[2]https://github.com/fmfn/BayesianOptimization

The brute-force wrapper removes one feature at a time and is evaluated with 5-fold cross-validation. The model outputs a score in terms of the selected metrics'. If the model performs better without the metric for both the logarithmic loss and AUC score, the feature is removed.

4.2.6 Selecting machine learning model

The machine learning models used for test selection are Random Forest, and a gradient boosted decision tree model, XGBoost. These models both have properties that are suitable for this use-case: They are both robust, no normalization is required, it does not require high-end hardware and have support to handle unbalanced data.

The evaluation metrics used to compare the machine learning models were AUC ROC-curve, recall, selection rate, and change recall. The recall threshold used for comparison was based on previous research by Machalica et al.[9], which used a 95% recall. Also, this thesis will look at recall thresholds between as 96% to 99% recall threshold to explore the difference in selection rate between the models.

Both Random Forest and XGBoost returns a probability, which is the estimated probability of a test target failing. If $probablity > probablityCutOff$ the test will be selected to run. Also, by trying different $probablityCutOff$ which will result in an AUC-roc curve. The two models could then be compared to each other in terms of AUC-score and roc-curve.

The predictive performances were further explored by comparing the two machine learning models to a random test selector. The random test selection approach used failure rates to select what test to select. If a failure rate were above a threshold, the test was selected. Doing this for different thresholds resulted in a roc-curve which could be compared to the two machine learning models.

Chapter 5

Results and analysis

This chapter presents the results and analysis of the results in two sections. First, the result of the data generation and an analysis are presented in section 5.2. Second, the result of the machine learning as a test selection tool and analysis is presented in section 5.3. The analysis answers the research questions stated in this thesis.

5.1

Table 5.1 presents information about the dataset. Similar research used all mutations for predictive mutation testing [12]. The data was collected over two weeks. During that time, a total of 1323 set of mutants were inserted into the project, 725 of those could be compiled and executed successfully, i.e., 725 test runs. A test run is an execution of a set of tests targets given a code change. The dataset consists of 82437 entries where each entry is a test target, which is a test class. The number of test targets in commons-math is 497. 11.66% of the test targets in the dataset failed, which fails as a result of the inserted mutants. The average time it took for one test run was 176 seconds. The set of mutants that could not successfully be compiled were discarded.

Data generation information	**Values**
Number of java-files in commons-math	808 files
Number of unique test targets for commons-math	497 test targets
Number of entries in dataset	82437 entries
Number of successful test runs in dataset	725 test runs
Total number of test runs in dataset	1323 test runs
Average failure rate in dataset	11.66%
Average test run time	176 seconds

Table 5.1: Presents general information about the dataset. It also presents the information of the project size.

5.2 Data generation

This section presents the answer to research question one and two. The questions are also presented with results to support the answers to the questions. The questions are answered in section 5.2.1 and 5.2.2.

5.2.1 Answer to RQ1

Build success rate

Table 5.1 shows the $buildSuccessRate$ when a number of mutants were inserted into the project which are gathered over 30 test runs. The dotted line is the average build success rate over the entire dataset.

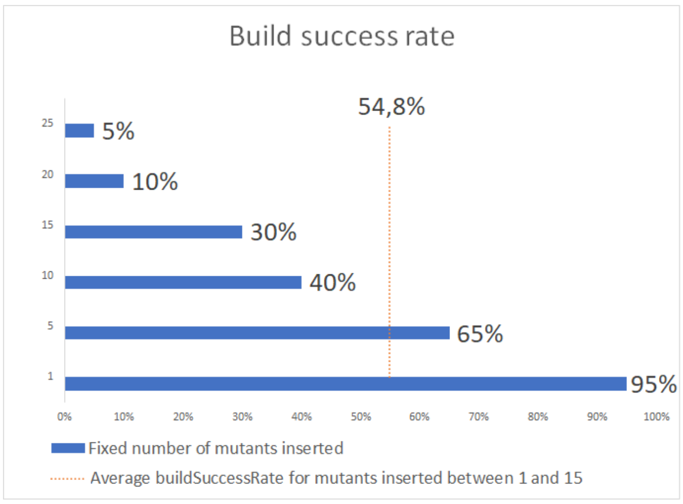

Figure 5.1: build success rate

RQ1: What is a good number of mutants to be used for data generation in EALRTS?

Lacking faulty code changes is typical for open source projects. This thesis presents an approach of how to insert multiple code defects into a project. This approach tries to replicate the number of file changes made by developers by inserting one or more mutants into the compiled version of the project. EALRTS uses Pitest to generate bytecode mutants, and the tool supports maven-based java project. Mutants generated at a bytecode-level are more efficient than the mutants generated at a source code-level [10].

