UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Natia Doliashvili

Predicting Survived and Killed Mutants

Master's Thesis (30 ECTS)

Supervisor(s): Dietmar Pfahl
Rudolf Ramler

Tartu 2019

Predicting Survived and Killed Mutants

Abstract:

Mutation Testing is a powerful technique for evaluating the quality of a test suite. During evaluation a large number of mutants is generated and executed against the test suite. The percentage of killed mutants indicates the strength of the test suite. The main idea behind this is to see if test cases are robust enough to detect mutated code. Mutation Testing is an extremely costly and time-consuming technique since each mutant needs to be executed against the test suite. For this reason, this paper investigates Predictive Mutation Testing (PMT) technique to make Mutation Testing more efficient. PMT constructs a classification model based on the features related to the mutated code and the test suite and uses the model to predict execution results of a mutant without actually executing it. The model predicts if a mutant will be killed or it will survive. This approach has been evaluated on several projects. Two Java projects were used to assess PMT under two application scenarios: cross-project and cross-version. C project was also used to explore if PMT can be applied to a different technology. PMT has been evaluated using only one version of a C project. The experimental results demonstrate that PMT is able to predict execution results of mutants with high accuracy. On Java projects it achieves above 0.90 ROC-AUC values and less than 10% Prediction Error values. On the C project it achieves above 0.90 ROC-AUC value and less than 1% Prediction Error value. Overall, PMT is shown to perform well on different technologies and be robust when dealing with imbalanced data.

Keywords:

Software testing, mutation testing, predictive mutation testing, machine learning

CERCS: P170: Computer science, numerical analysis, systems, control

Ellujäänud ja tapetud mutantide ennustamine

Lühikokkuvõte:

Mutatsioonitestimine on tarkvaratestimises kasutatav meetod hindamaks testikomplekti kvaliteeti. Hindamise ajal genereeritakse programmi lähtekoodist suur hulk mutante ja jooksutatakse nende peal testikomplekti. Tapetud mutantide osakaal kõigist mutantidest näitab testikomplekti headust. Eesmärk on mõista, kas testid suudavad leida muteerunud koodi, andes sellega infot testide kvaliteedi kohta. Mutatsioonitestimine on äärmiselt kulukas ja aeganõudev meetod, kuna kõikidel mutantidel peab ükshaaval jooksutama terve testikomplekti. Käesolevas töös uuritakse ennustavat mutatsioonitestimise meetodit, mille toel tõhustada mutatsioonitestimise protsessi. PMT treenib klassifitseerimismudeli, kasutades selleks muteeritud koodil ja testikomplektil põhinevaid tunnuseid. Treenitud mudeliga ennustatakse, kas mutant tapetakse või jääb ellu, mutanti ennast testikomplekti vastu jooksutamata.

Antud lähenemist katsetati mitme tarkvaraprojekti peal. Kaht Java keelel põhinevat projekti kasutati katsetamaks ennustavat mutatsioonitestimist kahes erinevas olukorras: üle mitme projekti ja üle mitme versiooni. C-keelel põhinevat tarkvaraprojekti kasutati uurimaks, kas ennustavat mutatsioonitestimist saab rakendada ka teistel tehnoloogiatel põhinevatel projektidel. Katsetulemused näitavad, et ennustav mutatsioonitestimine suudab ennustada mutantide ellujäämist või tapmist kõrge täpsusega. Java projektidel

saadi tulemuseks üle 0.90 ROC-AUC väärtused ja väiksemad kui 10% ennustusvea väärtused. C projektil saadi tulemuseks üle 0.90 ROC-AUC väärtus ja väiksema kui 1% ennustusvea väärtuse. Üldiselt on näidatud, et ennustav mutatsioonitestimine töötab hästi erinevatel tehnoloogiatel ja tuleb toime ka andmetes esinevate ebavõrdsete klasside suurustega.

Võtmesõnad:

Tarkvaratestimine, mutatsioonitestimine, ennustav mutatsioonitestimine, masinõpe

CERCS: P170: Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction 5
1.1 Motivation 5
1.2 Research Questions 6
2 Background and Related Work 6
2.1 Mutation Testing 6
2.2 Practical application: safety-critical systems 7
2.3 Predictive mutation testing 8
3 Methodology 9
3.1 Feature selection 9
3.1.1 Execution features 10
3.1.2 Infection features 10
3.1.3 Propagation features 10
3.2 Machine learning algorithm 13
3.3 Data balancing methods 14
3.4 Implementation 14
3.5 Used projects 15
3.6 Evaluation of the model 17
4 Results 18
4.1 Results for RQ1: Replication (Java) 18
4.1.1 Cross-project 18
4.1.2 Cross-version 19
4.1.2.1 Replication results 19
4.1.2.2 Fine tuning classifier parameters 26
4.1.2.3 Effect of removing mutants with no coverage 27
4.1.2.4 Balancing the data 28
4.1.2.5 Location as a feature 29
4.2 Results for RQ2: Transfer PMT from Java to C 32
4.2.1 Common features of C and Java projects 32
4.2.2 Performances of C and Java projects 35
4.2.3 Feature selection of C project 37
5 Discussions 38
6 Conclusions 39
7 Acknowledgments 39
References 40

1 Introduction

1.1 Motivation

Mutation Testing is a powerful technique for evaluation of the test suite quality [1]. A mutant is a variant of the source code where change has been made to a part of the code, for example, a certain statement has been altered (mutated). There are several different types of mutants that can be generated, but all the changes are very small not to affect the overall program. Basically, by creating a small fault in the initial program we generate mutants. Each mutant has only one fault. A number of mutants are generated in this way from the initial code and executed against the test suite. The goal of the Mutation Testing is to assess the quality of the test suite. We expect the test suite to be good enough to detect the change to the program by failing at least one of the test case. The execution results of the original program and mutated program are compared to each other. If the results are the same this means that the mutant has survived, otherwise we say that mutant has been killed. In order to evaluate the quality of the test suite mutation score can be used. Mutation score is defined as a number of killed mutants divided by a number of all mutants. Higher the mutation score is the better is the quality of the test suite.

Although mutation testing is a very useful software testing method, at the same time it is highly expensive technique. It requires generation of a large number of mutants and execution of each mutant against the test suite. There are many ways to reduce the cost of mutant generation, but despite the effort to reduce the cost of mutant execution part, it remains costly.

To lessen the cost of mutant execution a new method was introduced [2, 3] which obtains execution results without actual execution of the mutants. Predictive Mutation Testing (PMT) is the first method that uses machine learning to predict the outcomes of mutant execution. PMT collects easy-to-access features of the mutants that have already been executed against the test suite. These mutants can be either from an earlier version of the same project or from a different project. Machine learning predictive model is trained using the information of the mutants: the features collected for each of the mutants and their execution results. Trained model is able to predict, without actually executing, the outcome of the mutants of newer versions of the same project or mutants of the different project.

PMT was evaluated under three application scenarios: cross-version, cross-project and using only one version of the project. Cross-version and cross-project scenario was tested against Java projects, whereas single version scenario was tested on Java and C projects.

For the evaluation of effectiveness of PMT method several evaluation measures are used: accuracy, precision, recall, F-measure and AUC-ROC curve. Together they indicate how well the model is able to predict the results of mutant execution. Aside from above mentioned evaluation measures, there is one other measure that can be used. Mutation score is defined as a ratio of the killed mutants to all the mutants. It is used to evaluate test suite quality. We can also use it to evaluate how far model predictions are from the true results. More precisely, new measure - Prediction Error is defined as the difference between predicted (obtained from the model) and true mutation scores. Prediction Error can be used for evaluation of the effectiveness of PMT.

The experiments show that PMT performs very well under all three scenarios: cross-versions, cross-project and single-version. PMT achieves above 0.90 ROC- AUC values for most of the cases. It achieves best performance under cross-version scenario. All Prediction Error values are below 10%. PMT improves efficiency of mutation testing for Java and C projects.
This paper investigates findings of existing paper [2] and tries to investigate new ways to further improve PMT.

1.2 Research Questions

This paper answers two research questions:

- RQ1: Is it possible to replicate the results of the paper "Predictive Mutation Testing" [2] using the same projects but different tools under the cross-version and cross-project scenarios? This research question investigates how authentic are the results of the original paper.
- RQ2: Is it possible to transfer the predictive mutation testing method to a different technology? More precisely, this research question investigates if PMT can be applied to a C project instead of Java project as it was done in the original paper. To answer this question, we should answer the following questions:
    - RQ2.1: Are the available features comparable in Java and C projects?
    - RQ2.2: Are the results between Java and C projects comparable?

2 Background and Related Work

2.1 Mutation Testing

Mutation analysis was initiated in the 1970s and has a long history of advancements. It is a process of generated program variants that are called mutants and that have mutated part of the code. The aim is to introduce artificial defects in the code. Mutation testing uses mutation analysis to help the testing process by evaluating the strength of the test suite [4]. When a test can differentiate the behavior of a mutant from a behavior of the original program we say that this mutant is 'killed', in another case we say that this mutant is 'survived'. Usually, tests examine the output of the program, for example, things that program prints or results of assertions. Overall, in order for a mutant to be killed, it must cause a program state to be changed. This causes the problem of *equivalent mutants*: the mutants semantically equivalent to the original program are called equivalent mutants and they can never be killed. The mutated program is behaviorally equivalent to the original one. The detection of equivalent mutants is one of the main problems of using mutation testing.

Mutants are the altered version of the original code. Thus, there exist transformation rules called 'mutant operators'. These rules dictate how the changes should be made in the program. For example, the conditionals boundary mutator replaces the relational operators <, <=, >, >= with their boundary counterpart.
In mutation testing systems mutant operators are largely chosen to be not too easy to detect and minimize generations of equivalent mutants. For example, PIT [5] is a state of the art mutation testing system and it uses 7 mutant operators by default:

1. *Conditionals Boundary* (replaces the relational operators with their boundary counterpart)
2. *Increments* (replaces increments with decrements and vice versa)
3. *Invert Negatives* (inverts negation of integer and floating point numbers)
4. *Math* (replaces binary arithmetic operations with another operation)
5. *Negate Conditionals* (mutates all conditionals found)
6. *Return Values* (mutates the return values of method)
7. *Void Method Calls* (removes method calls to void methods)

Based on the language of the program different sets of mutant operators can be chosen. After choosing mutant operators they are used to generate mutants for the analysis. The aim of mutation testing is to detect weak parts of the test suite and improve them. To evaluate test suite strength, it is necessary to have some kind of evaluation measure. Mutation score is defined as the ratio of killed mutants to all the mutants. This score is a good measure in case all mutants have equal value but this is not always the case. There exist types of mutants that boost the mutation score excessively high and makes it hard to interpret. One of such mutant is equivalent mutant which is described above. Another such mutant is *redundant mutant.* Redundant mutants exist in forms of *duplicated* and *subsumed mutants*. Duplicated mutants are equivalent to each other but not with original program. Whereas subsumed mutants are the mutants that are jointly killed when other mutants are killed. If we remove this kind of mutants this does not affects test generation process but it will affect mutation score. Since identifying equivalent and redundant mutants is not an easy task, this makes it hard to evaluate the test suite quality based on mutation score.

2.2 Practical application: safety-critical systems

Safety-critical systems are the systems whose failure can cause loss of life, significant property damage, or damage to the environment. They must follow safety standards. Safety standards provide recommendations for the creation of a system that achieves a defined safety integrity level (SIL).

Testing plays a major role in the verification and validation of safety-critical system development. There are many recommendations by safety standards about testing approaches but they do not provide instructions on how these approaches should be applied in practice.

There is very little research done using the mutation testing technique for safety-critical systems [6]. An example of such a paper [7] is done by Ramler, Wetzlmaier and Klammer. They investigate the applicability and usefulness of using mutation testing technique to help increase the quality of a test suite for safety-critical software systems. Mutation analysis has been applied to the system and 75,043 mutants were generated out of which 27,158 survived test execution. 200 live mutants have been further studied manually by engineers and based on their discoveries existing test suite was improved. Engineers found that those 200 mutants contained 24% equivalent mutants and 12% duplicated mutants. They also found a weak spot in the testing approach and improved the test suite. While improving test suite two new faults were discovered in the code.

Their findings show that mutation testing is a useful technique for measuring test suite quality. In addition, it can be used to determine faults in the test cases that are hard to discover otherwise.

