PhD-FSTM-2022-064
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 17/05/2022 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Wei MA
Born on 23 October 1991 in Jiangsu (China)

NEXT GENERATION MUTATION TESTING: CONTINUOUS, PREDICTIVE AND ML-ENABLED

Abstract

Software has been an essential part of human life, and it substantially improves production and enriches our life. However, flaws in software can lead to tragedies, e.g. the failure of the Mariner 1 Spacecraft in 1962. At the moment, modern software systems are much different from what before. The issue gets even more severe since the complexity of software systems grows larger than before and Artificial Intelligence(AI) models are integrated into software (e.g., Tesla Deaths Report[1]).

Testing such modern artificial software systems is challenging. Due to new requirements, software systems evolve and frequently change, and artificial intelligence(AI) models have non-determination issues. The non-determination of AI models is related to many factors, e.g., optimization algorithms, numerical problems, the labelling threshold, data of the same object but under different collecting conditions or changing the backend libraries. We have witnessed many new testing techniques emerge to guarantee the trustworthiness of modern software systems. Coverage-based Testing is one early technique to test Deep Learning(DL) systems by analyzing the neuron values statistically, e.g., Neuron Coverage(NC) [203].

In recent years, Mutation Testing has drawn much attention. Coverage-based testing metrics can be misleading and easily be fooled by generating tests to satisfy test coverage requirements just by executing the code line. The test suite with one hundred percent coverage may detect no flaw in software. On the contrary, Mutation Testing is a robust approach to approximating the quality of a test suite. Mutation Testing is a technique based on detecting artificial defects from many crafted code perturbations (i.e., mutant) to assess and improve the quality of a test suite. The behaviour of a mutant is likely to be located on the border between correctness and non-correctness since the code perturbation is usually tiny. Through mutation testing, the border behaviour of the subject under test can be explored well, which leads to a high quality of software. It has been generalized to test software systems integrated with DL systems, e.g., image classification systems and autonomous driving systems.

However, the application of Mutation Testing encounters some obstacles. One main challenge is that Mutation Testing is resource-intensive. Large resource consumption makes it unskilled in modern software development because the code frequently evolves every day. This dissertation studies how to apply Mutation Testing for modern software systems, exploring and exploiting the usages and innovations of Mutation Testing encountering AI algorithms, i.e., ***how to employ Mutation Testing for modern software systems under test.*** AI algorithms can improve Mutation Testing for modern software systems, and at the same time, Mutation Testing can test modern software integrated with DL models well.

First, this dissertation adapts Mutation Testing to modern software development, Continuous Integration. Most software development teams currently employ Continuous Integration(CI) as the pipeline where the changes happen frequently. It is problematic to adopt Mutation Testing in Continuous Integration because of its expensive cost. At the same time, traditional Mutation Testing is not a good test metric for code changes as it is designed for the whole software. We adapt Mutation Testing to test these program changes by proposing ***commit-relevant mutants***. This type of mutant affects the changed program behaviours and represents the commit-relevant test requirements. We

[1]https://www.tesladeaths.com/

use the benchmarks from C and Java to validate our proposal. The experiment results indicate that ***commit-relevant mutants*** can effectively enhance code change testing.

Second, based on the aforementioned work, we introduce *MuDelta*, an AI approach that identifies ***commit-relevant mutants***, i.e., some mutants that interact with the code change. *MuDelta* uses manually-designed features that require expert knowledge. *MuDelta* leverages a combined scheme of static code characteristics as the data feature. Our evaluation results indicate that commit-based mutation testing is suitable and promising for evolving software systems.

Third, this dissertation proposes a new approach GraphCode2Vec to learn the general software code representation. Recent works utilize natural language models to embed the code into the vector representation. Code embedding is a keystone in the application of machine learning on several Software Engineering (SE) tasks. Its target is to extract universal features automatically. GraphCode2Vec considers program syntax and semantics simultaneously by combining code analysis and *Graph Neural Networks*(GNN). We evaluate our approach in the mutation testing task and three other tasks (method name prediction, solution classification, and overfitted patch classification). GraphCode2Vec is better or comparable to the state-of-the-art code embedding models. We also perform an ablation study and probing analysis to give insights into GraphCode2Vec.

Finally, this dissertation studies Mutation Testing to select test data for deep learning systems. Since deep learning systems play an essential role in different fields, the safety of DL systems takes centre stage. Such DL systems are much different from traditional software systems, and the existed testing techniques are not supportive of guaranteeing the reliability of the deep learning systems. It is well-known that DL systems usually require extensive data for learning. It is significant to select data for training and testing DL systems. A good dataset can help DL models have a good performance. There are several metrics to guide choosing data to test DL systems. We compare a set of test selection metrics for DL systems. Our results show that uncertainty-based metrics are competent in identifying misclassified data. These metrics also improve classification accuracy faster when retraining DL systems.

In summary, this dissertation shows the usage of Mutation Testing in the artificial intelligence era. The first, second and third contributions are on Mutation Testing helping modern software test in CI. The fourth contribution is a study on selecting training and testing data for DL systems. Mutation Testing is an excellent technique for testing modern software systems. At the same time, AI algorithms can alleviate the main challenges of Mutation Testing in practice by reducing the resource cost.

To my family

Acknowledgements

This dissertation would not have been possible without the help of my supervisors and my cooperators during my PhD career.

I want to express my deepest gratitude to my supervisor, Prof. Dr. Michail Papadakis, who provided me with an excellent opportunity to pursue my doctoral degree. I am grateful to the Head of the SerVal research group, Prof. Yves Le Tron, for the patience and valuable advice. I also want to give many thanks to Prof. Dr. Gilles Perrouin as a member of Comité D'encadrement de Thèse (CET) during my PhD life. I am also equally grateful to my colleagues, Dr. Thierry Titcheu Chekam, Dr. Ezekiel Soremekun and Dr. Maxime Cordy, who gave me a lot of help and support for my work.

I am also thankful to my cooperators for their efforts and feedback, including Dr. Jie M. Zhang, Prof. Dr. Xiaofei Xie, Dr. Mengjie Zhao, PhD Qiang Hu, and PhD Miloš Ojdanić, for their excellent discussions and suggestions.

I want to thank all the members of my PhD defence committee, including Prof. Dr. Yves Le Traon, Prof. Dr. Michail Papadakis, Prof. Dr. Lingming Zhang and Prof. Dr. Paolo Tonella. It is honourable to have them on my defence committee, and I express my deepest gratitude to them for their efforts to evaluate my PhD work. I want to thank all the colleagues from SerVal at SnT for the great interesting academic discussions.

Finally, I thank the unconditional support of my parents during my PhD study.