EALRTS generates a random number of files between 1 and 15 in the project, commons-math. Then mutants from the same class are randomly selected to be inserted into the project, commons-math. Once EALRTS selects a file, it cannot be selected again until EALRTS selected all files.

Figure 5.1 shows the build success rates of various thresholds of mutants. The maximum number of mutants chosen for this experiment is 15. The number is chosen by analyzing the $buildSuccessRate$ at a different number of mutant thresholds. The chosen threshold was 15 mutants, which had a 30.0% $buildSuccessRate$. Increasing the maximum number of mutants drastically lowered the $buildSuccessRate$. As shown in figure 5.1, when inserting 20 mutants into commons-math, it has only a 10% build success rate.

The maximum inserted mutants used in this experiment is 15, due to the drastic decrease in build success rate over 15 mutants. In order to have a more extensive variety of code changes, 15 mutants are chosen over ten mutants as well due to a relatively small trade-off in terms of build success rate. By lowering the build success rate increases the time to retrieve the dataset. Every time a set of mutants fails to compile or build, this takes time. For this thesis, the average fail time was approximately 9 seconds.

Further, by increasing the mutants also causes sample bias[23]. If the $buildSuccessRate$ is low, this means that there is a small chance that the build will succeed. The case could be that certain Java classes are more robust to inserted mutants and therefore create a sample bias. However, precautions were taken by not further lowering the build success rate, and therefore 15 mutants were selected.

RQ1: What is a good number of mutants to be used for data generation in EALRTS?

Answer: This thesis presents an approach to creating code defects with mutation generation. The mutants are created with Pitest, and multiple mutants are then inserted into the project. Successfully compiling the project is a challenge when inserting multiple mutants. This $buildSuccessRate$ has to be analyzed to determine how many mutants that can be inserted. For this thesis, 15 mutants were inserted, which had a 30% $buildSuccessRate$.

5.2.2 Answer to RQ2

Data generation information	**Values**
Average test failure rate	11.66%
Average time for build failure	8.92 sec
Average file cardinality	5.9 mutant files
Average connected file cardinality	1.4 mutant files
Average target cardinality	130.8 test targets
Average minimum distance	3.0

Table 5.2: average failure rate = the number of test targets that failed, **average time for build failure** = average time when the project failed to compile with mutants, **average file cardinality** = the average number of mutant files inserted into the project, **average target cardinality** = the number of test targets identified by STARTS given a code change

RQ2: How can minimum distance, file cardinality, connected file cardinality, and target cardinality be extracted from the STARTS to represent the code changes made?

EALRTS integrated a modified STARTS into its data extraction process, which is presented in section 4.2.3. First, the modified STARTS stores the checksums for all files once, which is for the original project. The checksums of the project with inserted mutants are not stored. This is necessary because STARTS will otherwise detect changes caused by the removed mutant. When the new set of mutants is inserted, they are instead compared against the original version of the project and not the previous run. This causes the features to better represent the actual code change made.

Then STARTS were modified to extract minimum distance and connected file cardinality. The extraction from STARTS was done by using the dependency graph that it creates and then conduct a breadth-first search to find the features. This approach of extracting data from STARTS dependency graph represents the possible faulty code changes.

Table 5.2 presents the average connected file cardinality, which is 1.4 mutant files. A larger connected file cardinality can indicate whether a file is more prone to breakage. In other words, the machine learning model will be trained on an average of 1.4 connected file cardinality for commons-math.

Table 5.2 also presents information, such as the average target cardinality, file cardinality, and an average failure rate. The average file cardinality was 5.9 files and given that the same files cannot be updated two test runs in a row, this number would effectively double if the checksums were stored after each test run. Because the next iteration would detect the changes made by the removed mutants, selecting more files for STARTS would cause more tests to be selected. The additional tests selected would also be the cause of time inefficiency, and it would not represent the actual possible faulty code change made. Also, by adding more non-failing tests could be selected, and the average failure rate would therefore decrease. Therefore, adding more tests would create a noisy dataset. The file cardinality can also be an indication of how many files should be changed to make optimal predictions for the learned model. Since this number is 5.9, it can suggest that the model is overfitted on smaller changes.

Further, Memon et al. [5] suggested that the distance in terms of edges in the dependency graph is an indication of a test target failing. Files closer to a test target is, therefore, more prone to breakage. If a developer changes a file close to a test target, it suggests that the test target is more likely to fail.

RQ2: How can minimum distance, file cardinality, connected file cardinality, and target cardinality be extracted from the STARTS to represent the code changes made?

Answer: The checksums of STARTS are stored once for the original project. When a set of mutants are tested, the checksums are not stored. This will reduce the amount of non-faulty affected tests to be found. The features found because of the code change, i.e., minimum distance, connected file cardinality, and target cardinality, can then be extracted with the help of STARTS dependency graph. The features are extracted by conducting a breadth-first search on the graph.