2.3 Predictive mutation testing

Mutation testing is known as computationally expensive technique. Predictive mutation testing is a new predictive method to help reduce the cost of executions of mutation testing [2]. This is the first approach that predicts the results of mutant execution without executing mutants against the test suite. More precisely, this approach trains a classification model using a set of features related to mutants and tests. Once the model is trained it is used to predict the results of mutants (killed or survived) without their executions. In the paper, the authors evaluated PMT on 163 real-world projects. The results show that PMT offers the reduction of execution cost in exchange for small accuracy loss.

Two application scenarios were used for evaluation: *cross-version* and *cross-project*. Evaluation measures used to assess the predictive ability of the model are precision, recall, F-measure, and AUC. Furthermore, prediction error was calculated as a difference between predicted and real mutation score. The results show that PMT is very effective under the cross-version scenario. It achieves over 0.90 precision, recall, F-measure and AUC. Whereas under cross-project scenario it achieves over 0.85 AUC and lower than 15% error on predicting mutation scores. Moreover, PMT was shown to be more efficient than traditional mutation testing.

In the paper, they select features based on PIE theory [8] using three categories of the features: execution, infection and propagation. Execution features are related to the mutated statement being executed by tests. Infections features are related to the program state being affected by the execution of the mutated statement. Lastly, propagation features are related to the infected program state being propagated so that it affects the output of the program and makes it distinct from the original program output.

Random Forest was the choice for the classification model. Besides Random Forest they also used Naïve Bayers, SVM and J48 to see if they performed better than Random Forest.

Two different strategies of balancing the data were applied to check if they can improve the performance of the PMT: cost-sensitive and under-sampling.

As for the implementation they used several tools. PIT and Major tools were chosen for mutation testing since Major is widely used and PIT is efficient. Cobertura was chosen to collect coverage related features. The infection features were collected straight from the mutation testing tools. For extraction of propagation features, they developed their own tools. Finally, for the machine learning part they used Weka machine learning library.

For the evaluation of PMT 9 base java projects were used and evaluation was extended on another 154 projects. All of them passing their Junit tests.

In the end, authors were able to demonstrate that PMT makes mutation testing more efficient in the exchange of small accuracy loss. The experimental results of 163 real-world Java projects reinforce this statement.

The extended version of this paper [3] also includes the results of investigations of the contribution of 14 individual features, comparison of different categories of features, predictability of the mutants under PMT and ways to further improve the effectiveness of PMT.

3 Methodology

Mutation testing is believed to be an expensive technique to use. To lessen the cost of executions of the mutants PMT was introduced. PMT predicts mutation testing results without executions of the mutants itself. Execution results can be either killed or survived. Since there are only two possible options of mutant execution, in machine learning their classification can be seen as a binary classification problem.
PMT treats this problem as a binary classification and builds a machine learning model based on some easy-to-access features and the results of mutant executions. First is it necessary to execute mutants against the test suite and save their outcome (killed or survived). Besides, some easy-to-access features should be collected for each mutant. PMT uses these features and execution results to build a classification model. Once new mutants are generated the classification model can predict whether a mutant will be killed or survived based on the same easy-to-access features as used during the training of the model. Note that features should be easy to collect to be able to obtain them quickly.
In the process of building a machine learning model, there are two main phases. The first phase is to determine which features should be collected for a mutant (Section 3.1). These features should be related to the execution result. Furthermore, features should be easy to access because of the efficiency reason. The second phase determines which machine learning algorithm to be used to build a prediction model (Section 3.2). A chosen algorithm should be able to learn from training data and make predictions with high accuracy. On top of these two phases, an imbalanced data issue was also investigated (Section 3.3).

3.1 Feature selection

Feature selection is done using PIE analysis technique: propagation, infection and execution analysis. This technique is related to mutation testing and estimates program characteristics that can affect the program's computation. The idea behind the PIE analysis is not to detect if there is a fault in the program, but instead, it identifies locations in the program where faults are likely to stay undetected by the test suite. PIE analysis estimates three program characteristics that can affect the behavior of the program, therefore they can be used as the conditions that need to be satisfied for a mutant to be killed. The first condition is execution: a mutated statement needs to be executed by the test. The second condition is infection: execution of the mutated statement affects the program state and therefore mutant is identified. The third condition is propagation: infected program state returns output that is distinct from the original program output. The values of above-mentioned conditions can be obtained for each mutant. According to those values, we can predict the results of mutant execution. These three conditions give us information about a mutant and its outcome. They are the features that describe each mutant and therefore can be used for the result prediction. So in the end, we have three different categories of features. The following subsections describe each feature category in detail.

3.1.1 Execution features

The execution feature category consists of features that are related to the execution of a mutated statement. These features should describe if a mutated part of the program was executed.

In Java language projects two such features can be found:

- *numExecuteCovered*
- *numTestCovered*

numExecuteCovered indicated how many times a mutated line of the program is executed by the test suite. As for *numTestCovered* it indicates how many tests from the test suite reach a mutated method of the program. To identify these values first, an original program must be executed against the whole test suite and record how often each statement is executed and how many tests execute it. Based on the data collected we can calculate values of two above features for each mutant.

In C language project only one such feature can be found:

- *numTestCovered*

numTestCovered indicates how many tests from test suite cover a mutated method of the program.

3.1.2 Infection features

The infection feature category identifies features corresponding to the infection that the mutated statement is causing in the program state. Changes in the program state depend on changes made to the mutated statement, consequently, we need a feature that describes the type of mutated statement before it was mutated and how was it modified.

In Java language projects following two features can be found:

- *typeStatement*
- *typeOperator*

The feature *typeStatement* indicates what type of statement was mutated. For example, it can be a conditional statement or return statement. The second feature *typeOperator* indicates what kind of mutation was done on the statement.

Apart from the above features in C language project, one additional feature can be obtained

- *Mutation*

Mutation indicates actual replacement for the mutated source code element.

3.1.3 Propagation features

The final category, propagation features, contains features that are related to the propagation of infected program state. This category investigates characteristics that are related to the complexity of the program. If a program is complicated, then there

is a high possibility that the program state produced by the mutated statement alters program output.

In Java language projects following features can be obtained to detect how much infected program state can spread and affect program output:

- *McCabe Cyclomatic Complexity*
- *Method Lines of Code*
- *Nested Block Depth*
- *Depth of Inheritance Tree*
- *Number of Children*
- *Afferent Coupling*
- *Efferent Coupling*
- *Instability*

McCabe Cyclomatic Complexity indicates exactly what it says: McCabe complexity of the mutated statement. *Method Lines of Code* feature describes the number of lines of code in the mutated method. *Nested Block Depth* refers to the depth of nested blocks in the mutated method. *Depth of Inheritance Tree* shows what is the length from mutated class to the root class. *Number of Children* describes how many subclasses the mutated class has. *Afferent Coupling (Ca)* indicates how many classes outside of the mutated package depend on classes inside the package. *Efferent Coupling (Ce)* indicates how many classes inside the mutated package depend on classes outside the package. Lastly, *Instability* is calculated using the previous two features: Ce/(Ce+Ca).

In C language project we have different features:

- *McCabe Cyclomatic Complexity*
- *Branches*
- *Loops*
- *Maintainability*
- *Sloc*
- *Lines*
- *Operands*
- *Operators*
- *Unique_operands*
- *Unique_operators*
- *Volume*

McCabe Cyclomatic Complexity is the same as above. *Branches* describe the number of branches, for example, the number of if statements in the tested function. *Loops* indicates the number of loops, for example, the number of for statements in the tested function. *Maintainability* refers to the maintainability index. *Sloc* indicates the number of source code lines without blank lines. *Lines* indicates the number of source code lines with blank lines. *Operands* indicates the number of total operands. *Operators* indicates the number of total operators. *Unique_Operands* is the number of distinct operands. *Unique_Operators* is the number of distinct operators. *Volume* describes Halstead complexity.

All the above features are related to the complexity of the program to see how much can the infected state will spread. In addition to this, several other features can be identified that will help to predict if a mutant will be killed or survived. The features

that are related to the outcome of the program and show if the test suite has the ability to detect differences between a mutant and original program output. In some cases, if the program does not return anything and there are no other ways of checking if the program works as expected we cannot detect if the execution result of a mutant is different from that of the original program execution result.
For this reason, we might consider using the following additional features.
In java language test assertions can be obtained as well as type of return values. The list of features related to checking the program execution result is following:

- *numMutantAssertion*
- *numClassAssertion*
- *typeReturn*

numMutantAssertion indicates the number of assertions in the test methods that cover each method. *numClassAssertion* indicates the number of assertions in the test class that covers the mutated class. *typeReturn* is the return type of the mutated method.

In C language there is a return type but there are no assertions. Instead of assertions several other features can be used:

- *numMutationAssertions_iparam*
- *numMutationAssertions_oparam*
- *numClassAssertions*

numMutationAssertions_iparam is a substitute measure using a number of in parameters asserted by the test alongside with *numMutationAssertions_oparam* feature which is a substitute measure using a number of out parameters asserted by the test. These two features substitute *numMutationAssertion* feature. *numClassAssertions* is also substitute measure using all asserted parameters for all tested functions in the same C file.

The feature list of C project that differs from the Java project feature is the following:

- *Mutation*: an actual replacement of the mutated source code element
- *cfileId*: tested C file (70 unique value)
- *methodId*: tested function/method in the C file (45055 methods)
- *Line*: line location of the mutated source code element
- *Column*: column location of the mutated source code element
- *branches*: number of branches, e.g. if statements, in the tested function (source code metric)
- *loops*: number of loops, e.g. for statements, in the tested function (source code metric)
- *maintainability*: maintainability index (source code metric, maintainability index calculates an index value between 0 and 100 that represents the relative ease of maintaining the code)
- *operands*: number of total operands (source code metric, used to compute Halstead volume)
- *operators*: number of total operators (source code metric, used to compute Halstead volume)
- *unique_operands*: number of distinct operands (source code metric, used to compute Halstead volume)

- *unique_operators*: number of distinct operators (source code metric, used to compute Halstead volume)
- *volume*: Halstead complexity (source code metric)

For the rest of the features the names are different from Java project features. Below is the description of the common features:

- *typeOfMutant*: types of mutant operators. (typeOperator in Java)
- *numTestCovered*: number of tests covering the mutated line (numTestCovered in Java)
- *numMutationAssertions_iparam*: number of assertions by testMethod (substitute measure using number of in parameters asserted by the test)
- *numMutationAssertions_oparam:* Number of assertions by testMethod (substitute measure using number of out parameters asserted by the test)
  Note that 2 above feature are replacement of numMutantAssertion in Java.
- *numClassAssertions*: number of assertions in the whole test class (substitute measure using all asserted parameters for all tested functions in the same C file) (numClassAssertion in Java)
- *typeReturn*: return types. (typeReturn in Java)
- *mccabe*: Cyclomatic complexity (source code metric, used to indicate the complexity of a program) (McCabe Cyclomatic Complexity in Java)
- *sloc*: number of source code lines without blank lines (source code metric)
- *lines*: number of source code lines with blank lines (source code metric)
  Above two feature are similar to Method Lines of Code in Java

3.2 Machine learning algorithm

Machine learning is an important part of PMT. The classification model is trained based on the training data and used afterwards for predictions. The trained model needs to predict if a mutant will be killed or survived, so we have only to classes of target values. Therefore, this is a binary classification problem. Training data consists of the mutants which already have been executed against the test suite and hence we have the results of executions for all the mutants. The model makes predictions on new mutants and classifies them as either killed or survived. There are a variety of classification methods but in this paper only *Random Forest* and *Support Vector Machine (SVM)* methods were used.

Decision Trees are building blocks of Random Forest. The Decision Tree classifier chooses a feature at a time that splits the instances into two groups where instances of different groups are as distinct from each other as possible and instances in the same groups are similar. Random Forest creates a large number of decision trees based on a randomly selected subset of the training set. In order to make a prediction on a test instance, it aggregates votes of each individual decision tree and predicts the class with the most votes. It is important that decision trees have low correlations between them. The reason for this is that while some trees make wrong predictions many other trees make correct predictions. Hence, as a group, they will be able to make the correct prediction.

SVM, on the other hand, uses a different approach. The main idea is to find a hyperplane in an n-dimensional space that separates data instances (where n is the number of features). There are many possible hyperplanes that can separate two classes data instances. SVM chooses the one with the maximum margin. In other

words, the one with the maximum distance between the instances of different classes.