Wei Ma
University of Luxembourg
Feb. 2022

Contents

List of figures

List of tables

1 Introduction

In this chapter, we first introduce the concept and application of Mutation Testing for software testing and artificial intelligence testing. Then, we review the challenges of Mutation Testing. In the end, we summarize the contributions and organization of the dissertation.

Contents

1.1 Context

Over the past decades, the complexity of software systems has been increasing a lot, e.g., code size and interactions among the software modules. Software systems frequently evolve and have changes in their life cycle. Especially since Artificial Intelligence(AI) algorithms participate in modern software systems as one module [142], software systems have been much different from the traditional ones due to the added uncertainty and complexity. For example, autonomous driving systems may behave unexpectedly and inconsistently for the data from the same object but with different collection conditions. Another example is that the AI models which depend on the threshold may make the contradictory decision if changing the threshold configuration. Some researchers have named such software systems as intelligent software systems. It is questionable how to test them because the behaviours of intelligent software systems are not deterministic. Recent works have modified the exiting testing techniques or invented new ones to test intelligent software systems. Mutation Testing [166] draws intensive attention in the research field because of its non-deterministic property which Mutation Testing introduces the changes to software and affect the behavior of software. Mutation Testing is a robust methodology, simply saying, observing the behaviour-change of the program after introducing changes into the software.

1.1.1 Classical Mutation Testing

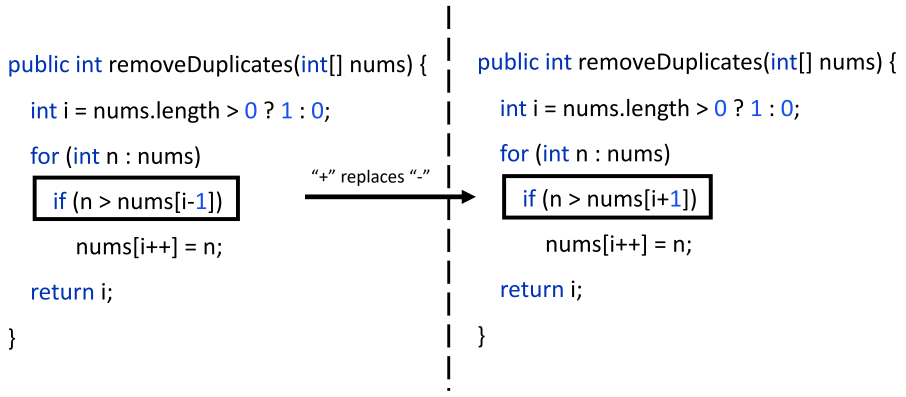
Test criteria are a group of metrics measuring the degree to which software systems have been tested [7]. They rely on the notion of test requirements, i.e., defining the test content. Depending on what test requirements a test suite covers, a test criterion defines a measurement that reflects the extent to which it tests the system w.r.t. to the intended behaviour. Test criteria have been used to drive different activities of the testing process, such as test generation [60] or test selection [233]. Test requirements are then utilized to determine which new tests are required or which tests are redundant. Test criteria can also be used to assess the thoroughness of a test suite, e.g. to decide if more effort should be devoted to testing or if sufficient confidence in the proper behaviour of the system has been gained. Test criteria are also used to assess other criteria [163], e.g., Mutation Testing.

Mutation analysis is a test criterion [119] that measures the capability of a test suite to detect artificial defects. Multiple versions of the program under test, called mutants, contain the artificial defects used as test requirements. The ability of the test suite to differentiate the program under test and these mutants is then evaluated. Mutants are systematically generated, following a set of replacement rules called mutation operators. Different mutation operators can be used in order to tailor the mutants created and thus the test requirements. This allows the tester to focus on different aspects of the test suite. Similarly, these operators can be applied only to specific parts of the program, should the tester only want to focus on those. Once mutants, i.e., test requirements, are created, the test suite is run against the program under test and the mutants in order to compare their behaviour. This behaviour is usually represented by the output of the program, captured by test or program assertions.

Mutation Testing lies on two principles [91], Competent Programmer Hypothesis (CPH) and Coupling Effect [46, 49]. CPH states that the developers are competent so that the program is close to the correct version. Coupling Effect assumes that the complex bugs are coupled with the simple bugs in the programs. Table 1.1 lists a set of mutant operators [153] defined in some mutation frameworks. Five operators in Table 1.1, ***ABS***, ***AOR*** , ***LCR*** , ***ROR*** and ***UOI***, are efficient enough to achieve almost all mutation coverage [153]. If some test fails for one specific mutant but passes for the original program, the mutant is killed. Otherwise, the mutant is a live mutant. The critical principle of Mutation Testing is trying to kill more mutants by augmenting the test suite. These mutants are regarded as the seeded defects, and killing more mutants means the test suite has the stronger ability to find the real bugs. Figure 1.1 shows one mutant example that uses the operator ***AOR***, replacing $-$ with $+$ in the framed expression.

Table 1.1: Mothra Mutant Operators [153]

Operator	Description
AAR	Array Reference for Array Reference Replacement
ABS	***Absolute Value Insertion***
ACR	Array Reference for Constant Replacement
AOR	***Arithmetic Operator Replacement***
ASR	Array Reference for Scalar Variable Replacement
CAR	Constant for Array Reference Replacement
CNR	Comparable Array Name Replacement
CRP	Constant Replacement
CSR	Constant for Scalar Variable Replacement
DER	DO statement End Replacement
DSA	DATA Statement Alterations
GLR	GOTO Label Replacement
LCR	***Logical Connector Replacement***
ROR	***Relational Operator Replacement***
RSR	RETURN Statement Replacement
SAN	Statement Analysis
SAR	Scalar variable for Array reference Replacement
SCR	Scalar for Constant Replacement
SDL	Statement Deletion
SRC	Source Constant Replacement
SVR	Scalar Variable Replacement
UOI	***Unary Operator Insertion***



```
public int removeDuplicates(int[] nums) {
  int i = nums.length > 0 ? 1 : 0;
  for (int n : nums)
    if (n > nums[i-1])
      nums[i++] = n;
  return i;
}
```

```
public int removeDuplicates(int[] nums) {
  int i = nums.length > 0 ? 1 : 0;
  for (int n : nums)
    if (n > nums[i+1])
      nums[i++] = n;
  return i;
}
```

Figure 1.1: One Example of Mutation Testing, applying AOR operator

Figure 1.2 demonstrates the sketch of Mutation Testing. There is one preliminary condition that the original program should pass all the test cases before starting Mutation Testing. We apply a set of mutant operators defined in Table 1.1 or custom-defined mutant operators to Software under Test(SUT) in the beginning step $\textcircled{1}$. This results in a group of mutants, namely different faulty versions of the programs, denoted as $\mathbf{M} = \{m_0, m_1, ..., m_n\}$, where m_i represents one mutant. We also need to detect and remove the equivalent mutants to reduce resource cost before executing the test suites denoted as the $\textcircled{2}$. Next, we run the test suite for the mutants and the original program at step $\textcircled{3}$ in Figure 1.2. If one test fails for some mutant, we mark the mutant ***killed***. Otherwise, we label the mutant ***live***. After executing all mutants, we compute ***Mutation Score***(MS) according to the execution matrix, defined by $MS = \frac{\text{Number of Killed Mutants}}{\text{Total Number of Mutants}-\text{Number of Equivalent Mutants}}$. We compare MS with the pre-defined threshold a at the step $\textcircled{4}$. Suppose MS is less than a. In that case, we try to generate new test cases to kill the remaining mutants, as illustrated by steps $\textcircled{5}$ and $\textcircled{6}$. Otherwise, we run the whole test suite for the original program after fixed. If the latest original program passes the test suite, we stop mutation testing as described $\textcircled{10}$. Otherwise, we fix the programs and continue the mutation testing loop denoted as (8) and (9).