5.3 Evaluation of machine learning models for test selection

This section presents results and analysis for the machine learning as a test selection tool. First, hyperparameters are presented in section 5.3.1. Then the feature importance is presented in section 5.3.2. Once, result and answer to research, question 3 is presented in 5.3.3. Finally, the research question 4 is presented in section 5.3.4.

5.3.1 Hyperparameters

The hyperparameters were set with Bayesian optimization. The hyperparameters used are explained in the two subsections below. Then the values are presented in table 5.3 and 5.5 followed by feature importance.

Random Forest hyperparameters

The parameters modified for the Random Forest model: model[1]:

- *Max depth* increases the depth of the decision tree.
- *The number of estimators* is the number of trees to fit.
- *Max features* are the ratio of the number of features when looking for the best split
- *Min sample split* the minimum number of samples required to split an internal node

Parameter	Value
Max depth	24
Number of estimators	750
Max features	0.04804
Min sample split	0.32508

Table 5.3: Final hyperparameters for the Random Forest model

Table 5.4, shows the importance of the hyperparameters. If the value is over one the feature increases its predictive performance. Since all features

[1]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

have a value above one, the features increased the learned model predictive performance.

Parameter	Parameter importance
Max depth	1.002
Number of estimators	1.540
Max features	1.042
Min sample split	1.331

Table 5.4: hyperparameters importance for Random Forest. The parameter importance is a relative AUC-score compared to when all features are present. All values are above one shows that the value improves the predictive performance.

XGBoost hyperparameters

Table 5.5 presents the final parameters used in the XGBoost model. The parameters modify the XGBoost model in the following way[2]:

- *Max depth* increases the depth of the decision tree.
- *The number of estimators* is the number of trees to fit.
- *Gamma* is the minimum loss reduction required to make a further partition on a leaf node
- *Column sample ratio by tree* is the ratio of features used for each tree.
- *The scale of positive weight* is the balancing of positive and negative weights.

Parameter	Value
Max depth	20
Number of estimators	100
Gamma	0.7592
Column sample ration by tree	0.6823
Scale of positive weight	1

Table 5.5: Final hyperparamters for XGBoost.

Table 5.6, shows the importance of hyperparameters. If the parameter importance is over one, the feature increases its predictive performance. Since all

[2]https://xgboost.readthedocs.io/en/latest/python/python_api.html

features have a value above one, the features increased its predictive performance.

,

Parameter	Parameter importance
Max depth	1.006
Number of estimators	1.014
Column sample ration by tree	1.010
Gamma	1.010
scale pos weight	1.007

Table 5.6: hyperparameters importance for XGBoost. The parameter importance is a relative AUC-score compared to when all features are present. All values are above one shows that the value improves the predictive performance.

5.3.2 Feature importance

Random Forest

Table 5.7 suggested that file cardinality performed better when it was removed. This feature was therefore removed for Random Forest.

Features	AUC-score	Log loss
File cardinality	0.98	0.93
Target Cardinality	1.01	1.00
Minimum distance	1.08	1.21
Number of tests	1.01	1.04
Connected file cardinality	1.02	1.00
Change history	1.05	1.02
Failure rate	1.09	1.13

Table 5.7: Feature importance for Random Forest measured with an exhaustive wrapper method. A score below 0 suggests that it performs better without the feature. Mesured for log loss and AUC-score

Feature importance for XGBoost

Table 5.8 suggests that file cardinality performs better when it is removed. This feature was therefore removed for XGBoost.

Features	AUC-score	Log loss
File cardinality	0.99	0.98
Target Cardinality	1.01	1.00
Minimum distance	1.10	1.01
Number of tests	1.00	1.06
Connected file cardinality	1.01	1.06
Change history	1.01	1.10
Failure rate	1.07	1.20

Table 5.8: Feature importance for XGBoost measured with an exhaustive wrapper method. A score below 0 suggests that it performs better without the feature. Mesured for log loss and AUC-score.

5.3.3 Answer to RQ3

Recall, change recall and selection rate

Table 5.9 shows the selection rate at various recall rates for both Random Forest and XGBoost. It also shows the selection rate and the difference in selection rate between the models.