3.3 Data balancing methods

In mutation testing the number of killed and survived mutants are usually different from each other hence we have unbalanced data issue. Mostly there are more killed mutants then survived which is an uneven distribution of classes. There are several methods in machine learning to deal with unbalanced data.
One of the strategies is under-sampling. Under-sampling has several different methods. One of the simple methods is random under-sampling for the majority class. This method removes instances of majority class randomly and uniformly. It can lead to information loss if essential instances are being removed but if instances of majority class are near to each other this method has a good result.
Another good strategy for unbalancing data is cost-sensitive learning. Cost-sensitive learning assigns a higher cost to the misclassification of minority class and therefore minimizes total cost. For example, Random Forest is designed to minimize overall error rate, therefore it focuses on maximizing the accuracy of majority class predictions and results in lower accuracy predictions for the minority class. Weighted random forest assigns weights to both classes but minority class has larger weight. This means the misclassification cost is higher.

3.4 Implementation

For mutation testing PIT tool was used. PIT is state of the art mutation testing system for Java. More precisely, PIT runs unit tests on automatically modified versions of the code. Modified code has some faults introduced in it so the result should be different from the original program result and this should cause unit tests to fail. If none of the tests fail, then the test suite needs improvement. Justification for the choice of PIT tool is that it is fast, easy to use and results are easy to interpret.

Mutation data was collected from PIT tool. The report contains the location of the mutant and the execution results of each mutant. Execution results are 4 different types: *killed, lived, no coverage* and *timed out*. A mutant is killed if at least one test will fail during the execution of this mutant. A mutant is called lived if none of the test fails during its execution. No coverage means none of the tests exercised the line of the mutated code hence none of the tests fail during its execution. Time out means that mutated code causes an infinite loop, for example, removing the increment from a counter in a for loop can cause a time out. PIT reports also used mutation operators for each mutant which can be used as a typeOperator feature.

For the collection of features, several tools were used and gathered data was combined. numExecuteCovered feature was extracted using OpenClover coverage tool. OpenClover is a tool for measuring code coverage for Java projects. It collects metrics of the code to detect the most untested areas of the application as well as find the riskiest parts of code.

To extract propagation features using metrics plugins is the easiest way in Eclipse. This plugin calculates several different metrics for the code during the build cycle. Following metrics data can be used to obtain the values of the features:

- *Number of Children* (numChildren)
- *Depth of Inheritance Tree* (depInheritance)
- *Nested Block Depth* (depNestblock)

- *Method Lines of Code* (LOC)
- *McCabe Cyclomatic Complexity (*infoComplexity)
- *Afferent Coupling* (Ca)
- *Efferent Coupling* (Ce)
- *Instability* (instability)

Note, Method Lines of Code is a total number of lines of code inside method bodies, excluding blank lines and comments.
The feature typeReturn is also included in OpenClover report.

Test coverage data was extracted using PIT tool again. It has one parameter called exportLineCoverage which if indicated “true” will export line coverage data. The exported file contains the list of tests that cover each method. From this file feature numTestCovered was calculated. Since this file contains information for the methods and not for lines, numTestCovered was calculated as the number of tests that cover the mutated method.
Finally, assertion features were simply collected by analyzing code.

All the machine learning part of the project was done using python 3. Training the classification models and evaluation of trained models.

Feature collection on C project was done by Software Competence Center Hagenberg (SCCH). Collected data was given to me. Mutants were generated using Milu tool.

3.5 Used projects

Since this is a replication of the existing paper [2] the same projects were used for the evaluation of PMT. Out of 9 base projects mentioned in the paper 2 were used to replicate the results of the paper: Java apns (apns) and Linear Algebra for Java (la4j). Java apns is a Java client for Apple Push Notification service. This library aims to provide a highly scalable interface to the Apple server. la4j is an open-source Java library. It provides Linear Algebra primitives (matrices and vectors) and algorithms.
For each of the projects, the same commits were used from their github pages that were used in the paper. apns and la4j projects were used for the cross-version and cross-project scenarios and different versions of the project were gathered. Each version is at 30 commits distance from each other. Information about all versions of the projects are presented in **Table 1** and **Table 2**. "C." letter at the beginning of the column name means that these values were extracted from the OpenClover report. Likewise, "P." means that this value was extracted from the paper [2]. Columns “P.SLOC” and “C.NCLOC” are exactly the same as it is expected because they present the numbers of lines of executable code of the same project. Column "Test run" was extracted from eclipse tool and shows the number of tests of each version. “P.Test” is the same as “Test run” but extracted from OpenClover tool. “Test run” and “P.Test” differ in some cases for apns project which should not be the case. The reason behind this is that one test was removed from the test suite because it was failing on the original program and prevent PIT from executing. Column “default Mutants” shows the number of mutants generated by PIT with the mutation operator parameter indicated to default. Column “All Mutants” shows the same number but mutation operator parameter indicated to all. Column “Killed Mutants” is the number of killed mutants of “All Mutants” and “Distribution” shows the number of “Killed mutants” divided by the number of “All Mutants”. **Table 3** and **Table 4** show the changes between each versions of projects.

Version	C. NCLOC	P. SLOC	Test run	P. Test	default Mutants	All Mutants	Killed Mutants	Distribution
v0	666	666	64	65	143	526	374	0.71
v1	859	859	64	65	233	789	463	0.59
v2	1221	1221	67	66	338	1107	534	0.48
v3	1221	1221	67	67	338	1107	529	0.48
v4	1288	1288	74	75	365	1162	590	0.51
v5	1503	1503	84	87	416	1398	809	0.58

Table 1 Information of apns project

Version	C. NCLOC	P. SLOC	Test run	P. Test	All Mutants	Killed Mutants	Distribution
v0	5810	5810	245	245	8846	4025	0.46
v1	6804	6804	353	353	9862	4581	0.46
v2	7074	7074	396	396	10248	4795	0.47
v3	7264	7264	463	463	10705	5261	0.49
v4	8202	8202	581	581	11531	7006	0.61
v5	8035	8035	621	621	11646	7083	0.61
v6	7086	7086	625	625	10870	7080	0.65

Table 2 information of la4j project

versions	changed files	addition	deletion	changes
v0-v1	25	785	217	193
v1-v2	34	1.321	232	362
v2-v3	3	37	19	0
v3-v4	21	533	214	67
v4-v5	28	1363	344	215

Table 3 Changes between two successive commits of apns project

versions	changed files
v0-v1	994
v1-v2	270
v2-v3	190
v3-v4	938
v4-v5	167
v5-v6	949

Table 4 Changes between two successive commits of la4j project

C project is the software of a safety-critical industrial system from the existing paper [5]. The embedded software controls the electrical and mechanical components of the overall mechatronic system. The embedded software system consists of a real-time operating system, platform-specific libraries and an application structured in 30

domain-specific components. The whole system is written in the C programming language. The application has about 60,000 LOC (lines of code). Component sizes range between 400 and 7,000 LOC. Mutation testing produced 75,043 mutants of which 27,158 passed test execution.

3.6 Evaluation of the model

The effectiveness of the classification models is evaluated using the following evaluation metrics*: Accuracy, Precision, Recall, F-measure, AUC* and *Confusion Matrix*. Beside, PMT can also be used to predict the mutation score of the project based on the ration of the mutants predicted as killed to all the available mutants. *Prediction error* calculates the difference between true mutation score and mutation score calculated using predicted results of mutants.

Accuracy is the ratio of a number of correct predictions to the total number of test set instances. This is a good measure only if there are an equal number of instances presented from each class.

Precision is the number of true positive instances (instances that were correctly predicted as positive) divided by the number of positive instances predicted by the classifier.

Recall is the number of true positive instances divided by the number of instances that should have been identified as positive.

F-measure F1 Score is the Harmonic Mean between precision and recall. It tries to find a balance between precision and recall. It shows how precise and robust is the classifier. The higher value means the better performance of the model.

Area Under Curve(AUC) is one of the most widely used metrics for evaluation. This measure is usually used for binary classification problems. AUC is the area under the curve of plot False Positive Rate vs True Positive Rate at different points in [0, 1]. True Positive Rate is the number of positive data points that are correctly classified as positive, divided by the number of all positive data points. False Positive Rate is the number of negative data points that are incorrectly classified as positive, divided by the number of all negative data points. The higher value means the better performance of the model.

Confusion Matrix as the name indicates generates a matrix as output and describes the complete performance of the model. It is a base for other metrics. Calculates true positive, false positive, true negative and false negative values of predictions.

Prediction Error is the difference between the mutation score calculated on test data and the mutation score calculated on predicted results of mutants of test data. the mutation score is the ration of the mutants predicted as positive (killed) to the number of all available mutants.

4 Results

This section presents the results of the research questions. 4.1 section answers the first research question: if it is possible to reproduce the results of the existing paper [2]. The performances under two application scenarios are displayed: cross-version and cross-project. In addition to replication results, several new ideas were investigated under the cross-version scenario. 4.2 section answers the second research question: if it is possible to transfer PMT to C language project. The performance of C project is presented under a single-version scenario. First, the performance of C and Java projects using the same set of features are displayed (section 4.2.1). Second, the results of both projects with all their available features are presented (section 4.2.2). Furthermore, several features of C project were chosen to investigate how good results can be received using only those features.

4.1 Results for RQ1: Replication (Java)

To answer the RQ1 this section presents the performances of two Java projects (apns and la4j) under cross-project and cross-version scenarios. In addition to replication results several new ideas were investigated under cross-version scenario:

- Fine-tuning classification model parameters
- Effects of removing mutants with no coverage
- Applying two balancing technique to imbalance data (after removing mutants with no coverage)
- Addition of location feature.

4.1.1 Cross-project

Cross-project scenario uses latest versions of each project and trains classification model using mutants from one of the project and evaluates it on another projects. In the paper [2] they used 9 base projects. They use mutants from one of the project as a test and mutants from all the remaining projects as a train set. Since we have only two java projects, cross-project evaluations were done using latest versions of each project. Classification model was built on one of the project and evaluated on another project. For example, apns project was used to build the model and la4j project was used to test it and vice versa. The results are presented in **Table 5** for two configurations of Random Forest: with default parameters and after fine tuning of some parameters. The same experiment results from the paper [2] are displayed in **Table 6** for comparison. Their results are produced using Random Forest with default parameters.

parameter fine-tuning	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
no	apns-la4j	0.691	0.866	0.515	0.646	0.858	22.1
yes	apns-la4j	0.756	0.848	0.675	0.751	0.88	11.2
no	la4j-apns	0.873	0.846	0.954	0.897	0.919	7.4
yes	la4j-apns	0.888	0.846	0.986	0.911	0.923	9.6

Table 5 The results of Random Forest under cross-project scenario

Sub.	Prec.	Recall	F.	AUC	Err.
lafj	0.888	0.876	0.869	0.876	10.72%
apns	0.897	0.884	0.884	0.935	8.72%

Table 6 The results of Random Forest under cross-version scenario from the paper [2]

As we can see in the above tables some metrics values are very close to each other even though different train sets were used for building the models. Apns project evaluation metrics (3rd and 4th row in **Table 5**) are more close to the paper [2] results (2nd row in **Table 6**). This might be the result of training data size since la4j has a larger number of mutants than apns project. Therefore, the model had more training data and was able to learn better. However, la4j project results are far from the paper [2] results. This outcome is expected because the model was trained using apns project which has a small number of mutants. It is obvious that the model was not able to be as good.

4.1.2 Cross-version

4.1.2.1 Replication results

The cross-version scenario uses already executed mutants of earlier versions of the program and collects easy-to-access features to build a classification model. PMT uses the trained model to predict mutation testing results of newer versions of the project without executing the mutants. There are two different ways of cross-version approach.

In the first case, the classification model is trained using mutants of one version of the program and applied to the mutants of the next version of the program to make predictions. In other words, to apply PMT to a version of the program, the classification model needs to be trained using an immediate previous version of this program. Mutants of the version (v) of the program are the test set while mutants of the previous version (v-1) are used as a training set to build the classification model. The detailed experimental results are presented in **Table 7** for two different configurations of PIT tool [5]. The first configuration generates mutants using the "Default" group of mutators and the second configuration creates mutants using the "All" group of mutators. PIT has a parameter named *mutators* and passing the name of a group in this parameter will generate all types of mutants from that group. In further experiments "All" group of mutators is used.