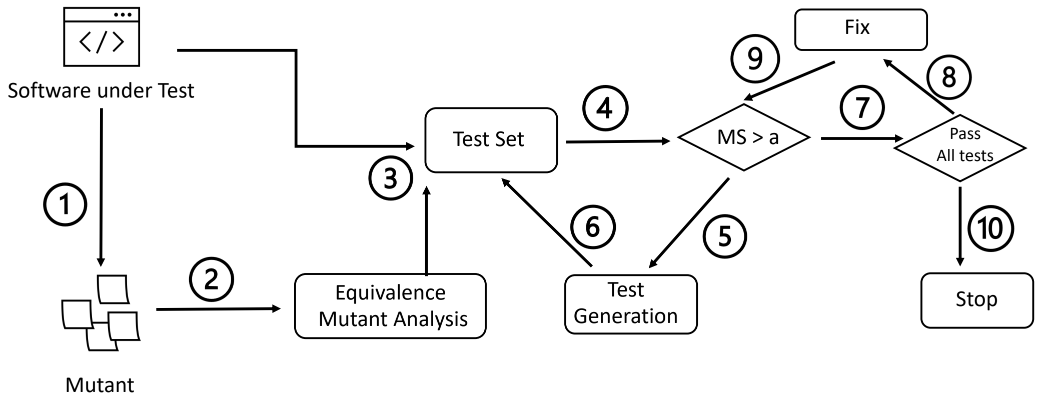


Figure 1.2: Workflow of Mutation Testing

Equivalent mutants analysis is necessary for Mutation Testing in Figure 1.2. Although they are syntactically different, equivalent mutants have the same semantic behaviours as the original program. When using mutation analysis to measure the thoroughness of a test suite, we should not take equivalent mutants into account, as even a perfect test suite will not kill them. Equivalent mutants have proven to be a significant challenge in Mutation Testing [166], as identifying them is an undecidable problem [25]. Ideally, we should only consider killable mutants for which there exists an input for which their behaviours are different from the original program's. Interestingly, many killable mutants are equivalent to others, introducing another problem, skew in the Mutation Score. A high mutation score does not necessarily mean the high quality of the test suite due to the issue. Kintis *et al.* [100] have shown this to be problematic and suggest getting rid of these "duplicated" mutants (mutants equivalent to others). Subsuming Mutation Score(sMS) is one alternative of Mutation Score based on the subsuming mutants. One subsuming mutant represents one set of mutants in which killing the subsuming mutant results in killing all mutants.

In the workflow of Mutation Testing in Figure 1.2, we have a stopping criterion $MS > a$ to judge if we quit Mutation Testing. Theoretically, the mutation score should be 100%, implying two conditions *1)* we exclude all equivalent mutants and *2)* kill all the remaining mutants. However, it is hard to satisfy the two conditions. In practice, the developers use one threshold to decide if stop the mutation testing. However, the correlation between the mutation score and the ability of test suites to find the bugs is questionable. For instance, achieving a mutation score of 80% does not mean that the test suite is necessary to reveal the faults in the program.

The mutation working flow is optimized based on Figure 1.2 in the different Mutation Testing frameworks. For instance, the mutation tools usually prefer executing tests early that kill more mutants , and mutate the bytecode to avoid compiling the code for each mutant. Pitest [40] is a popular mutation tool for Java programs and works at Java bytecode level. Mart [36] is a mutation tool for C programs like Pitest. The repository[1] introduces state of the art tools for Mutation Testing.

1.1.2 Continuous Integration

Continuous Integration(CI)[24] is one methodology of the software development process, i.e., a process to produce a software product. CI defines a set of development principles to merge all local working repositories to a central shared repository a few times a day. Although the terminology, Continuous

[1]https://github.com/theofidry/awesome-mutation-testing

Integration(CI), has existed since the 1990s, CI is still one of the best practices during the software evolution and is widely used by many developing teams. As shown in Figure 1.3, the CI loop consists of four(4) steps. In the beginning, the developers clone the base code locally. When the developers make the code changes, they can build and test the code locally before submitting these changes to the central code repository. After the developers submit the changes to the shared repository, the CI server builds the project and runs the tests. The developers can release their software after passing all tests. Next, the developing plan updates with the new issues or requirements of the software. CI can boost software development with many benefits. The key benefit is that CI can detect the code errors early before the release date. The process of CI is transparent and visualized, which is good for the team communication. Since CI includes multiple changes a day, keeping a high-quality test suite is crucial.

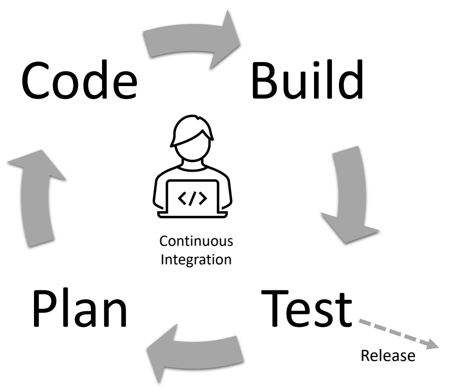


Figure 1.3: Continuous Integration

1.1.3 Artificial Intelligence Software Testing

As Artificial Intelligence(AI) succeeds in many fields, e.g., Computer Vision and Natural Language Processing, more and more software systems contain AI components such as the autonomous vehicles. Testing these AI-based systems is vital and has become a hot research topic. There are three(3) levels of AI testing, data testing, model testing and code testing. Data testing improves the data quality to help the AI model learning. False data labels in the training data will confuse the model and the incorrect test data will bias the test results. Curtis G. Northcutt *et al.* [152] have found many label errors in the public datasets, e.g., ImageNet and Amazon Reviews. Model testing focus on checking if the models work well. Jie M. Zhang *et al.* [238] comprehensively study Machine Learning Testing. The testers should focus on seven(7) properties of AI models, Correctness, Model Relevance, Robustness & Security, Efficiency, Fairness, Interpretability and Privacy. Code testing is utilized to check for bugs in the program code and machine learning libraries, e.g., dividing zero and wrong tensor shape.