	10% evlation set split				
Recall	Selection rate(xgb)	Selection rate(rf)	Change recall(xgb)	Change recall(rf)	diff
95%	43.7 %	39.7 %	99.95 %	99.95 %	4.0 %
96%	46.7 %	43.9 %	100.00 %	99.95 %	2.8 %
97%	51.7 %	49.0 %	100.00 %	99.95 %	2.7 %
98%	56.0 %	56.4 %	100.00 %	100.00 %	−0.4 %
99%	66.0 %	68.8 %	100.00 %	100.00 %	−2.8 %

Table 5.9: Data shown in the table is the average over 30 unique evaluation sets, with the evaluation set being 10% of the total data, 10% evaluation set split. **RF** = Random Forest, **XGB** = XGBoost, and **diff** = difference in selection rate the models (diff=xgboost-random forest)

Recall	Selection rate diff, 10% evaluation set	Selection rate diff, 25% evaluation set
95%	4.0 %	5.1 %
96%	2.8 %	6.1 %
97%	2.7 %	5.8 %
98%	−0.4 %	2.8 %
99%	−2.8 %	−0.2 %
Average:	1.3 %	3.9 %

Table 5.10: The difference in selection rate between XGBoost and Random Forest at different recall rates, diff= xgboost-random forest

AUC roc-curves

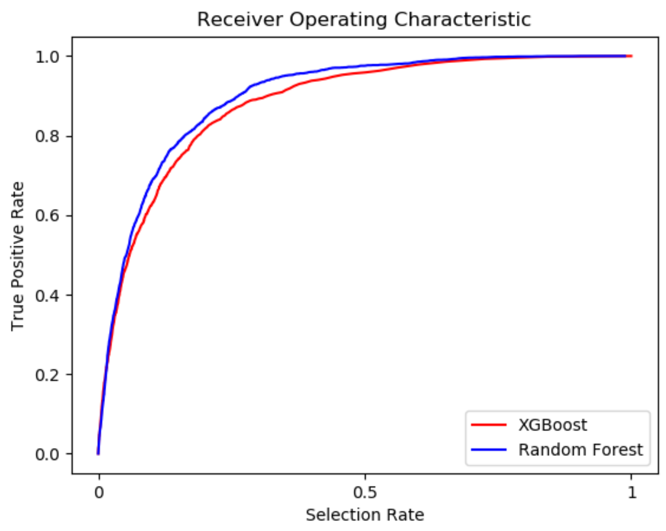

Figure 5.2: XGBoost vs. Random Forest. AUC score for XGBoost is 0.86, and Random Forest is 0.89

RQ3: When predicting what test to select with machine learning, how does Random Forest and XGBoost compare to each other in terms of recall, selection rate, and AUC-roc curve?

Figure 5.2 presents the AUC ROC-curve, where the x-axis is the selection rate, and y-axis are the recall. Random Forest has a higher AUC-score than XGBoost. The score suggests that Random Forest performed slightly better than XGBoost. To make a more thorough comparison, Figure 8 presents a roc-curve comparison between XGBoost and Random Forest. The plotted curves suggest that Random Forest seems to select fewer test than XGBoost as the recall lowered, which is the gap between the curves.

Table 5.9 presents how well the Random Forest performs compared with XGBoost. It suggests that Random Forest outperforms XGBoost, in terms of selection rate, as the recall threshold lowers from 99% to 95%. The data in table 5.9 used a 10% evaluation set. The selection rate for a 95% recall threshold, Random Forest selected 39.7% of the tests while XGBoost selected 43.7% of the tests, with a difference between the learned models of 4.0%. Random Forest still has a lower selection rate for 96% and 97% recall where the difference is 2.8% respectively 2.7%. When increasing the recall threshold to 98%, the difference between Random Forest and XGBoost has lowered to

-0.4%. Finally, XGBoost performs better than Random Forest when the recall rate is 99%.

Figure 5.2 also suggests that the models should perform roughly as equal for higher recall rates. However, as the gap increases between the curves, the Random Forest model performs better than XGBoost. The increased gap is the cause of a higher AUC-score for the Random Forest.

Table 5.10 suggests that Random Forest performs better when lowering the threshold from 99% to at least 95%. Also, Random Forest seems to outperform XGBoost when the recall is lowered. Table 5.10 shows that an increase in the dataset could reduce the predictive performance of the models. However, the average selection difference is higher, which suggests that Random Forest performs better with a reduction in data.

Machalica et al. [9] achieved a reduction of tests by a factor of two and had a change recall of 99.9% while maintaining a recall rate of 95%. They also used historical code defects made by developers as input, had a multilingual repository, and consisting of multiple projects. They tested used it on one repository, which was Facebook's mobile application.

In this thesis, the best reduction of selection rate was 60.3% with a change 99.97% recall when using a 95% recall rate. The failure rate in this thesis was 11.66%. The failure rate is the result of code changes that only consists of mutants, which are supposed to generate test failures. A higher failure rate trivially increases the chances of finding at least one test failure per test run, i.e., a high change recall. However, the failure rate was not disclosed by Machalica et al.[9].

However, there are some differences between Machalica et al. [9] and this thesis. Machalica et al. used a multilingual repository consisting of multiple projects. In this thesis, a single project was used with only one language. Further, Machalica et al. used historical code defects made by developers. In this thesis, one main contribution was to find a solution to this problem by instead using mutation generation. Similar results were achieved in this thesis as Machalica et al., however, they had to deal with multiple languages and a more complex repository.