Project	Mutators	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
apns	Default Mutators	v0-v1	0.944	0.934	0.979	0.956	0.979	3
		v1-v2	0.938	0.903	0.982	0.941	0.956	4.4
		v2-v3	0.935	0.917	0.951	0.933	0.967	1.8
		v3-v4	0.904	0.906	0.911	0.909	0.976	0.3
		v4-v5	0.923	0.934	0.934	0.934	0.976	0
	All Mutators	v0-v1	0.932	0.92	0.968	0.943	0.965	3
		v1-v2	0.915	0.883	0.949	0.915	0.974	3.6
		v2-v3	0.933	0.917	0.945	0.931	0.982	1.4
		v3-v4	0.921	0.898	0.953	0.924	0.976	3.1
		v4-v5	0.911	0.951	0.891	0.92	0.974	3.6
la4j	All Mutators	v0-v1	0.886	0.859	0.877	0.868	0.96	0.9
		v1-v2	0.931	0.908	0.937	0.922	0.983	1.4
		v2-v3	0.899	0.85	0.945	0.895	0.953	5.1
		v3-v4	0.853	0.904	0.819	0.86	0.936	5.2
		v4-v5	0.905	0.893	0.928	0.91	0.955	2.1
		v5-v6	0.928	0.9	0.977	0.937	0.979	4.6

Table 7 The results of Random Forest for default mutants and all mutants

As **Table 7** shows PMT performs exceptionally well under this application scenario. All the Prediction Errors are below 6% and most of the metric values are above 0.90. These results also show that PMT has similar metrics values on different projects.

For comparison **Table 8** presents results from the paper [2] devoted to PMT performance for the same experiment that is presented in **Table 7**. In the paper, they used the Random Forest algorithm and the naive imbalanced data for building the prediction model. Column “Sub” shows the name of a project. Column “changes” correspond to the differences between the numbers of lines of code of two versions of the program. All the metric measures are self-explanatory. Column “Err” indicates to Prediction Error.

Sub.	Ver.	changes	Prec.	Recall	F.	AUC	Err.
apns	v0-v1	193	0.949	0.949	0.949	0.988	2.05%
	v1-v2	362	0.915	0.914	0.914	0.983	2.10%
	v2-v3	0	0.966	0.966	0.966	0.997	0.35%
	v3-v4	67	0.947	0.946	0.946	0.992	1.89%
	v4-v5	215	0.925	0.925	0.925	0.981	1.36%
la4j	v0-v1	994	0.913	0.911	0.911	0.968	2.73%
	v1-v2	270	0.945	0.945	0.945	0.991	1.90%
	v2-v3	190	0.912	0.908	0.908	0.968	5.27%
	v3-v4	938	0.841	0.826	0.829	0.935	-7.48%
	v4-v5	167	0.914	0.913	0.911	0.96	4.50%
	v5-v6	949	0.927	0.926	0.925	0.972	4.43%

Table 8 The results of Random Forest from the paper

According to **Table 8** Prediction Errors are all below 6% and most of the metric values are above 0.9. These results are quite similar to the results in **Table 7**. All the metrics are very close to the ones from the paper for both Java projects. More precisely, F-measure values differ maximum by 0.05 and minimum by 0.001, ROC-AUC values differ maximum by 0.03 and minimum by 0.001. Prediction errors are very close too. Overall, this means that the replication of results for this part was successful. Small differences between values are expected because the conditions of the experiment were not exactly the same. For example, different tools were used for building classification models.

In the second version of the cross-version scenario, the impact of version intervals on the performance is investigated. The first version of a project is used as the train set and all the other versions are used as test sets. The classification model is trained using the first version of the program. The trained model is used to make predictions for the mutants of newer versions of the program.
Detailed results are shown in **Table 9**. ROC-AUC values are all above 0.90, all the other metrics values are above 0.82 and prediction error values are below 9%. Note that as version difference increases ROC-AUC value decreases. This is an effect of changes between the versions, more changes are made to the project more different versions are. This effect is shown in Figure 2 and Figure 1. For comparison, the results of the same experiment from the paper [2] are presented in Figure 3 for both java projects.

Project	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
apns	v0-v1	0.944	0.934	0.979	0.956	0.979	3
	v0-v2	0.92	0.879	0.977	0.925	0.959	5.6
	v0-v3	0.893	0.832	0.975	0.898	0.951	8.3
	v0-v4	0.896	0.849	0.974	0.907	0.948	7.7
	v0-v5	0.899	0.879	0.959	0.917	0.95	5.3
la4j	v0-v1	0.886	0.859	0.877	0.868	0.96	0.9
	v0-v2	0.883	0.85	0.886	0.868	0.955	1.8
	v0-v3	0.871	0.829	0.901	0.864	0.931	3.9
	v0-v4	0.831	0.896	0.784	0.836	0.922	6.9
	v0-v5	0.849	0.868	0.834	0.851	0.926	2.1
	v0-v6	0.849	0.846	0.886	0.865	0.914	2.6

Table 9 The results of Random Forest showing Impact of version intervals

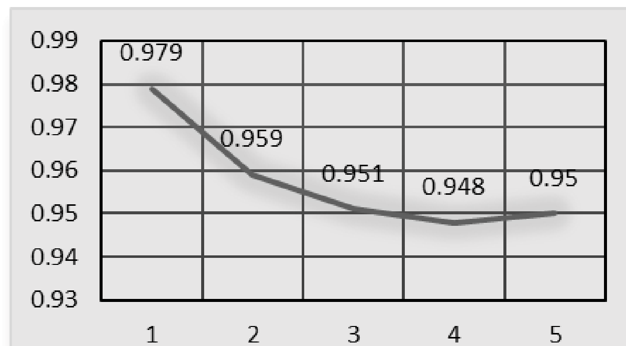

Figure 2 Effect of version intervals on ROC-AUC of apns project

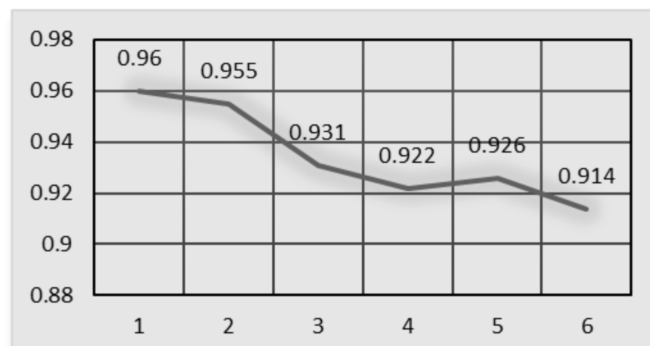

Figure 1 Effect of version intervals on ROC-AUC of la4j project

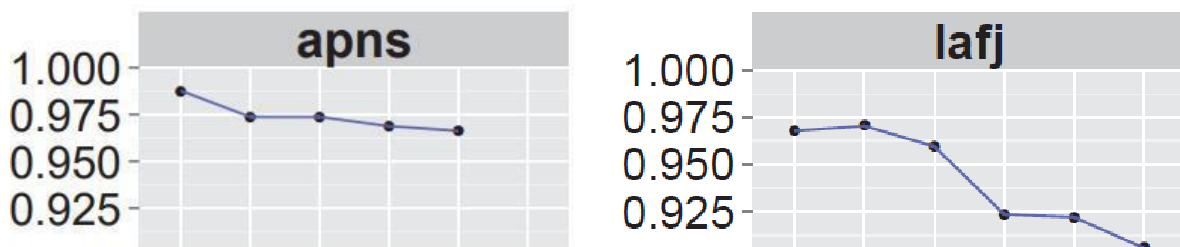

Figure 3 Effect of version intervals on ROC-AUC from paper [2]

Table 9 shows that PMT performs very well even when versions are very different from each other. This means that a model can be trained on a version and used to make predictions on the several following versions with high accuracy.
The effect of intervals shown in Figure 2 and Figure 1 also demonstrates the accuracy of replication of the results by being so close to the results presented in the paper [2] shown in Figure 3. The values on the graphs are close to each other for both projects. Using two different projects also supports finding. Apns project ROC-AUC values simply decrease so the replicated results are close to original ones. Whereas la4j replicated values act the same way as original ones and reinforce the reliability of replication results.

The default model used in this section is Random Forest classifier with default parameters. All the 14 features are collected and used for 6 versions of apns project and 7 versions of la4j project. The list of features and their importance is shown in Figure 4 and Figure 5. Feature Importance values are extracted from the model which was trained on the first version (v0) of the program and evaluated on the second version (v1). Categorical features are: typeReturn and typeOperator. One-hot encoding was used to convert categorical features into numeric. Categorical feature importance was averaged in order to get the overall importance.