1.2 Challenges of Mutation Testing and Artificial Intelligence Testing

In Mutation Testing, we create a large number of mutants, but not all of them are valuable to reveal the errors of the program and it is costly if we use all mutants. Significantly, modern software development requires frequently code change, which means the test activity is also frequent, as shown in Figure 1.3. Continuous Integration(CI) is a development pipeline where the developers often contribute code into a shared project repository. The developers change the code according to the plan, e.g., fixing the bugs in the bug list. Usually, the developers need to give new test cases for the changes. The changes should pass the tests locally before merging into the shared repository. Then, CI builds and tests the program automatically on the side of the shared central repository. High cost makes Mutation Testing difficult in the current software development because the CI loop in Figure 1.3 can happen several times a day. The researchers have proposed selective mutation testing [149] to improve the efficiency of Mutation Testing by reducing the number of mutants. However, these reduction techniques do not consider the behaviour difference caused by the code change. This dissertation studies the issue and proposes a machine learning approach to select mutants as the test requirements for the code change, as demonstrated in Chapter 3 and Chapter 4. When applying machine learning algorithms in Software Engineering, we often need to design the data feature manually depending on the domain knowledge. This way is not general because we need to carefully design the feature for each task. In Chapter 5, this dissertation demonstrates a general approach to extracting the code feature and validating it in multiple software engineering tasks, including the mutant prediction task.

Artificial Intelligence algorithms usually require a large number of data for training and testing, especially Deep Learning(DL) models. Selecting informative data to train and test DL models is vital and meaningful to estimate the performance of the DL models reliably. Mutation Testing creates different variants of the models to select the data based on the uncertainty. This dissertation shows a study about data selection in Chapter 6 to test and retrain the DL models.

1.3 Contribution

We gather up the contributions of the dissertation. Overall, the contributions are about Mutation Testing in Continuous Integration, Code Representation for Software Engineering tasks, and Mutation Testing for AI systems

Commit Aware Mutation Testing adapted to Continuous Integration [135] (accepted by ICSME 2020) In Continuous Integration, developers want to know how well they have tested their changes. Unfortunately, in these cases, the use of mutation testing is suboptimal since mutants affect the entire set of program behaviours and not the changed ones. Thus, the extent to which mutation testing can be used to test committed changes is questionable. To deal with this issue, we define commit-relevant mutants; a set of mutants that affect the changed program behaviours and represent the commit-relevant test requirements. We identify such mutants in a controlled way, and check their relationship with traditional mutation score (score based on the entire set of mutants or on the mutants located on the commits). We conduct experiments in both C and Java, using 83 commits, 2,253,610 mutants from 25 projects. Our findings reveal that there is a relatively weak correlation (Kendall/Pearson 0.15-0.4) between the sought (commit-relevant) and traditional mutation scores, indicating the need for a commit-aware test assessment metric. Our analysis also shows that traditional mutation is far from the envisioned case as it loses approximately 50%-60% of the commit-relevant mutants when analysing 5-25 mutants. More importantly, our results demonstrate that traditional mutation has approximately 30% lower chances of revealing commit-introducing faults than commit-aware mutation testing.

A machine learning approach of Mutation Testing for the commit change [134] (accepted by ICSE 2021) To effectively test program changes using mutation testing, one needs to use mutants that are relevant to the altered program behaviours. We introduce *MuDelta*, an approach that identifies commit-relevant mutants; mutants that affect and are affected by the changed program behaviours. Our approach uses machine learning applied on a combined scheme of graph and vector-based representations of static code features. Our results, from 50 commits in 21 Coreutils programs, demonstrate a strong prediction ability of our approach; yielding 0.80 (ROC) and 0.50 (PR-Curve) AUC values with 0.63 and 0.32 precision and recall values. These predictions are significantly higher than random guesses, 0.20 (PR-Curve) AUC, 0.21 and 0.21 precision and recall, and subsequently lead to strong relevant tests that kill 45% more relevant mutants than randomly sampled mutants (either sampled from those residing on the changed component(s) or from the changed lines). Our results also show that *MuDelta* selects mutants with 27% higher fault revealing ability in fault introducing commits. Taken together, our results corroborate the conclusion that commit-based mutation testing is suitable and promising for evolving software.

A novel approach to learn the universal code representation [138] (accepted by MSR 2022) Code embedding is a keystone in the application of machine learning on several Software Engineering (SE) tasks. To effectively support a plethora of SE tasks, the embedding needs to capture program syntax and semantics in a *generic* way. To this end, we propose the *first self-supervised pre-training* approach (called GraphCode2Vec) which produces task-agnostic embedding of lexical and program dependence features. GraphCode2Vec achieves this via a synergistic combination of *code analysis* and *Graph Neural Networks*. GraphCode2Vec is *generic*, it *allows pre-training*, and it is *applicable to several SE downstream tasks*. We evaluate the effectiveness of GraphCode2Vec on four (4) tasks (method name prediction, solution classification, mutation testing and overfitted patch classification), and compare it with four (4) similarly *generic* code embedding baselines (Code2Seq, Code2Vec, CodeBERT, GraphCodeBERT) and 7 *task-specific*, learning-based methods. In particular, GraphCode2Vec is more effective than both generic and task-specific learning-based baselines. It is also complementary and comparable to GraphCodeBERT (a larger and more complex model). We also demonstrate through a probing and ablation study that GraphCode2Vec learns lexical and program dependence features and that self-supervised pre-training improves effectiveness.

An empirical study on test selection of Deep Learning systems based on model uncertainty metrics [137] (accepted by TOSEM) Testing of deep learning models is challenging due to the excessive number and complexity of the computations involved. As a result, test data selection is performed manually and in an ad hoc way. This raises the question of how we can automatically select candidate data to test deep learning models. Recent research has focused on defining metrics to measure the thoroughness of a test suite and to rely on such metrics to guide the generation of new tests. However, the problem of selecting/prioritising test inputs (e.g. to be labelled manually by humans) remains open. In this work, we perform an in-depth empirical comparison of a set of test selection metrics based on the notion of model uncertainty (model confidence on specific inputs). It is achieved by creating different versions of the model to measure uncertainty. Intuitively, the more uncertain we are about a candidate sample, the more likely it is that this sample triggers a misclassification. Similarly, we hypothesise that the samples for which we are the most uncertain, are the most informative and should be used in priority to improve the model by retraining. We evaluate these metrics on 5 models and 3 widely-used image classification problems involving real and artificial (adversarial) data produced by 5 generation algorithms. We show that uncertainty-based metrics have a strong ability to identify misclassified inputs, being 3 times stronger than surprise adequacy and outperforming coverage related metrics. We also show that these metrics lead to faster improvement in classification accuracy during retraining: up to 2 times faster than random selection and other state-of-the-art metrics, on all models we considered.