RQ3: When predicting what test to select with machine learning, how does Random Forest and XGBoost compare to each other in terms of recall, selection rate, time to select tests, and AUC-roc curve?

Answer: For a 95% recall rate, Random Forest has a 4.0% lower selection rate compared to XGBoost. For 96% and 97%, Random Forest has a 2.8% respectively, 2.7% lower selection rate than XGBoost. Then both models perform roughly equal for 98% recall rate with a difference of -0.4%. For 99% recall rate, XGBoost performs better than Random Forest with a 2.8% difference. Random Forest generally seems to perform better than XGBoost. Also, Random Forest performs better in terms of AUC-score.

5.3.4 Answer to RQ4

Time to select test

STARTS took 22.88 seconds to select tests. While the time it took for Random Forest to select tests was 0.01 seconds and XGBoost selected tests in 0.002 seconds. The time is estimated for the machine learning model to select a test for an entire test run given an already trained machine learning model.

ROC-curves
The simple test selector selects tests depending on the failure rates.

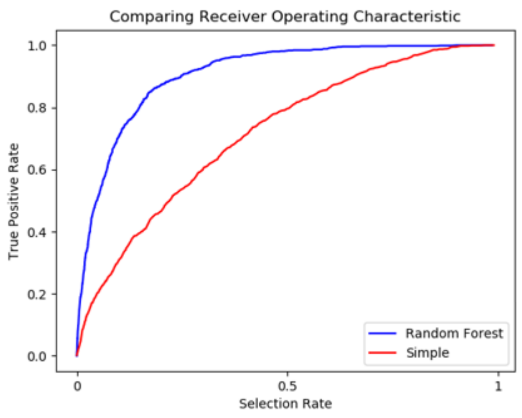

Figure 5.3: Random Forest vs. simple test selector

RQ 4: How well does EALRTS improve STARTS in terms of selection rate, recall, change recall, and time to select tests?

EALRTS is a tool that uses STARTS to further reduces its tests selected. The purpose of EALRTS is to further reduce the amount of tests from an already existing RTS tool, or STARTS. This means that EALRTS cannot find any new test failures that STARTS do not detect. EALRTS is, therefore evaluated on how well it optimizes STARTS in terms of selection rate, recall, change recall, and time to select tests.

Furthermore, the evaluation was made on one case study since a different project would require training on a different dataset. And the feature *failure rate* cannot be transferred between case studies. Also, generating another dataset is computationally expensive. Therefore, EALRTS was only evaluated on one project, i.e., commons-math.

The selected recall used in EALRTS is the same used by Machalica et al.[9], which used a 95% recall rate. From the analysis made for research question 3, Random Forest seemed to outperform XGBoost. Therefore, Random Forest is the model used in EALRTS. Also, the analysis showed that EALRTS could significantly reduce selection rate while maintaining a high recall and change recall. EALRTS could reduce the selection rate by a factor of two, when using a 95% recall rate. Also EALRTS found at least one test failure in 99.97% of the different samples.

Furthermore, to show the predictability of EALRTS, it is compared to a simple test selection tool which chooses a test based on failure rate. Figure 5.3 shows that the Random Forest performs significantly better than the simple test selection tool. It suggests that Random Forest can classify what tests to select better than random.

Once STARTS identified the selected tests, EALRTS took an additional 0.01 seconds per test run. The time it took for STARTS to identify tests was 22.88 seconds. This study, based on one use case and insertion of mutants, suggests that machine learning is also time efficient to reduce tests selected.

RQ 4: How does the EALRTS compare STARTS in terms of selection rate, recall, change recall, and time to select tests?

Answer: EALRTS had a selection rate of 39.7% while maintaining a 95% recall rate. EALRTS also found at least one test failure in 99.97% of the test runs from STARTS. The results also showed that EALRTS could find patterns in data extracted from STARTS to perform significantly better than at random. Random Forest takes 0.01 seconds to select tests while XGBoost takes 0.002.

Chapter 6

Discussion

In order to use machine learning as a test selection tool, or PRTS, it requires different sources of information from selecting and running the tests affected by the code defects. Each source of information is a small indication of whether a test should be executed or not. The purpose of PRTS is to use an RTS tool and further reduce the number of test with machine learning. However, lacking code defects made by developers is a common problem for open source projects. In this thesis, EALRTS is presented, which uses multiple mutants to insert into a project instead of historical code defects. This thesis demonstrated that it is possible to learn from inserting multiple mutants into a maven-based java project. The learned model has shown promising predictive performance, but the result should be approached cautiously, which is further discussed in 6.1 and 6.2

6.1 Limitations

There are some limitations to EALRTS. The number of mutations inserted increases the chance of build failure, and for 15 mutants, the build success rate is 30%, and by 25 changes, the build success rate is 5%. This thesis does not explore what set of mutations that causes build failures or why it fails. EALRTS is also limited to the number of mutants possible to insert into a project.