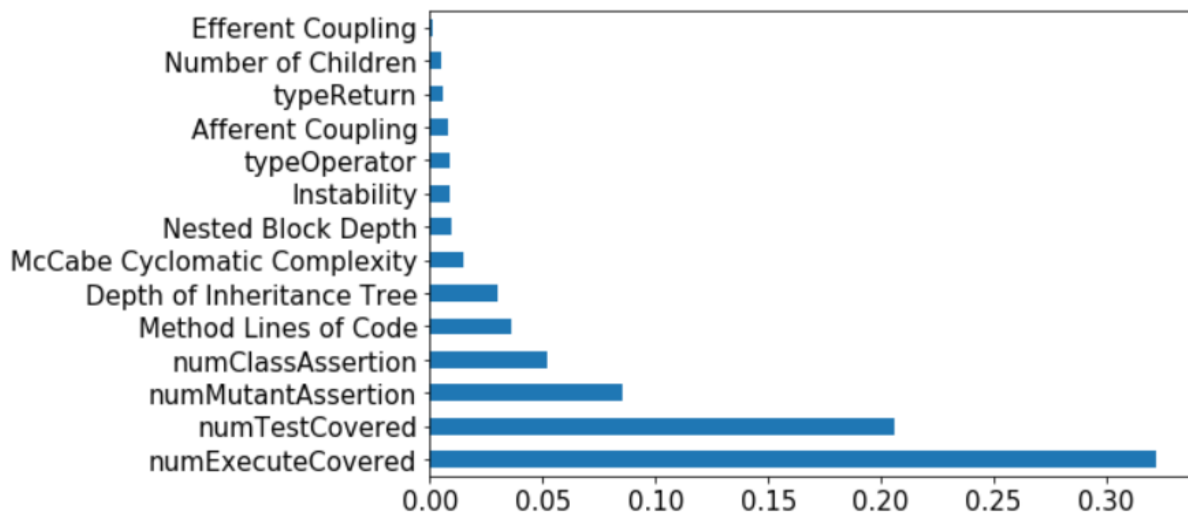

Figure 4 Feature importance of apns project

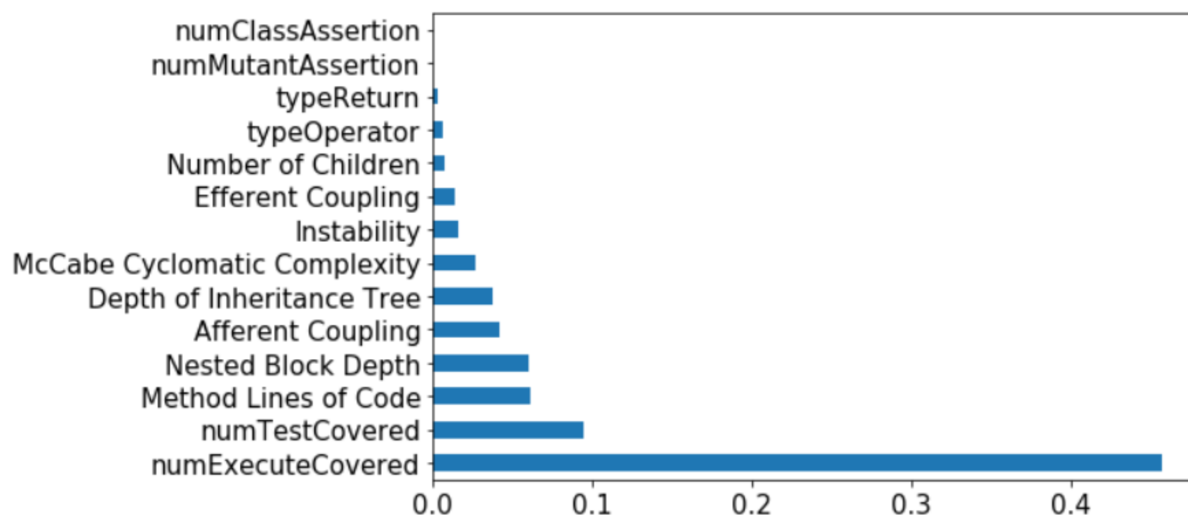

Figure 5 . feature importance of la4j project

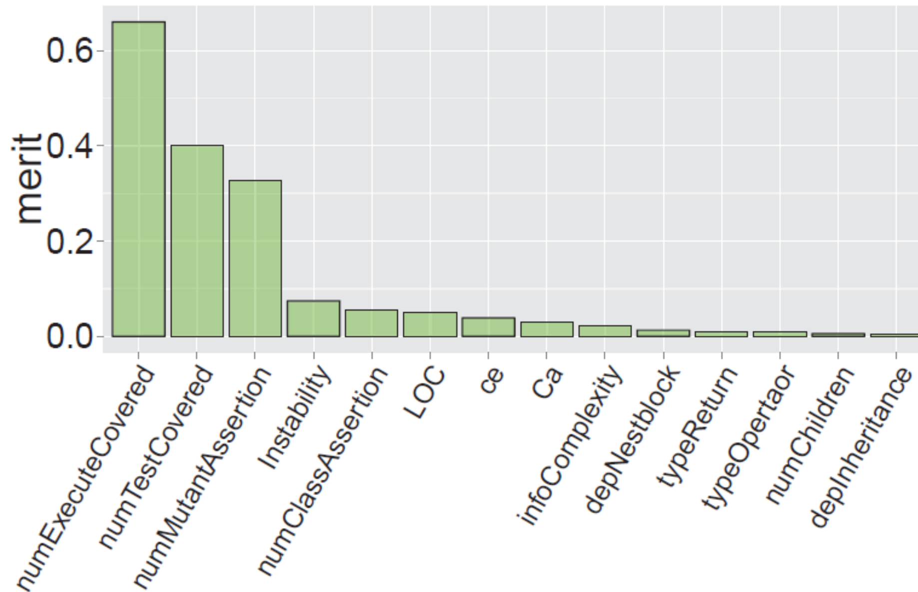

Figure 6 Feature importance from the paper [2]

As we can see from the above figures feature importance is similar between replicated results and original results. The first two most important features are numExecutedCovered and numTestCovered. These two features are related to the test suite so it is logical that they contribute more than other features.

As mentioned above, PIT has 4 different execution results of mutants. Binary classification requires to have only two different labels: positive and negative. Because of this 1 (positive) is defined as killed and 0 (negative) is defined as survived. The data also contains *no coverage* and *time out* labels. To solve this problem, no coverage was converted as survived since no test covers such mutants they will survive test executions. Whereas time out was converted as killed because time out itself means that there is an infinite loop. Hence program behavior is different from the original program behavior and this means that such mutants will be detected.

Note that information about the mutants that are not located in any method are discarded from the dataset because some features cannot be extracted for them. For example, OpenClover generates feature numExecutedCovered only for method lines and not for other lines. If a class code contains a member variable declaration a mutant can be generated for such code but obviously, those lines of code do not have return type or method lines of code. Even though OpenClover does not produce numExecutedCovered for such lines of code, there might be some tests that cover it. Therefore, these types of mutants cannot be treated as not covered. Overall, information for those kinds of mutants cannot be obtained. The best way from this situation is to remove them from training and testing data. Their number is quite insignificant. For the first version of the apns project, only 14 such mutants are detected and removed.

From this point on the dataset used for training and testing is the one generated with the “All” group mutator operators. The justification for this choice is that there is no big difference between the performance of classifiers trained using the dataset of “All” and “default” mutator operators. “All” mutator operators generate more mutants and those mutants also include the ones generated by “default” mutator operators. For instance, all mutator types and their counts of the first version of apns project are presented in **Figure 7**.

```
NonVoidMethodCallMutator                 107
InlineConstantMutator                     54
ReturnValsMutator                         52
NullReturnValsMutator                     43
ConstructorCallMutator                    43
MemberVariableMutator                     40
VoidMethodCallMutator                     35
NegateConditionalsMutator                 25
ArgumentPropagationMutator                22
MathMutator                               21
RemoveConditionalMutator_EQUAL_IF         18
RemoveConditionalMutator_EQUAL_ELSE       18
NakedReceiverMutator                      13
RemoveConditionalMutator_ORDER_IF          7
RemoveConditionalMutator_ORDER_ELSE        7
ConditionalsBoundaryMutator                7
EmptyObjectReturnValsMutator               3
BooleanTrueReturnValsMutator               3
IncrementsMutator                          3
PrimitiveReturnsMutator                    3
RemoveIncrementsMutator                    2
```

Figure 7 “All” group mutator types and their counts

Besides Random Forest classifier, there are several other classifiers that can be used for this type of problem. In this paper, SVM is used to see if it can perform better and make better predictions than Random Forest classifier. Using all the 14 features SVM is evaluated on apns project. Default parameters values are left. The results of the SVM are displayed in **Table 10**.

classifier	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
SVM	v0-v1	0.765	0.93	0.78	0.848	0.846	13.7
	v1-v2	0.777	0.859	0.865	0.862	0.698	0.6
	v2-v3	0.771	0.944	0.758	0.841	0.9	15.7
	v3-v4	0.812	0.878	0.893	0.886	0.722	1.4
	v4-v5	0.821	0.92	0.869	0.894	0.782	4.8

Table 10 The results of SVM on apns project

From the results of SVM on apns project it is clear that Random Forest gives better performance. SVM has very unsteady performace according to ROC-AUC values and Prediciton Errors. The paper [2] also says that it performs much worse than Random Forest base on all evaluation measures.
The replication of the paper [2] results ends here. As we have seen, the results of two application scenarios can be successfully replicated: cross-project and cross-version. Since cross-version scenario was performed under the similar conditions it has much closer results to the paper [2]. Overall, these finding prove that PMT is very effective for evaluating test suite quality. PMT is able to predict the execution results of mutants without their executions and it does this with high accuracy.

4.1.2.2 Fine tuning classifier parameters

Fine-tuning Random Forest classifier parameters can improve performance. To identify the best combination of parameter values for a model a set of possible values are chosen and given to the model for training. Several models are trained using combinations of parameter values. The best estimator is the model with the highest values of ROC-AUC measure.
The best estimator was found with the following parameters: max_depth = 15, max_features = 20, n_estimators = 200. The evaluation of this classification model is presented in **Table 11** alongside Random Forest results from the paper [2] on the right side of the table (AUC and Err metrics).

train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error	AUC	Err.
v0-v1	0.932	0.923	0.963	0.943	0.982	2.5	0.988	2.05%
v1-v2	0.924	0.895	0.955	0.924	0.978	3.3	0.983	2.10%
v2-v3	0.938	0.92	0.953	0.936	0.989	1.7	0.997	0.35%
v3-v4	0.923	0.888	0.969	0.927	0.981	4.6	0.992	1.89%
v4-v5	0.92	0.944	0.916	0.93	0.979	1.7	0.981	1.36%

Table 11 Metrics values of Random Forest on apns project

Tools used for extracting features are different as well as tools used for machine learning part (Machine learning part was done using Jupyter Notebook). However, the ROC-AUC metric values differ only by 0.02 value and Prediction Error values are very similar.

According to the results from **Table 11**, if we set several parameter values of classification model the performance improves (based on ROC-AUC measure) compared to using default parameter values (results presented in **Table 7** with “All” mutators). Consequently, this estimator is used for further experiments. Feature importance of the best estimator is presented in **Figure 8**. As we can see they are similar to the ones where model is trained with default parameters.

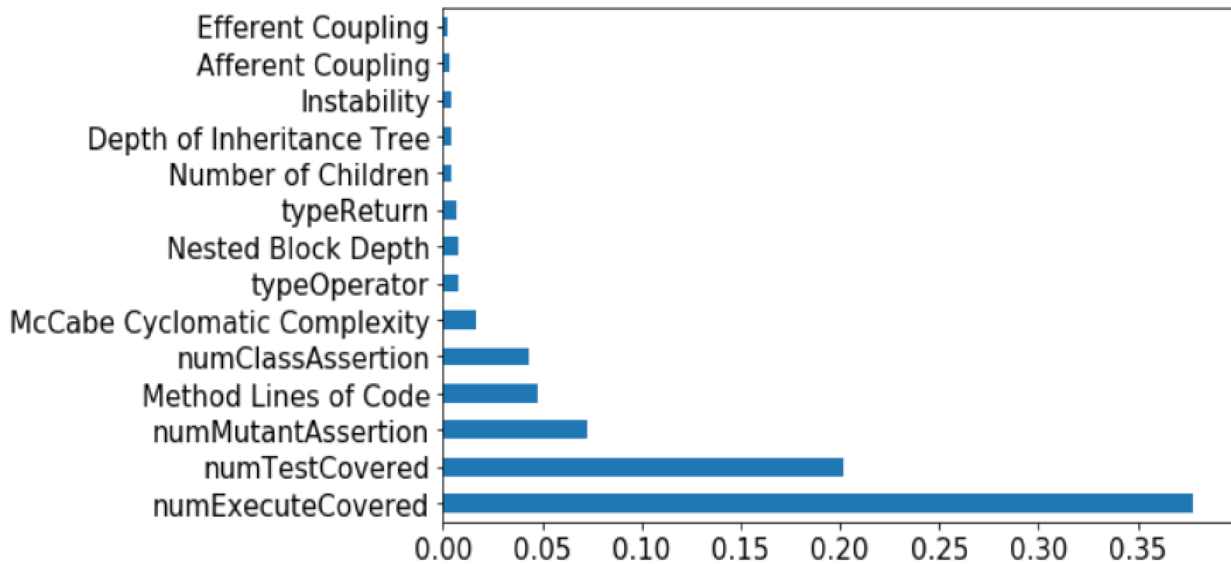

Figure 8 Feature importance of Random Forest on apns project

4.1.2.3 Effect of removing mutants with no coverage

When there is a mutant with no coverage it is easy to say that it will survive because there is no test that covers it. Because of this, it can be useful to inspect the effect of removing such mutants from the dataset. All the mutants that have label no coverage are removed. For example, out of 526 mutants of the first version (v0) of the apns project, only 433 are left. The distribution of killed mutants is 0.86%. This means that the majority of the mutants are killed. This causes feature importance to change greatly and the performance to decrease slightly. **Table 12** present the effect of removing mutants with no coverage. Random Forest was trained on the first version of apns project and evaluated on the second version. ROC-AUC value decreased by 0.055 and Prediction Error increased by 1.1. New feature importance is presented in **Figure 9**. The most important feature numExecuteCovered had the value of importance higher than 0.35 when no coverage mutants were included in the training dataset. Since now all the mutants are covered numExecutedCovered decreased and it is below 0.14. **Table 13** and **Table 14** present the Confusion Matrix of both models. As we can see the majority of the survived mutant are removed but despite this, classifications of other mutants are quite good.
The motivation behind this experiment is that there is no need to predict the execution results of a mutant with no coverage. It can be classified as survived. This means that they can be removed from the training dataset for big projects and make the training process faster.
The result of this experiment shows that even without mutants with no coverage good results can be obtained.

no coverage mutants	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
included	v0-v1	0.932	0.923	0.963	0.943	0.982	2.5
removed	v0-v1	0.905	0.925	0.965	0.945	0.927	3.6