1.4 Organization of the Dissertation

In the remaining of dissertation is organized, Chapter 2 introduces the background and the related work of this dissertation. Chapter 3 presents the adaptation of Mutation Testing for the code change in the evolving software systems. Chapter 4 presents an AI approach for the relevant mutant, *MuDelta*, and shows how it is useful for Mutation Testing in Continuous Integration to test the code behavior change. Chapter 5 states our code embedding approach GraphCode2Vec for the general Software Engineering tasks, including the mutant prediction task. Chapter 6 presents our work on how to select the test data for DL systems.The finial Chapter 7 summarizes this dissertation and future work.

2 Background

The section introduces the related work about this dissertation.

Contents

2.1 State-of-the-art Mutation Testing

Mutation Testing Cost Reduction. Historically, different techniques have been proposed to reduce the mutation testing cost. C. Ferrari *et al.* [44] and Pizzoleto *et al.* [178] empirically study the cost reduction techniques for Mutation Testing. Offutt *et al.* [155] categorize these techniques into three groups, *do fewer*, *do smarter*, and *do faster*. The *do fewer* group targets fewer mutant programs within the tolerance loss. Mutant selection is one primary technique in the group. A straightforward way is to select random mutants. J. Zhang *et al.* [237] study the scalability of selective mutation testing, and later they predicted mutant testing results without execution [235]. Another method is to combine different mutants to reduce the execution time, called Higher order mutation. The *do smarter* group seeks to distribute the computational cost over a few machines or factor the cost over several executions by keeping state information between runs or avoiding complete execution. Both techniques of weak mutation and parallel executions are representative approaches of *do smarter*. Weak mutation checks the internal state instead of the final state of the program. Parallel execution launches multiple processors to execute Mutation Testing by reducing the total time. The *do faster* group focuses on ways of generation and execution of each mutant program as quickly as possible. One from the *do faster* category is mutant schemata which includes all mutants in one meta program.

C. Fabiano *et al.* [44] indicate that *do fewer* and *do smarter* approaches draw more attention than *do faster* at the moment. It should be noted that Mutation Testing has been employed more and more in the industry, e.g., Goolge [175]. Recently, AI has been applied in Mutation Testing to generate more mutants like the bugs [207]. A. Garg *et al.* [64] uses the natural language model to predict the subsuming mutants. T.C. Thierry *et al.* [205] presents a new machine learning approach to select fault revealing mutants.

Regression Mutation Testing. Applying mutation during regression testing has long been proposed. In particular, Cachia *et al.* [28] proposed applying change-based mutation testing by considering only the mutants located on the altered code. Zhang *et al.* [240] proposed Regression Mutation Testing, a technique that speeds up mutant execution on evolving systems by incrementally calculating the mutation score (and mutant status, killed/live). As such, they assume that testers should use the entire set of mutants when testing evolving software systems.

Existing mutation testing tools, such as Pitest [41], include some form of incremental analysis in order to calculate the mutation score (and mutant status, killed/live) of the entire systems or class under test. Petrovic and Ivankovic [174] use mutation within the code review phase, by randomly picking some mutants located on the altered code areas.

Test Generation and Prioritization for Mutation Testing. One main challenge in Mutation Testing is to generate test cases to kill the mutants. The popular test generation tool cannot efficiently create tests to kill the mutants because they are guided by coverage test metrics, e.g., EvoSuite[1]. And manual test generation needs much effort. Symbolic execution has been used to solve the problem[35, 79, 161, 241]. Symbolic execution symbolizes the inputs of the program and executes the program abstractly. Symbolic execution can target a specific mutant by instrument, and thus it can effectively generate test cases for the mutants. However, symbolic execution has some limitations to impede its application, e.g., execution path explosion and the symbolization of arrays. It is still far away to automatically generate test cases to kill the mutants. Mutation Testing usually results in the growth of the number of test cases. We need to run all test cases until one test kills the mutant. Therefore, the text execution order plays an important role in the mutation testing cost. Faster Mutation Testing (FaMT) [239] prioritizes and reduce tests to quickly kill the mutants.

[1]https://www.evosuite.org/

2.2 Deep Learning Testing

Testing of learning systems is typically performed by selecting a dedicated test set randomly from available labelled data [224]. When an explicit test dataset is not available, one can rely on cross-validation [105] to use part of the training set to anticipate how well the learning model will generalize to new data. These established procedures, however, often fail to cover many errors. For instance, research work in adversarial learning has shown that applying minor perturbations to data can make models give a wrong answer [71]. Nowadays, those adversarial samples remain hard to detect and bypass many state-of-the-art detection methods [31]. In order to achieve better testing, multiple approaches have been proposed in the recent years. We distinguish four categories of contributions: (i) metrics for measuring the coverage/thoroughness of a test set; (ii) generation of artificial inputs; (iii) metrics for selecting test data; (iv) detection of adversarial samples.

DeepXplore, proposed by Pei et al. [169], comprises both a coverage metric and a new input generation method. It introduces neuron coverage as the first white-box metric to assess how well a test set covers the decision logic of DL models. Leaning on this criterion, DeepXplore generates artificial inputs by solving a joint optimization problem with two objectives: maximizing the behavioural differences between multiple models and maximizing the number of activated neurons. Pei et al. report that DeepXplore is effective at revealing errors (misclassifications) that were undetected by conventional ML evaluation methods. Also, retraining with additional data generated by DeepXplore increases the accuracy of the models. In some models, the increase is superior (1 to 3%) to an increase obtained by retraining with data generated by some adversarial technique [71]. Pei et al. also show that randomly-selected test data and adversarial data achieve smaller neuron coverage than data generated by DeepXplore. While they assume that more neuron coverage leads to better testing, future research showed that this metric is inadequate [97, 131].

In a follow-up paper, Tian et al. [204] propose DeepTest as another method to generate new inputs for autonomous driving DL models. They leverage metamorphic relations that hold in this specific context. Like DeepXplore, DeepTest utilizes the assumption that maximising neuron coverage leads to more challenging test data. The authors show that different image transformations lead to different neuron coverage values and infer that neuron coverage is an adequate metric to drive the generation of challenging test data. However, this claim was not supported by empirical evidence.

A related method was proposed by [133]. It works under the assumption that there is a recurring defect in the DNN model, such that inputs from one particular class are often misclassified. The method is based on differential analysis to identify features/neurons responsible for this defect, so as to fix the model. On the contrary, the uncertainty metrics we propose to use are independent of the particular class of the inputs and are lightweight (they do not require more expensive computations/analyses).