Also, EALRTS does not control that a particular mutant introduces a test failure since testing all mutants for larger projects is computationally expensive,e.g., 37674 mutants for commons-math. The mutants were instead randomly selected, which resulted in compilation failures.

Further, the entire dataset used for training and evaluating the machine learning model consisted of 4277 different mutations over 725 test runs.

Previous research by Zhang et al.[12] used the entire mutation set generated by Pitest for nine projects of different sizes. They used nine different projects and generated mutants over multiple versions for each project. E.g., for commons-lang, they generated between 22,762 and 23,118 over seven different versions. However, they had access to a more powerful machine resource. Since generating the dataset with EALRTS is computationally expensive, the number of test runs was as many as could be gathered 4 hours a day over two weeks. Also, there is a possibility that more data can improve the results.

EALRTS never compared the predictive performance of faulty changes made by developers. Therefore, the interpretation of the results does not suggest that the selection rate can be reduced to 39.7% for changes made by developers. The result suggests that it is possible to reduce the number test with Random Forest and XGBoost when multiple mutants inserted into a project.

Also, this approach does not take flaky tests into account. Although possible, the machine resources used for this experiment were limited, and testing for flaky tests was no possibility in the given time frame. Machalica et al. [9] found that flaky testing can reduce the predictive performance of the machine learning model.

6.2 Threats to validity

Internal validity

EALRTS randomly selected what mutants to insert into the project. This technique made it hard to compile the project, and the average file cardinality was 5.9 for this experiment. The more mutants inserted, the lower the $buildSuccessRate$. In other words, there is a possibility that EALRTS is biased towards smaller changes, which is caused by the random insertion strategy that EALRTS uses.

External validity

Although the reduction of selection rate by 60.3% while maintaining a 95% recall seems good at first glance. The dataset averages 5.9 changes per test run; there is a risk that the model has overfitted for smaller changes. Cautions were taken to reduce overfitting, such as the usage of cross-validation and randomize a new number of mutants if a build fails, to make sure that particular combination of files has a higher chance of build success.

Also, the reduction in selection rate does not reflect changes made by developers. The evaluation was only made on the dataset gathered from the inserted mutants. Therefore, no conclusion can be made of how well the model performs outside the context of this work. Also, it was only conducted on one project since the data gathering process of EALRTS was computationally expensive.

Chapter 7

Conclusion

7.1 Summary

This thesis introduced EALRTS, a predictive regression test selection tool. EALRTS introduced a way to extract a set of features from STARTS when no historical code defects are available. EALRTS used mutation generation to create code defects, which were inserted into a maven-based java project. Features were then extracted from STARTS and resulted in a dataset. A machine learning model trained on the dataset and EALRTS used a Random Forest model to predict what tests to select. The results showed that Random Forst outperformed XGBoost.

This thesis evaluated EALRTS on one project, that is commons-math. The reason is that machine resources were limited and EALRTS is computationally expensive for generating the dataset. This resulted in EALRTS was only evaluated one project. The project that EALRTS was evaluated on resulted had a selection rate of 39.7% with a 95% recall and a 99.97% change recall.

7.2 Future work

The area of predictive regression test selection is an area with many possibilities. Many areas for future research were identified during this thesis. For the data generation process, it was found that more research can be made to increase the build success rate by having a better strategy than to insert mutants at random. Also, more research is required of why a particular set of mutant's cause builds to fail. Do certain mutant operations have a higher chance of build failure? Does it matter were in the project mutants are inserted? Also, it would be interesting to see if mixing non-faulty changes with mutants could make the model more robust and if it is beneficial to predicting errors made by developers.

The results shown suggested that a larger dataset increased the predictive performance of the machine learning models. The dataset was limited to how many test runs could be executed for two weeks. Therefore, it would be interesting to see a performance difference with a larger dataset or an even larger repository.

EALRTS did not take flaky tests into account. It would be interesting to see how this approach is affected by flaky tests. The number of flaky tests can be reduced by re-testing the failing tests. If a test fails for all the re-tests, it can be considered non-flaky. Then the learned model can be trained on the dataset without flaky-tests.

EALRTS were also never tested on real changes made by developers. It would be interesting to see how well it performs on actual changes made by developers. Then evaluate how well it performs on these changes.

It would also be interesting to investigate different or more features for this approach. Also, it would be interesting to see the benefits of different machine learning models. An evaluation of more sophisticated machine learning models that could yield a better result.

Bibliography

[1] Jesper Öqvist, Görel Hedin, and Boris Magnusson. “Extraction-based regression test selection”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM. 2016, p. 5. (Visited on 05/22/2019).