Table 12

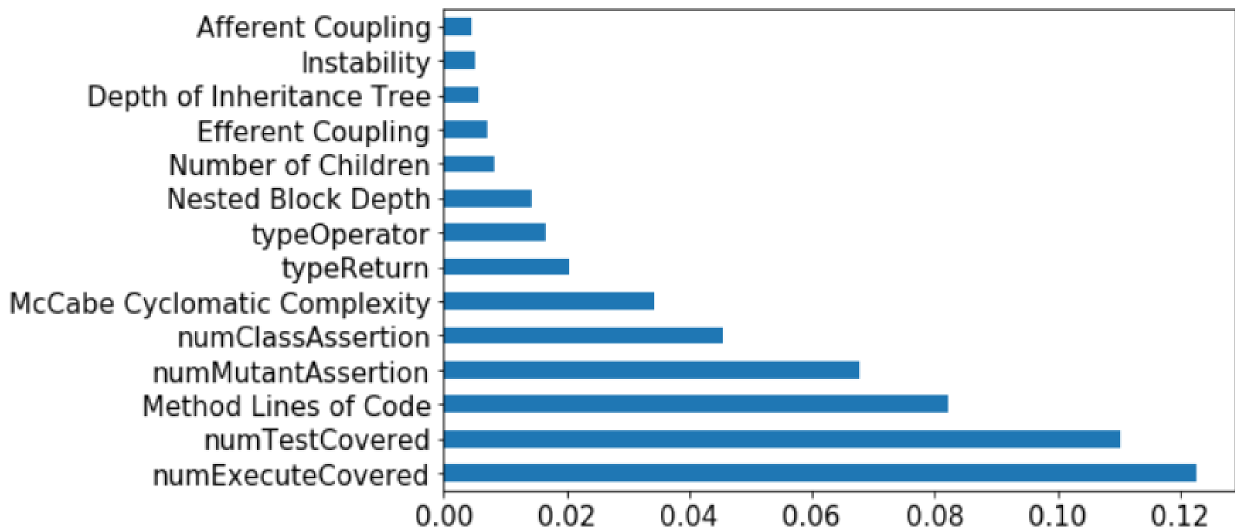

Figure 9. Feature importance of Random Forest on apns project

true/predicted	Survived	Killed	All
Survived	289	37	326
Killed	17	446	463
All	306	483	789

Table 13 Confusion matrix of Random Forest on apns project. Mutants with no coverage mutants are included

true/predicted	Survived	Killed	All
Survived	49	36	85
Killed	16	447	463
All	65	483	548

Table 14 Confusion matrix of Random Forest on apns project. Mutants with no coverage mutants are removed

4.1.2.4 Balancing the data

In mutation testing it is preferable to have a false alarm (misclassifying a mutant as survived while it is in fact killed) rather than no alarm (misclassifying a mutant as killed while it is in fact survived). Survived mutants correspond to the bugs in the code, hence failing to detect survived mutants is the same as failing to detect hidden bugs in the code.

When no coverage mutants are removed from the dataset, the distribution of classes killed and survived becomes very unbalanced. For example, after removing such mutants from the first version of the apns program distribution of killed mutants becomes 86%. For mutation testing problem it is critical to correctly classify survived mutants. Survived mutants indicate parts of the test suite that need to be improved. Most classification algorithms do not work well on imbalanced data problems because they try to minimize the total error rate rather than pay more attention to the minority class. This paper investigates two different techniques for handling imbalanced data: under-sampling and cost-sensitive learning. The results of both approaches are presented in **Table 15** alongside the result of row imbalanced data. The aim to correctly classify more survived mutants was achieved using balancing techniques. The confusion matrices show this in **Table 16.**

Balancing technique	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
under-sampling	v0-v1	0.752	0.963	0.734	0.833	0.869	20.1
cost sensitive learning	v0-v1	0.892	0.935	0.937	0.936	0.924	0.2
none	v0-v1	0.905	0.925	0.965	0.945	0.927	3.6

Table 15 The results of balancing techniques on apns project

	true/predicted	Survived	Killed	All
under-sampling	Survived	72	13	85
	Killed	123	340	463
	All	195	353	548
cost sensitive	true/predicted	Survived	Killed	All
	Survived	55	30	85
	Killed	29	434	463
	All	84	464	548
none	true/predicted	Survived	Killed	All
	Survived	49	36	85
	Killed	16	447	463
	All	65	483	548

Table 16 Confusion matrixes of balancing techniques

Table 15 shows that out of 2 balancing techniques cost-sensitive method performs much better according to all the metrics. The performance of a model without any balancing also performs well and achieves similar results as the cost-sensitive method but Confusion matrixes of both methods show that balancing helps to correctly classify more mutants as survived. For this reason, the cost-sensitive approach was chosen to use for further experiments.

4.1.2.5 Location as a feature

Each line of the code has the same coverage from the test suite. For each statement, there might be several mutants generated. For such mutants, test-related features are the same because they are located at the same place in code. From the above experiments, we learned that test-related features are important during predictions. Is it interesting to see if a location feature can be a good indicator of the outcome of a mutant execution. For example, if there are two mutants generated on the same line and the execution result is known for one of them can we say that the second mutant will have the same outcome or not. This paper investigates if the addition of the location feature will affect performance.

The location of the mutant consists of information about package, source (file), class, method and line in which this mutant is located in the application code. Features related to the test suite contain information for each method, therefore including the line information in the location feature becomes less important. Statistics were collected for each line and analyzed. The results show that the mutants on the same line mostly have the same execution result. Based on these findings the location feature was added to the feature set. Note that the location feature is added as a four different feature: package, source, class and methodName. They are all categorical features. For converting categorical features into numeric three approaches were used:

- One-hot encoding
- Replacing the variables by the respective frequencies counts of the variables in the column
- Label encoding

The results of all three techniques are presented on **Table 17**. ROC-AUC values are around 0.92 and the Prediction Errors are around 2%. Compared to the results without location feature all the metrics are very close.

encoding	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
one hot	v0-v1	0.898	0.928	0.952	0.94	0.913	2.2
frequencies	v0-v1	0.894	0.926	0.95	0.938	0.918	2.2
labels	v0-v1	0.898	0.93	0.95	0.94	0.916	1.8
no location feature	v0-v1	0.892	0.935	0.937	0.936	0.924	0.2

Table 17 The results of three different encoding of categorical features and the results without location feature

Feature importance for all three encoding approaches is presented in **Figure 10**, **Figure 11** and **Figure 12**. One-hot encoding feature importance is hard to estimate since all the newly added features have many different values. Averaging them gives very low importance values shown in **Figure 10**.
As we can see in all these figures location feature has high values of importance. Using frequency encoding class, source and methodName are the most important features after numExecutedCovered, numTestCovered and numMutantAssertion which all are test related features. This finding supports the idea that location can be a good indicator of the execution results of mutants.

For future experiments, the frequency encoding approach was chosen according to ROC-AUC measure which is slightly higher than others.
Note that Java projects have two other categorical features: typeReturn and typeOperator. One-hot encoding is used for them. Since there are only several types of them using one-hot encoding will not add too many features after encoding.