DeepGauge [131] and DeepMutation [132] are two test coverage metrics proposed by Ma et al. With DeepGauge, they push further the idea that higher coverage of the structure of DL models is a good indicator of the effectiveness of test data. They show, however, that the basic neuron coverage proposed previously is unable to differentiate adversarial attacks from legit test data, which tends to indicate the inadequacy of this metric. As a result, they propose alternative criteria with different granularities, i.e. at the neuron level and the layer level. Their experiments reveal that replacing original test inputs by adversarial ones increases the coverage of the model wrt. DeepGauge's criteria. However, they did not assess the capability of their criteria to prioritize test samples likely to trigger misclassifications.

Similarly, DeepMutation leverages the mutation score used in traditional mutation testing to DL models. From a given model, it generates multiple mutant models by applying different operators such as, e.g., neuron switch or layer removal. Then, they define the mutation score of a test input as the number of mutants that it killed (i.e. those that yield a different classification output for the test input than the original model). Thus, the mutation score assesses how sensitive the model is wrt. the test inputs rather than how these cover the neurons of the network. DeepMutation++ [86]

extends DeepMutation to Recurrent Neural Network(RNN) with more mutation operators as a public tool[2]. DeepCrime [88] as the state-of-the-art work leverages the mutant operators to investigate the probability to mimic the effects of actual DL errors. The work defines 35 mutation operators and find these mutant operators effectively are killable and vital. DeepGini [56] prioritizes the test data based on the statistical properties of the deep learning models. Uncertainty-wizard [221] is a tool that supports quantify such uncertainty and confidence in artificial neural networks.

Nevertheless, both DeepGauge and DeepMutation measure the coverage/thoroughness of a test set but do not aim at selecting individual inputs to undergo labelling. Moreover, a recent study [98] has shown that neuron coverage criteria do not necessarily increase when more misclassified/surprising inputs are added.

Later on, Ma et al. [130] proposed DeepCT, a test coverage metric suggesting that within a given layer, all tuples of neurons should be covered by at least one test input. They also propose an algorithm to generate artificial inputs to cover as many t-wise interactions as possible. They show, first, that random test selection cannot cover a large part ($> 65\%$) of the 2-wise neuron interactions. Second, they show that retraining on the inputs generated by their algorithm allows the detection of up to 10% of adversarial samples that could not be detected by retraining on randomly selected inputs. An alternative proposed by Xiaofei Xie et al. [227] is DeepHunter, a fuzzing-based test generation algorithm to hunt defects in DL models. The fuzzing is guided by the coverage metrics defined in DeepXplore [169] and DeepGauge [131]. Their evaluation shows that the fuzzing algorithm manages to increase the intended coverage metrics. Both DeepCT and DeepHunter focus on generating artificial inputs and are not directed towards the selection of available challenging data for testing and retraining.

Most recently, Kim et al. [98] proposed metrics for test coverage *and* test selection. They highlight the fact that coverage criteria fail to discriminate the added value of *individual* test inputs and are therefore impractical for test selection. They argued that test criteria should guide the selection of individual inputs and eventually help improving the DL models' performance. As a consequence, they propose *surprise adequacy* as a metric that measures how surprised the model is when confronted with a new input. More precisely, the degree of surprise measures the dissimilarity of the neurons' activation values when confronted to this new input wrt. the neurons' activation values when confronted to the training data. They hypothesise that a set of test inputs is preferable for both testing and retraining when it covers a diverse range of surprise values. In other words, a good test set should range from very similar to very different inputs to those observed during training. Kim et al. show experimentally that (a) surprise coverage is sensitive to adversarial samples and (b) retraining on such samples yields better improvements in accuracy. In Chapter 6, we show that model uncertainty is more effective at triggering misclassifications and improving the accuracy of the models than input diversity. Nevertheless, surprise adequacy is complementary to our work since it aims for a diversity of surprise degrees and thus better applies to models that are not yet well-trained, while uncertainty metrics aim at reinforcing well-trained models against inputs that remain challenging.

Uncertainty of DL systems was theoretically studied by a number of authors. Gal and Ghahramani [62] proved that the variance of the softmax function resulting from neuron dropout is a good estimate for model uncertainty. Kendall and Gal [96] propose a model to capture jointly *aleatoric* uncertainty (originating from noise inherent to the observations) and *epistemic* uncertainty (induced by the fact that the model is not trained on all possible data). The latter type is what is commonly referred to as *model uncertainty*, which can be captured by dropout variance [96].

In [74], Smith and Gal provide evidence that uncertainty metrics can be used to detect adversarial samples, although this does not hold for data that are far away from the training set. Compared with this line of work, the contribution of the dissertation is to study uncertainty from a new perspective and how well it can achieve the purpose of selecting inputs that trigger misclassifications. Akin notions were used by Feinman et al. [55] to detect adversarial samples. This method uses kernel

[2]https://sites.google.com/view/deepmutationpp/home

density estimation of neuron activation (similar to likelihood-based surprise adequacy [98]) and Bayesian uncertainty based on dropout variance (similar to what we use here). Wang et al. [216] proposed computing how much the labels predicted by a model change when (after training) this model is slightly altered. They showed that adversarial inputs are more likely to increase the label change rate. A purely Bayesian generative adversarial method is proposed in [186], where the adversarial sample generator and the discriminator are Bayesian neural networks trained with stochastic gradient Hamiltonian Monte Carlo sampling. Specifically, the discriminator network is capable of efficiently detecting adversarial samples because of the competition-based structure, which forces learning to be a repeated contest between the generator and the discriminator. Another detection method based on uncertainty is that of Sheikholeslami and Giannakis [194], which promotes scalability by measuring uncertainty on sampled hidden layers. Pinder's master thesis [176] reports other experiments demonstrating that adversarial images yield a significantly greater uncertainty than original images. In other settings, though, Grosse et al. [74] show that there exist adversarial examples which do not affect the uncertainty of the model.

Overall, the aforementioned studies aim at detecting adversarial examples. Compared to them, our goal is to select examples that are most likely to be misclassified, be these real or adversarial (studied separately and together). Another key difference is that we consider both well-classified and misclassified adversarial examples. Doing so, Chapter 6 demonstrates that the metrics are even more sensitive to the noise introduced by adversarial algorithms than they are to the classification results, which is in line with the results of [74, 176, 194, 216]. Also, we consider a broad range of metrics, including (but not limited to) multiple metrics that approximate uncertainty. Specifically, compared to [216], the dropout variance we use is more fine-grained than the label change rate as it is computed over classification probabilities.

2.3 Natural Language Processing

Natural Language Processing(NLP) is a research field that allows the computer program to understand human language. It has many applications, e.g., language translation and search engines. Recently, NLP techniques are used in Software Engineering tasks, e.g., code clone detection and automatic program repair.