[2] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. “Ekstazi: Lightweight test selection”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 713–716. (Visited on 05/22/2019).

[3] Owolabi Legunsen, August Shi, and Darko Marinov. “STARTS: STAtic regression test selection”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 949–954. (Visited on 05/22/2019).

[4] Lingming Zhang. “Hybrid regression test selection”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 199–209. (Visited on 05/22/2019).

[5] Atif Memon et al. “Taming Google-scale continuous testing”. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press. 2017, pp. 233–242. (Visited on 05/22/2019).

[6] Swarnendu Biswas, Rajib Mall, and Manoranjan Satpathy. “A regression test selection technique for embedded software”. eng. In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.3 (2013), pp. 1–39. ISSN: 1558-3465. (Visited on 05/22/2019).

[7] Simone Romano et al. “SPIRITuS: a SimPle Information Retrieval regressIon Test Selection approach”. eng. In: *Information and Software Technology* 99 (2018), pp. 62–80. ISSN: 0950-5849. (Visited on 05/22/2019).

[8] J Andrews, L Briand, and Y Labiche. "Is mutation an appropriate tool for testing experiments?" eng. In: *Proceedings of the 27th international conference on software engineering*. ICSE '05. ACM, 2005, pp. 402–411. ISBN: 1581139632. (Visited on 05/22/2019).

[9] Mateusz Machalica et al. "Predictive Test Selection". In: *arXiv preprint arXiv:1810.05286* (2018). URL: `https://arxiv.org/pdf/1810.05286.pdf` (visited on 05/22/2019).

[10] Henry Coles et al. "PIT: a practical mutation testing tool for Java (demo)". eng. In: *Proceedings of the 25th International Symposium on software testing and analysis*. ISSTA 2016. ACM, 2016, pp. 449–452. ISBN: 9781450343909. (Visited on 05/22/2019).

[11] S.A. Irvine et al. "Jumble java byte code to measure the effectiveness of unit tests". In: *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*. 2007, pp. 169–175. ISBN: 0769529844. (Visited on 05/22/2019).

[12] Jie Zhang et al. "Predictive Mutation Testing". eng. In: *IEEE Transactions on Software Engineering* PP.99 (2018), pp. 1–1. ISSN: 0098-5589. (Visited on 05/22/2019).

[13] Mary Jean Harrold et al. "Regression test selection for Java software". In: *ACM Sigplan Notices*. Vol. 36. 11. ACM. 2001, pp. 312–326. (Visited on 05/22/2019).

[14] Owolabi Legunsen et al. "An extensive study of static regression test selection in modern software evolution". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 583–594. (Visited on 05/22/2019).

[15] A Rogier T Donders et al. "A gentle introduction to imputation of missing values". In: *Journal of clinical epidemiology* 59.10 (2006), pp. 1087–1091. (Visited on 05/22/2019).

[16] Ian Jolliffe. *Principal component analysis*. Springer, 2011. (Visited on 05/22/2019).

[17] I. Iguyon and A. Elisseeff. "An introduction to variable and feature selection". In: *Journal of Machine Learning Research* 3 (2003), pp. 1157–1182. ISSN: 15324435. (Visited on 05/22/2019).

[18] Salvador García, Julián Luengo, and Francisco Herrera. *Data Preprocessing in Data Mining*. eng. Vol. 72. Intelligent Systems Reference Library. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-10246-7. (Visited on 05/22/2019).

[19] Alexey Natekin and Alois Knoll. “Gradient boosting machines, a tutorial”. In: *Frontiers in neurorobotics* 7 (2013), p. 21. (Visited on 05/22/2019).

[20] Tin Kam Ho. “Random decision forests”. In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282. (Visited on 06/13/2019).

[21] Andy Liaw, Matthew Wiener, et al. “Classification and regression by randomForest”. In: *R news* 2.3 (2002), pp. 18–22. (Visited on 05/22/2019).

[22] David Muchlinski et al. “Comparing Random Forest with Logistic Regression for Predicting Class-Imbalanced Civil War Onset Data”. In: *Political Analysis* 24.1 (2016), pp. 87–103. ISSN: 1476-4989. (Visited on 05/22/2019).

[23] Corinna Cortes et al. “Sample selection bias correction theory”. In: *International conference on algorithmic learning theory*. Springer. 2008, pp. 38–53. (Visited on 05/22/2019).

[24] Tom Fawcett. “An introduction to ROC analysis”. In: *Pattern recognition letters* 27.8 (2006), pp. 861–874. (Visited on 05/22/2019).

Appendix A

Github repository

This is an open source project available on github.[1] The repository contains a Readme, modified STARTS, data generation program, the machine learning program and, the generated data.