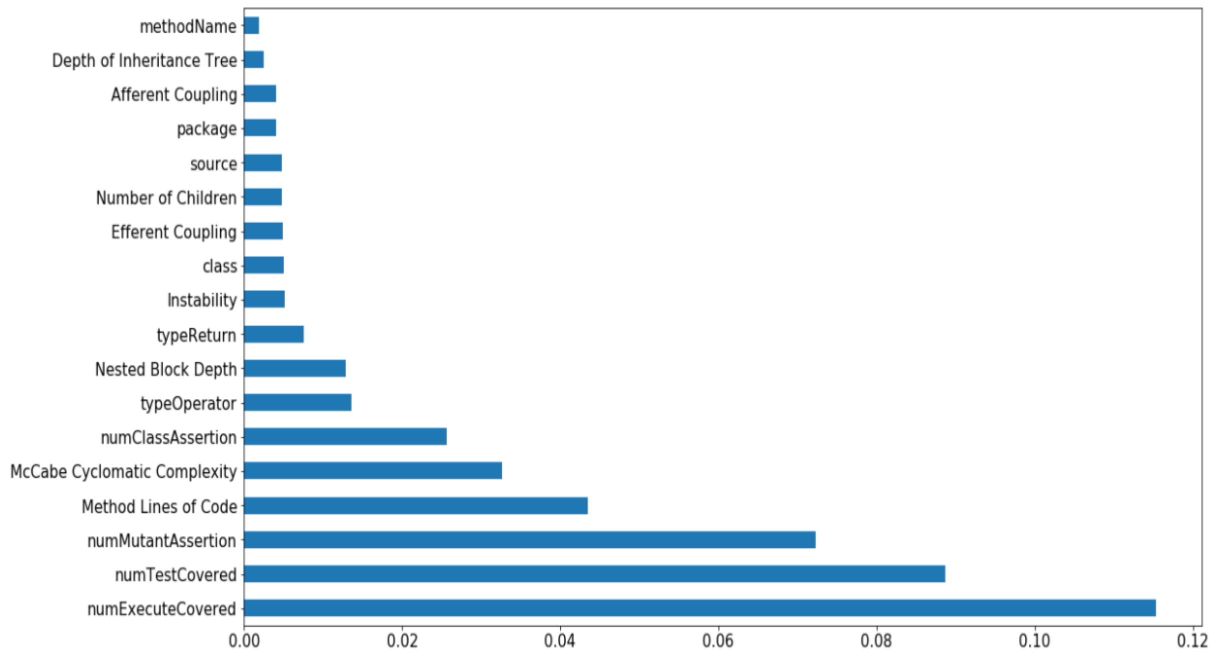

Figure 10 The feature importance of one hot encoding

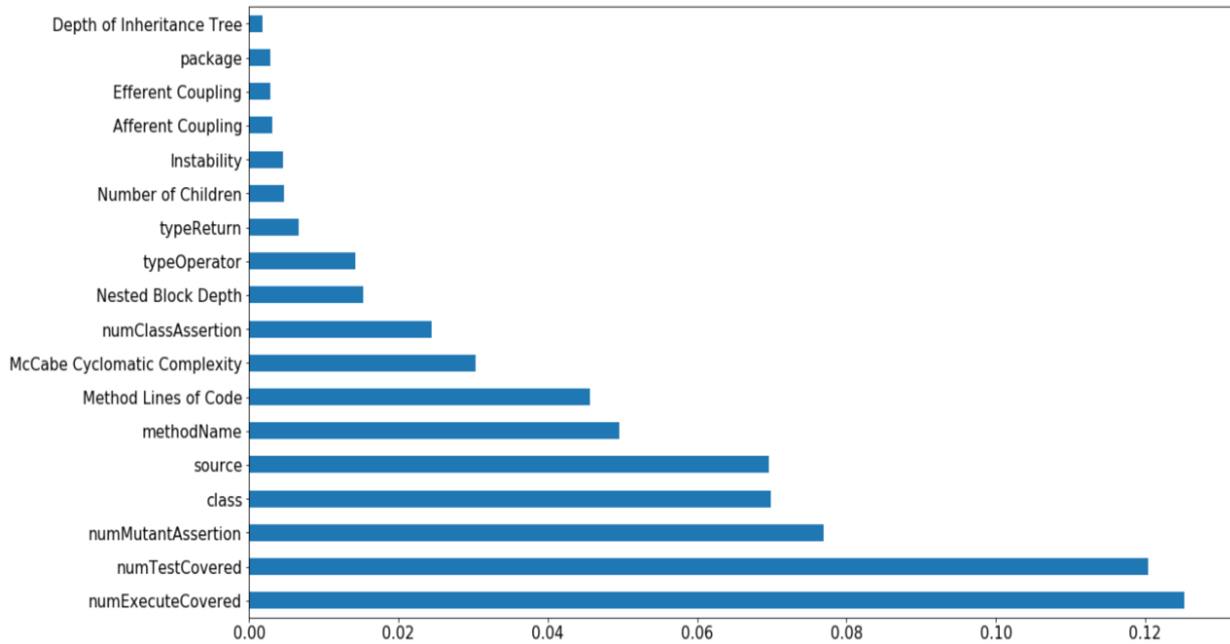

Figure 11 The feature importance of frequency encoding

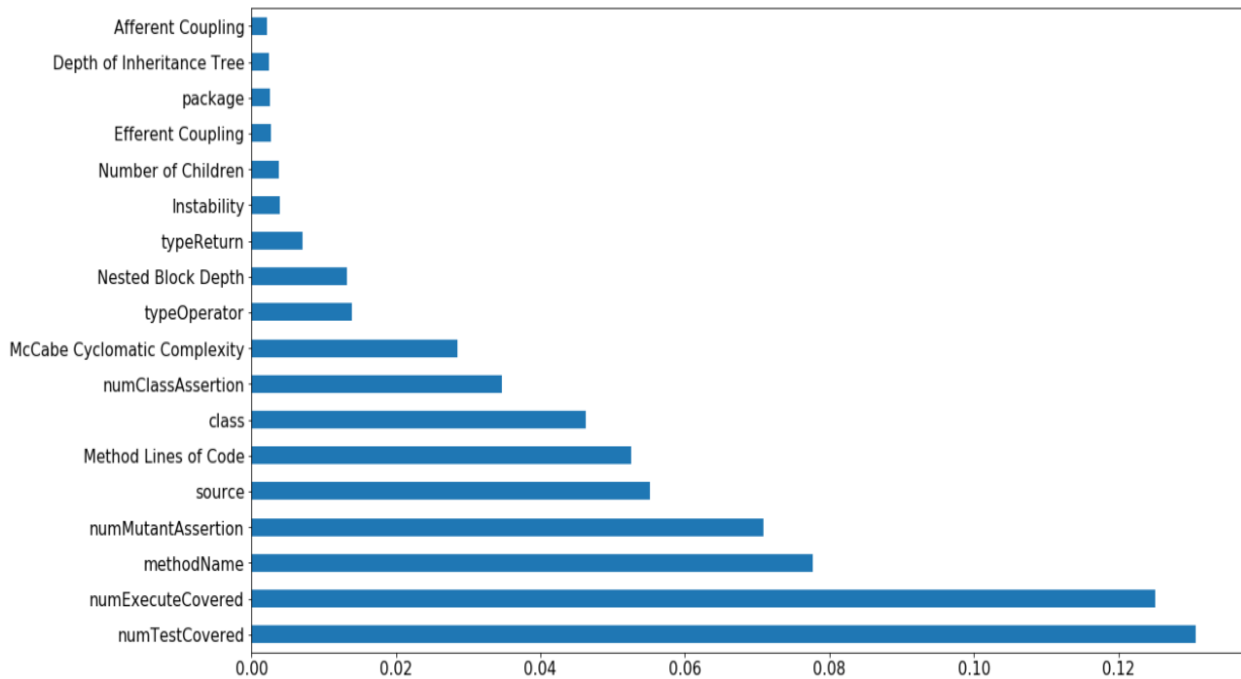

Figure 12 The feature importance of label encoding

4.2 Results for RQ2: Transfer PMT from Java to C

This section answers the second research question. 4.2.1 section presents the results of C and Java projects under the same conditions (using the same classifier and same feature set). 4.2.2 section presents the general results of C and Java projects (using all their features).

The process of feature extraction for C project is different from that of Java project. All the tools used for collecting features are different. Machine learning part is the same for both projects. 24 unique features were extracted for C project. Distribution of survived and killed mutants is 0.57%, more precisely, out of 58432 mutants 33419 are killed. Categorical features are typeOfMutant, typeReturn, location and Mutation. The location feature is given in a form of a string that represents location hierarchy: package and class that mutant is located in. All categorical features are converted using one-hot encoding. For all the experiments except one C project data is divided into train and test sets. 80% is train and 20% is test set. Train data has 46745 mutants and Test data has 11687 mutants.

4.2.1 Common features of C and Java projects

C and Java projects have several features in common. This shows that the features of C project and Java project are comparable. From both projects similar feature can be extracted.

In this section, only those common features are used to build the Random Forest models and evaluate them. Note that not all the chosen features are the same as what was used in Java projects but they are similar. Chosen features from C project are:

- typeOfMutant (typeOperator)
- numTestCovered (numTestCovered)
- numMutationAssertions_iparam (numMutantAssertion)

- numMutationAssertions_oparam (numMutantAssertion)
- numClassAssertions (numClassAssertion)
- typeReturn (typeReturn)
- mccabe (McCabe Cyclomatic Complexity)
- sloc (Method lines of code)
- lines (Method lines of code)

Random Forest was trained on apns Java project and C project using default parameters. For apns two versions were used: the first version for training and the second for evaluation. Single C project version was used and the data was divided into train and test sets. The outcome of the models is presented in **Table 18** with the results of the same experiment performed on the Java project. The feature importance of both projects is presented in **Figure 13** and **Figure 14**. The Confusion Matrixes of both models are shown in **Table 19**.

Language	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
Java	v0-v1	0.88	0.927	0.931	0.929	0.917	0.4
C	80%-20%	0.734	0.789	0.736	0.762	0.822	3.9

Table 18 Java and C project results of common feature set

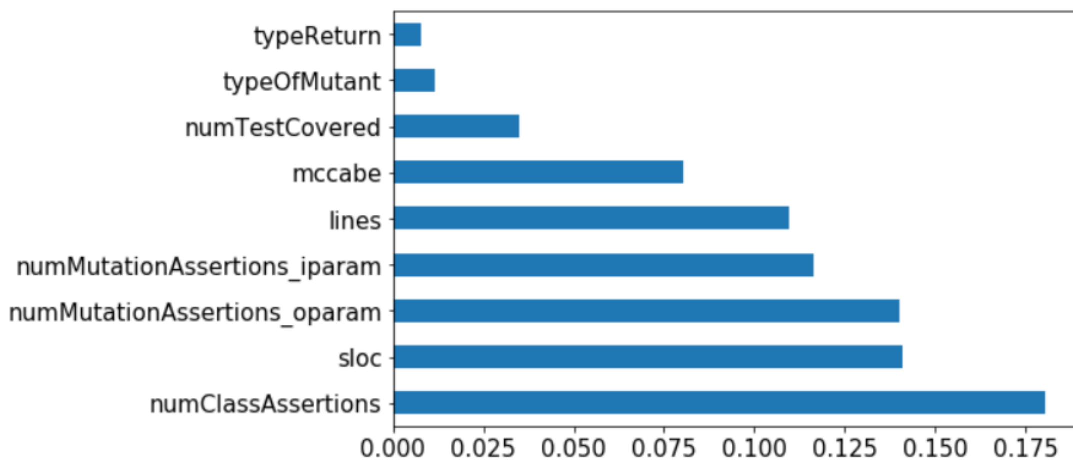

Figure 13 Feature importance of Random Forest on C project

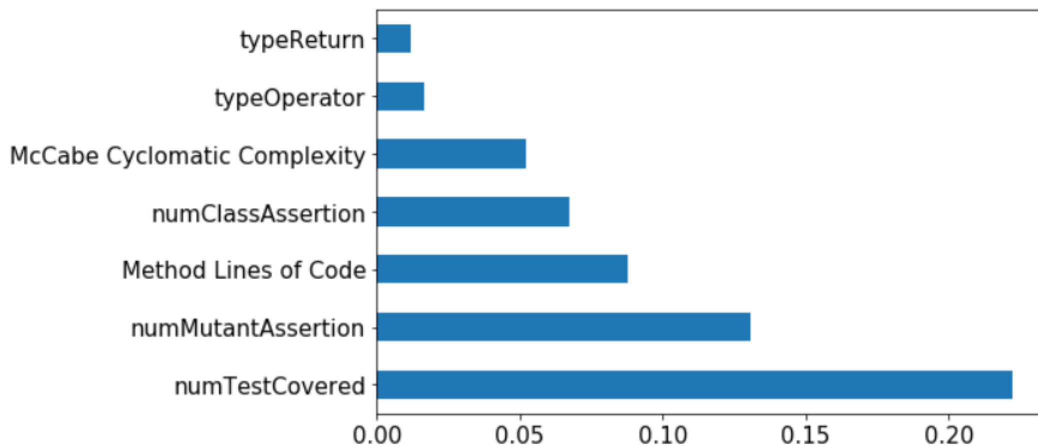

Figure 14 Feature importance of Random Forest on apns Java project

	true/predicted	Survived	Killed	All
Java	Survived	51	34	85
	Killed	32	431	463
	All	83	465	548
C	true/predicted	Survived	Killed	All
	Survived	3608	1329	4937
	Killed	1781	4969	6750
	All	5389	6298	11687

Table 19 Confusion Matrixes

As we can see on **Table 18** apns performance is better than C project. This might be a result of using cross-version approach for Java project. Since the model is trained using similar mutants that might help the performance. Whereas for C project single version is used and so model might have not seen similar mutants. Feature importance of both projects are very different from each other which indicates that different languages has different characteristics and if numTestCovered is important for Java project it does not mean that it will be as important for C project.

Java project (apns) has a much less number of mutants than C project. Therefore, two versions of the program are used to evaluate PMT for Java. Besides, C project data does not contain mutants with no coverage so they were removed from the Java project data as well. To make the conditions similar for each project cost-sensitive balancing was used for Java project because after removing no coverage data train set contains much more killed mutants than survived. Whereas in C project both classes have a similar number of mutants. In addition, all the 6 versions of Java project were used under the single version scenario. The data of each version was divided into a train (80%) and test (20%) sets and evaluated. For C project dataset was split into two sets 5 times. Each time different size train and test sets were generated. All the results are presented in **Table 20**.

Since there are many differences between these projects the most similar cases are C project with naïve Random Forest and Java project with the single version with balanced Random Forest and no coverage mutants removed. According to the results of this case, C project ROC-AUC is 0.822 when divided into 80-20% of a train and test sets. With the same division, this result is closest to Java project v2 which has 0.803 value and it is farthest from v5 value 0.736. This proves that using common conditions gives similar results.
Table 20 answer to second research question. The results between Java and C projects are comparable when testing under the same conditions. Using the similar set of feature and classification models. For all versions of apns project ROC-AUC values are less then C project values.

Language	balanced	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
Java	no	v0-v1	0.876	0.925	0.929	0.927	0.878	0.4
		v1-v2	0.863	0.893	0.942	0.917	0.895	4.4
		v2-v3	0.889	0.916	0.947	0.931	0.945	2.7
		v3-v4	0.873	0.922	0.922	0.922	0.916	0
		v4-v5	0.899	0.939	0.946	0.942	0.893	0.6
	yes	v0-v1	0.874	0.921	0.931	0.926	0.892	0.9
		v1-v2	0.86	0.909	0.918	0.913	0.886	0.8
		v2-v3	0.881	0.933	0.917	0.925	0.936	1.4
		v3-v4	0.883	0.936	0.919	0.927	0.907	1.5
		v4-v5	0.886	0.947	0.921	0.934	0.907	2.4
	yes	v0 (80%-20%)	0.897	0.901	0.986	0.942	0.8	8
		V1 (80%-20%)	0.827	0.903	0.894	0.898	0.772	0.9
		V2 (80%-20%)	0.789	0.868	0.868	0.868	0.803	0
		V3 (80%-20%)	0.767	0.829	0.87	0.849	0.764	3.8
		V4 (80%-20%)	0.779	0.897	0.821	0.857	0.785	6.9
		V5 (80%-20%)	0.824	0.881	0.91	0.895	0.736	2.7
C	no	90%10%	0.746	0.766	0.814	0.789	0.821	3.7
		80%-20%	0.734	0.789	0.736	0.762	0.822	3.9
		70%-30%	0.741	0.753	0.815	0.783	0.818	4.7
		60%-40%	0.741	0.76	0.803	0.781	0.818	3.2
		50%-50%	0.738	0.76	0.793	0.777	0.815	2.5

Table 20 The results PMT on C and Java projects

4.2.2 Performances of C and Java projects

This section presents an evaluation of PMT on C project. For the evaluation, all the available 24 features were used to build Random Forest classification model. The results are shown in **Table 21** and **Table 22**. alongside with apns Java project results with all its features and mutants with no coverage removed since in C project there are no such mutants. The ROC-AUC values are almost the same for both projects and Prediction Errors are close. These results indicate that if we use all the resources available for different languages (C and Java) similar results can be obtained.

Language	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
Java	v0-v1	0.894	0.926	0.95	0.938	0.918	2.2
C	80%-20%	0.841	0.86	0.867	0.863	0.916	0.5

Table 21 The results PMT on C and Java project

	true/predicted	Survived	Killed	All
Java	Survived	50	35	85
	Killed	23	440	463
	All	73	475	548
C	true/predicted	Survived	Killed	All
	Survived	3982	955	4937
	Killed	901	5849	6750
	All	4883	6804	11687

Table 22 Confusion matrixes

Figure 15 presents feature importance of C project. As we can see the most important features are Column and Line which indicate location of a mutant in a code. numMutationAssertions are quite important feature as well and the most important feature related to test suite. Most of the feature are different from Java project ones. In Java projects when mutants with no coverage are removed the most important feature has similar values as Column feature does in C project.