Word Embedding Word embedding is one elementary technique that encodes the word text into a real-value vector semantic representation in Natural Language Process(NLP). The words that have similar meanings should be close in the latent space. A straightforward word-embedding technique is one-hot-encoding. However, this method will break the similarity between two similar meaning words, and the vector dimension is huge due to the many words. Baroni *et al.* [13], Pennington *et al.* [171] and Li *et al.* [120] categorize word embeddings into two types, i.e., prediction-based approaches and count-based approaches. Prediction-based approaches utilize the neural language models (NNLMs [16]) to learn an embedding layer. Word2Vec [146] is one prevalent method from this group, including two different learning models, CBOW(continuous bags of words) model and continuous Skip-Gram Model. Count-based approaches use the statistical information from the corpus to encode the word into a vector, e.g., using word frequency. GloVe [171] is one famous count-based method .

Pretraining in NLP Pretraining is one of the central topics in current Natural Language Process(NLP) research: A deep neural network like LSTM [84] or Transformer [211] is trained on a large text corpus using self-supervised learning without any human supervision (e.g., human-annotated datasets). The representative pretrained models like ELMo [173] and BERT [47] provide high-quality generic representations containing rich syntactic [70] and semantic [243] information about languages. These representations are highly transferable, significantly benefiting NLP systems in a wide range of downstream tasks such as sentiment analysis [199] and natural language inference [223] using straightforward transfer learning [47, 85, 122, 244]. Transferability is also an essential requirement in our approach as we aim to build code models that can perform various tasks.

Interpretability in Language Processing The impressive performance of BERT stimulates loads of work trying to interpret and understand these newly invented large-scale blackbox models. *Probing* [42, 183, 243] is one of the most prominent techniques that has been widely leveraged for interpretability. Probing aims at diagnosing which types of regularities are encoded in an input vector representation extracted from data. The hypothesis of probing is that if a *simple* classifier built upon the representations can solve a task sufficiently well, then the representations should contain informative features about the task already. For example, Goldberg [70] probes syntactic information encoded in BERT; Zhao *et al.* [243] demonstrate how BERT contextualizes words; Vulić *et al.* [215] assess lexical semantics in BERT using probing; Lin *et al.* [124] probe the numerical commonsense of BERT; Goodwin *et al.* [72] probe for linguistic systematicity.

NLP in Big Code Researchers notice that code data have common properties with the text, e.g., function name and API documents. M. Allamanis *et al.* [2] propose the naturalness hypothesis,

Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.

The hypothesis stands as coding is an activity of communication, and code corpora have rich patterns like natural language. NLP techniques are widely used in code analysis, e.g., code search [30], code generation and completion [220]. However, J. Mosel *et al.* [214] indicate that the models trained with human natural language have trouble understanding Software Engineering(SE) domain terminology. Models trained with SE domain data are good at understanding SE context. One advanced topic is code embedding, like word embedding. The target of code representation is to learn the distribution of code properties in the latent space. Lots of methods on code embedding have been proposed recently, e.g., Code2Vec [5], Code2Seq [4] and CodeBert [57]. These code representations are applied in a wide range of software engineering downstream tasks, e.g., flaky test detection [54].

3 Commit-Aware Mutation Testing

In Continuous Integration, developers want to know how well they have tested their changes. Unfortunately, in these cases, the use of mutation testing is suboptimal since mutants affect the entire set of program behaviours and not the changed ones. Thus, the extent to which mutation testing can be used to test committed changes is questionable. To deal with this issue, we define commit-relevant mutants; a set of mutants that affect the changed program behaviours and represent the commit-relevant test requirements. We identify such mutants in a controlled way, and check their relationship with traditional mutation score (score based on the entire set of mutants or on the mutants located on the commits). We conduct experiments in both C and Java, using 83 commits, 2,253,610 mutants from 25 projects. Our findings reveal that there is a relatively weak correlation (Kendall/Pearson 0.15-0.4) between the sought (commit-relevant) and traditional mutation scores, indicating the need for a commit-aware test assessment metric. Our analysis also shows that traditional mutation is far from the envisioned case as it loses approximately 50%-60% of the commit-relevant mutants when analysing 5-25 mutants. More importantly, our results demonstrate that traditional mutation has approximately 30% lower chances of revealing commit-introducing faults than commit-aware mutation testing.

Contents

3.1 Introduction

Modern software development involves the continuous submission and integration of code modifications from many developers into a common codebase [59]. This continuous development is performed by automatic procedures that build and test the software products. Automated testing is used to establish confidence that the committed code behaves correctly, while at the same time it does not break any of the previously developed program functionalities.

When developers commit their code, they are interested in testing the delta of behaviours between their pre- and post-commit versions in order to discover issues and side effects caused by their changes. Thus, developers are interested in knowing how well they have tested the program behaviours affected by their changes. To this end, many studies suggest using mutation testing (or other test adequacy criteria) to drive test generation, or to assess test thoroughness on the evolving software [174, 240].

Mutation testing has long been established as one of the strongest test criteria [166]. It operates by measuring the extent to which test suites can distinguish the behaviour of the original program from that of some slightly altered (syntactically altered) program versions, which are called mutants. Testers can use mutants to design test cases [46] or to measure test suites' thoroughness [9].

Previous research has shown that mutation testing leads to fault revelation [7, 38] and can be used for test assessment as it effectively quantifies the test suites' strengths [9]. Unfortunately, traditional mutation testing aim at testing the entire codebase, rather than specific program changes/commits as would naturally be requested by developers.

There are many studies aiming at making the mutation score metric accurate either by using specific mutant types [154], or by detecting equivalent mutants [100, 140], i.e., mutants that cannot be killed by any test case because they are semantically equivalent to the original program, or by eliminating redundant mutants [112, 163], i.e., mutants that are killed "collaterally" whenever other mutants are killed (subsumed by the subsuming mutants). Yet, little research has focused on measuring the effectiveness of test suites with respect to particular program changes or commits.

To form a commit-aware mutation criterion, it is necessary to identify mutants capturing the altered program behaviours, i.e., mutants interacting with the changed program behaviours, representing the sought commit-relevant test requirements. These mutants can then be used to judge whether test suites are adequate and, if not, to provide guidance in improving test suites (by creating tests that kill commit-relevant mutants).

One may assume that, since mutation score reflects test thoroughness (of the whole system, component or class under test), it also reflects, or at least the score delta between versions reflects, the extent to which changes are tested. Someone else may consider that the changed program parts can be tested by mutating only the modified code, assuming that mutant locations reflect their utility and relevance.

These assumptions may appear intuitive but unfortunately they do not hold. This is because of the large numbers of irrelevant (to the committed changes) mutants and the many relevant ones that are spread on the entire codebase. Since these mutants are unknown to mutation testers, they hinder their ability to distinguish between relevant and irrelevant mutant kills. Mutating only the modified code parts yields better results, but still, it is insufficient to cover all possible interactions between the unmodified and changed code.