Readme

The readme contains instructions of how the installation and usage of the program as well as pictures of how the program works. It also contains the results achieved in this thesis.

modified STARTS

The modified STARTS tool is named *starts-starts-1.3.zip* and installation and usage of the tool can be found in the readme.

data generation program

The data generation program can be found in the *eal_generateData* folder. This contains a pom.xml file in order to make it easier to build. Then in /src/-main/java there are four files. *ReplaceCompiledFiles.java* replaces then compiled mutants into the project and run STARTS. *RetrieveData.java* combines the data from the test result reports and the data that is extracted from STARTS at runtime.

AddChangeHistory.java and *change_file.sh* are used to add change history if a file history exists.

[1]https://github.com/kth-tcs/kth-test-selection/tree/master/eal%20predictive%20rts

machine learning program

The *Machine learning* folder consists of a *Preprocessing.py* file, which is used to preprocess the data. And *Training.py* which is used select features, train and evaluate the machine learning model. Instructions can be found in the readme.

A.1 Usage of data generation

The usage of the data collection process consists of three sections. Section A will describe how to generate the mutants; Section B will describe how to use the modified STARTS to store checksums of the original project; Section C will describe the implementation of data generation in EALRTS. A more extensive installation of the tool can be found at this thesis's Github.[2] Furthermore, this thesis investigated apache commons math.[3]

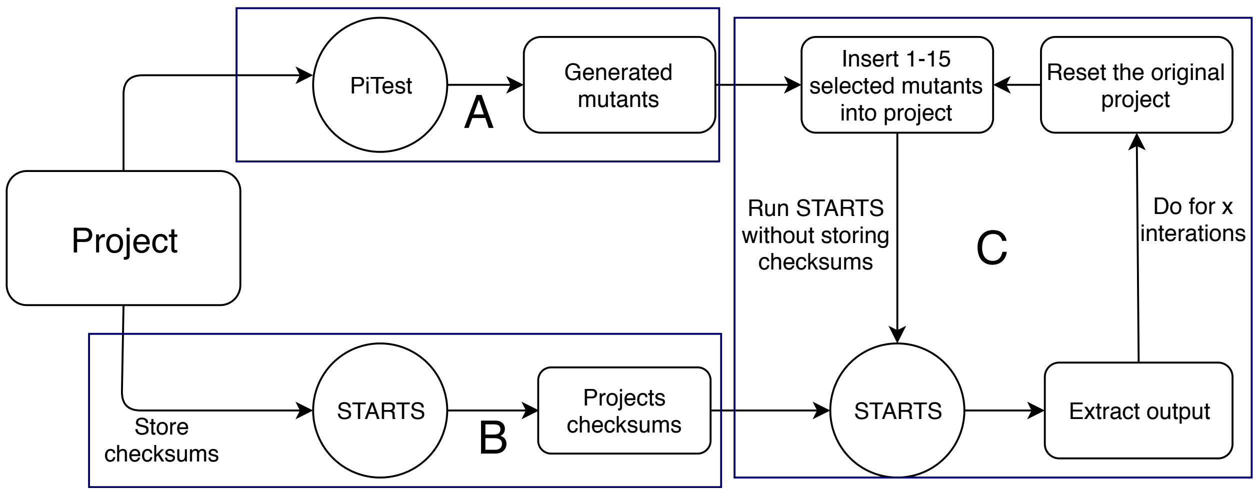

Figure A.1: Data extraction process of EALRTS divded into three sections, A, B, and C.

A. Implementation of PiTest

PiTest includes a maven plugin which is available from maven central and the installation can be found at the project's github[4].To generate mutants, the following command was executed in the directory of the project:

[2]https://github.com/kth-tcs/kth-test-selection/tree/master/eal%20predictive%20rts
[3]https://github.com/apache/commons-math
[4]https://github.com/kth-tcs/kth-test-selection/tree/master/eal%20predictive%20rts

```
$ mvn -Dfeatures=+EXPORT org.pitest:pitest-maven:
    mutationCoverage
```

The command will cause Pitest to generate mutants and persist the mutants to disk.

B. Implementation of STARTS

Install the modified starts from this thesis's Github and Run the following command in the project directory:

```
$ mvn starts:starts -DupdateRunChecksums=True
```

This command will cause starts to store the checksums for the given project.

C. Implementation of data generation in EALRTS

The following command was executed to run the data generation in EALRTS:

```
$ mvn exec:java -Dexec.mainClass=ReplaceCompiledFiles
```

This command will insert mutants into the project, run the modified STARTS version without storing the checksums, and finally extract the data from the starts and the test runs.

Once the process has finished, then run the "change_file.sh" script in the /src/main/java directory of the project:

```
$ Chmod +x change_file.sh

$ ./change_file.sh
```

This command will output a log.log file which contains change history for the files. The data can be combined with the data extracted from starts with the following command:

```
$ mvn exec:java -Dexec.mainClass=AddChangeHistory
```

TRITA -EECS-EX-2019:658