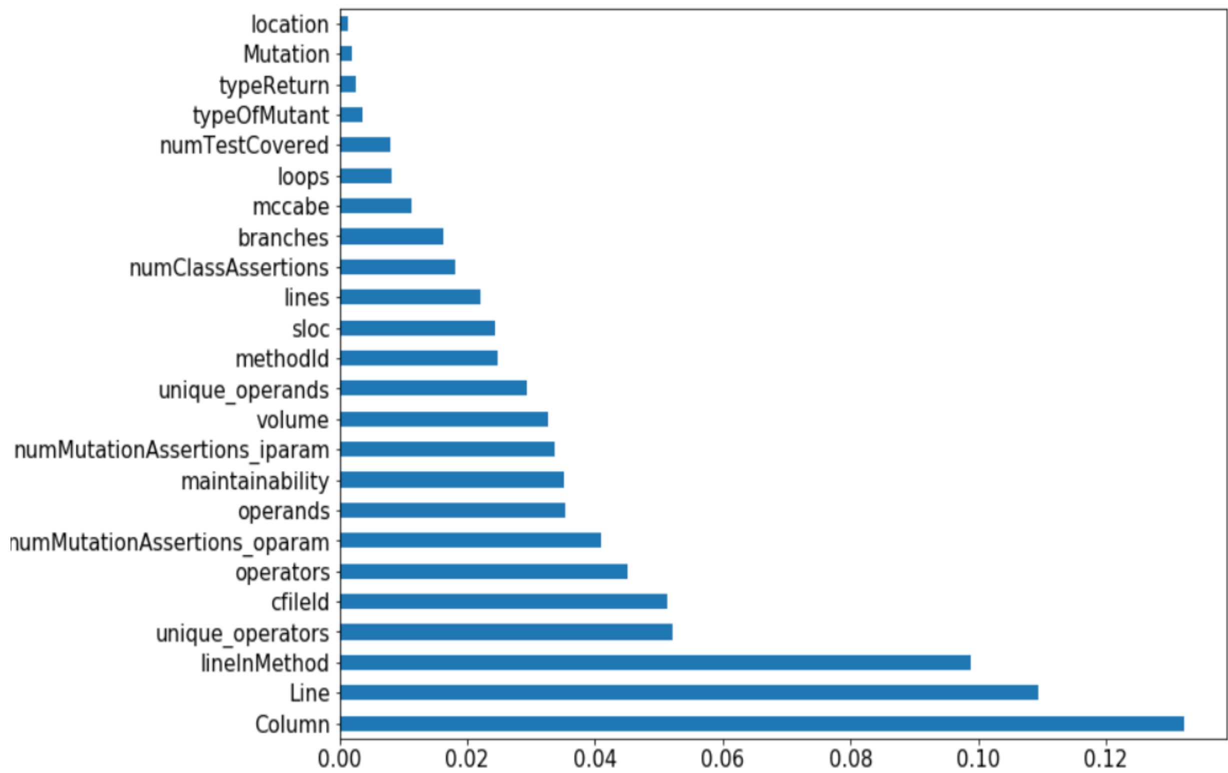

Figure 15 feature importance of C project

Overall, some of the features of C and Java projects are comparable and the results indicate that performances are comparable as well.

4.2.3 Feature selection of C project

This section investigates the effect of using only several features for C project. The motivation behind this is to see how good results can be obtained using only several features. According to the previous experiment results out of 24 features several location features are dominating: Column, Line, methodId, cfileId and lineInMethod. Because of this reason those location features were chosen for training. Besides these location features some test-related feature were added: numMutationAssertions_iparam, numMutationAssertions_oparam and numClassAssertions. Lines and sloc features are related to location feature so they were added also. In total 10 features were used. Furthermore, cfileId and methodId were dropped, used only 8 features for training. Lastly, only location features were used for training: Column, Line, methodId, cfileId and lineInMethod. The results are shown in **Table 23**. Their feature importance is in **Figure 16**. As we can see we can obtain almost the same performance using only several important features. This finding can greatly help PMT. Collecting only 10 or 5 features is much easier than collecting 24 features. Besides location feature are very easy to collect, unlike test related features. For test-related features, like numExecutedCovered, OpenClover needs to be executed.

Using only a few features can improve the efficiency of PMT with a small loss of accuracy and findings of this experiment supports this claim.

features	train-test	Accuracy	Precision	Recall	F-measure	ROC-AUC	Pred.Error
all	80%-20%	0.841	0.86	0.867	0.863	0.916	0.5
10	80%-20%	0.864	0.865	0.906	0.885	0.916	2.8
8	80%-20%	0.867	0.868	0.908	0.888	0.919	2.7
5	80%-20%	0.861	0.863	0.904	0.883	0.91	2.8

Table 23 Feature selection results of C project

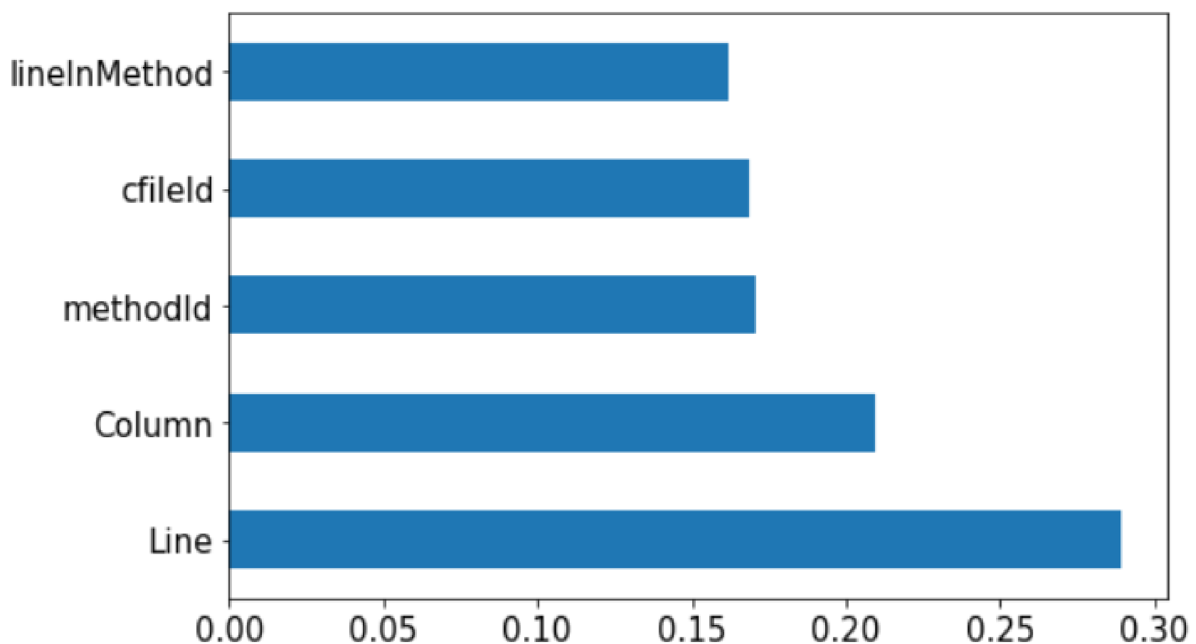

Figure 16 Feature importance of C project

5 Discussions

This section provides analysis of results of research questions. More precisely, it discusses the replication results of PMT, as well as its ability to be applied to a different technology.

The results of the paper [2] were successfully replicated. Under the cross-version application scenario all the metrics are very close to the ones from the paper for both Java projects. More precisely, F-measure values differ maximum by 0.05 and minimum by 0.001, ROC-AUC values differ maximum by 0.03 and minimum by 0.001. Prediction errors are very close too. The effect of intervals shown in Figure 2 and Figure 1 also demonstrates the accuracy of replication results by being so close to the results presented in the paper [2]. In the cross-project scenario I have used only one project for training and yet the results are surprisingly accurate. For example, a model build on apns project and evaluated on la4j has ROC-AUC value 0.858 and the same model from the paper [2] has 0.876. The difference is less than 0.03. The same is true for the model trained on la4j and evaluated on apns. For this model prediction errors are different by 1.32. All these findings support that the results of the original paper are authentic and it is possible to replicate them.

Several features can be collected for both Java and C projects. Since there are many differences between these projects the most similar cases are C project with naïve Random Forest and Java project with the single version with balanced Random Forest and no coverage mutants removed. According to the results of this case, C project ROC-AUC is 0.822 when divided into 80-20% of a train and test sets. With the same division, this result is closest to Java project v2 which has 0.803 value and it is farthest from v5 value 0.736. This proves that using common features gives similar results.

For the general evaluation of PMT on C and Java projects, we should look at the results of each project using all their available feature sets. As mentioned above C project only includes mutants with coverage so the results should be compared to Java project with removed no coverage mutants. This comparison is presented in **Table 21**. ROC-AUC values differ only by 0.002. Prediction errors are very low for both projects and all the other metrics values high. Based on this finding it is clear that PMT can be transferred to C language. This opens new possibilities for PMT. It can be applied to C projects and obtain good predictions of execution results. This reduces the cost of execution mutants and makes PMT useful for developers who need to obtain the execution results quickly in exchange for small accuracy loss.

This paper investigates the performance of PMT using Java and C projects. In the future, it can be also applied to other technologies like python for example. To see if characteristics of languages affect their feature sets or test-related features. If PMT can be more effective on another technology. In the future cross-language approach can also be investigated but it probably will not perform very well unless evaluated using large datasets.

Using only one version of C project is not enough proof that PMT will perform well on a different project. Several projects can be collected and investigated. Furthermore, in order to obtain better results on C project several versions of the same project can be used for analysis. In this case, I was given C project data and I was given only one version. So this limited my options.

Also, feature selection can be performed to detect what is the smallest feature set that can achieve similar results as it can be obtained using all the features. Less feature collection needs less time. This improves the efficiency of PMT. This research was able to find 5 features that can achieve such a result for C project. The same can be done for Java project. Even though it might be the result of the importance of the features. In C project most important features are related to location so using those features resulted in really good performance of the model. Because of the time issue, this approach will be studied in the future.

6 Conclusions

This thesis has focused on the replication of an existing paper [2]. Different parts of this paper were replicated and the obtained results support its findings. The same versions of programs and the same PIT tool were used. They helped greatly to make results as close to the original ones as possible. The replication of the cross-version approach was done in the same way: using the immediate previous version of a program for training and using the first version of a program for training. All the results indicate that the findings in the paper are genuine. The cross-project approach was done in a slightly different way. Instead of using 8 base projects for training only 1 project was used. Despite this, the results are very good.

This paper applied PMT to C project and was able to achieve very good performance. Predictions made by classification models is an accurate and fast way to generate the execution results of mutants. The results of C project are competitive to the results of Java project. In addition, this thesis shows that the same results can be obtained using only 5 features from the feature set of C project. Using only location-related features works well. It achieves more than 0.90 ROC-AUC value and Prediction Error is below 3%.

7 Acknowledgments

This thesis has been supported by Software Competence Center Hagenberg GmbH with funding by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET Center SCCH, grant no. FFG-865891.

References

[1] Jia, Y. and Harman, M., 2010. An analysis and survey of the development of mutation testing. IEEE transactions on software engineering, 37(5), pp.649-678.

[2] Zhang, J., Wang, Z., Zhang, L., Hao, D., Zang, L., Cheng, S. and Zhang, L., Predictive Mutation Testing.

[3] Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y. and Zhang, L., 2018. Predictive mutation testing. IEEE Transactions on Software Engineering.

[4] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y. and Harman, M., 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers* (Vol. 112, pp. 275-378). Elsevier.

[5] Coles, H., Laurent, T., Henard, C., Papadakis, M. and Ventresque, A., 2016, July. Pit: a practical mutation testing tool for java. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 449-452). ACM.

[6] Baker, R. and Habli, I., 2012. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, *39*(6), pp.787-805.

[7] Ramler, R., Wetzlmaier, T. and Klammer, C., 2017, April. An empirical study on the application of mutation testing for a safety-critical industrial software system. In Proceedings of the Symposium on Applied Computing (pp. 1401-1408). ACM.

[8] Voas, J.M., 1992. PIE: A dynamic failure-based technique. IEEE Transactions on software Engineering, 18(8), pp.717-727.

Appendix

License

Non-exclusive licence to reproduce thesis and make thesis public

I, Natia Doliashvili,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Predicting Survived and Killed Mutants

supervised by Dietmar Pfahl and Rudolf Ramler

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Natia Doliashvili

14/08/2019