We argue that covering all interactions between unmodified and modified code is particularly important because problematic regression issues arise from such unforeseen interactions [22, 188]. This is demonstrated by our results that show the majority of the altered program behaviours to be captured by mutants located on unmodified code parts.

In our analysis, we also considered the potential gains and losses of either using the entire set of mutants or those mutants that are located on the committed code. Obviously, by killing all the mutants, one achieves killing all the commit-relevant ones. However, this comes with the cost of analysing more mutants, and generating more test cases than needed. Perhaps more importantly, the killing of mutants irrelevant to the commit inflates the mutation score, hindering its ability to reflect test thoroughness w.r.t. to committed code. Similarly, by killing all the mutants located on the committed code, one fails to kill a significant number of commit-relevant mutants, loosing significant test effectiveness.

Interestingly, our results reveal that there is a relatively weak correlation between the sought commit-aware and traditional mutation scores, indicating the need for a commit-relevant test assessment metric. Our analysis also shows that when using mutants for test suite improvement [167] (by adding tests that kill mutants), traditional mutant selection is very far from the envisioned case, as it loses approximately 50%-60% of the commit-relevant mutants (when analysing 5-25 mutants). Perhaps more importantly, our results demonstrate that commit-relevant mutants have 30% more chances to reveal faults (real faults) than traditional mutation when analysing the same number of mutants (putting approximately the same amount of effort).

Overall, our contribution is the definition of the commit-relevant mutants and the envisioned commit-relevant mutation-based test assessment. We motivate this by providing evidence that mutation testing performed with the entire set of mutants or with the mutants located on the committed code is insufficient to assess test thoroughness or to provide cost-effective guidance to adequately test particular program changes.

Taken together, our key contributions can be summarised by the following points:

- We define the commit-relevant mutation testing, which is based on the notion of commit-relevant mutants, i.e., mutants capturing the interactions between modified and unmodified code.
- We investigate the extent to which mutation-based test assessment metrics such as a) the mutation score (score that includes the entire set of mutants), b) the delta of mutation scores between pre- and post-commit, c) the mutation score of mutants located on the committed code, correlate with the sought commit-relevant mutation score. Our results show that all three metrics have relatively weak correlations (less than 0.4), indicating the need for a commit-relevant test assessment metric.
- We further examine the potential guidance given by commit-relevant mutation testing by comparing the gains and losses of strategies that use the entire set of mutants, the mutants located on the committed code and the commit-relevant mutants. Our findings suggest that commit-relevant mutants have 30% higher fault revelation ability (wrt real commit-introduced faults) than the other strategies when analysing the same number of mutants.

3.2 Commit-Relevant Mutants

Informally, a commit-aware test criterion should reflect the extent to which test suites have tested the altered program behaviours. This means that test suites should be capable of testing and making observable any interaction between the altered code and the rest of the program. We argue that mutants can capture such interactions by considering both the behavioural effects of the altered code on mutants' behaviour and visa versa. This means that mutants are relevant to a commit when their behaviour is changed by the regression changes. Indeed, changed behaviour indicates a coupling between mutants and regressions, suggesting relevance.

Precisely, the regression changes interact with a mutant when the program version that includes both the regression changes and the mutant behaves differently from:

1. the version that includes only the mutant (mutant in the pre-commit version).
2. the version that includes only the regression changes (post-commit version).

This situation is depicted in Figure 3.1.

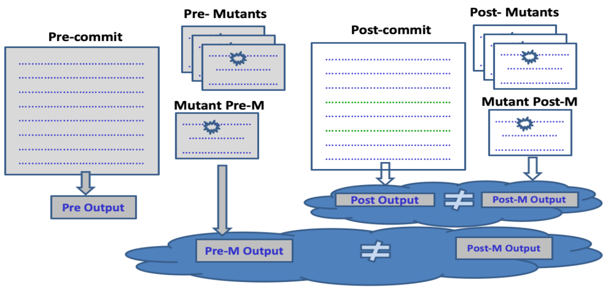
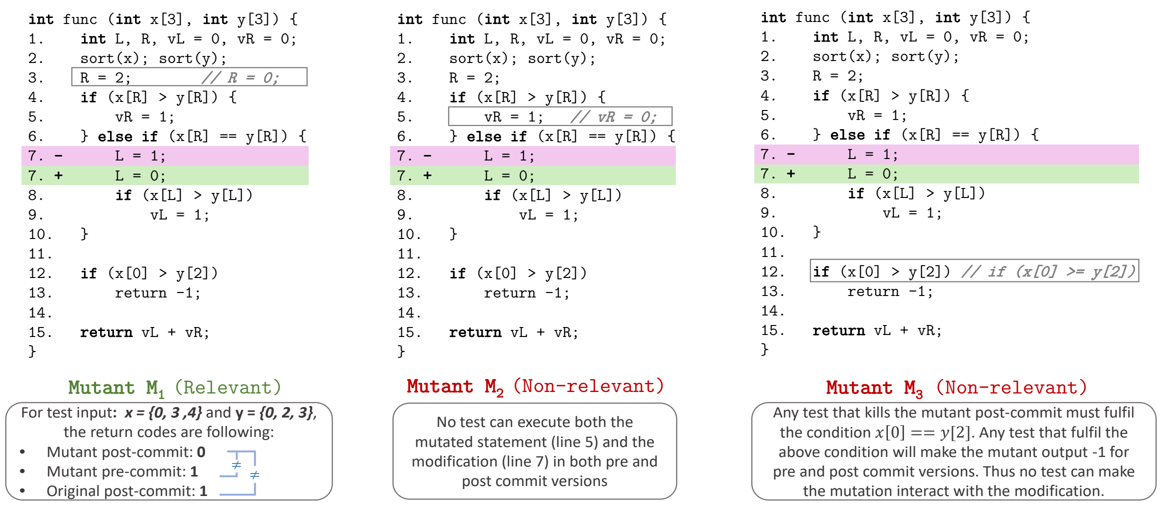


Figure 3.1: A mutant is relevant if it impacts the behaviour of the committed code and the committed code impacts the behaviour of the mutant.

3.2.1 Demonstrating Example



```
int func (int x[3], int y[3]) {
1.    int L, R, vL = 0, vR = 0;
2.    sort(x); sort(y);
3.    R = 2;            // R = 0;
4.    if (x[R] > y[R]) {
5.        vR = 1;
6.    } else if (x[R] == y[R]) {
7. -      L = 1;
7. +      L = 0;
8.        if (x[L] > y[L])
9.            vL = 1;
10.   }
11.
12.   if (x[0] > y[2])
13.       return -1;
14.
15.   return vL + vR;
}
```

Mutant M_1 (Relevant)

For test input: *x = {0, 3 ,4}* and **y** = *{0, 2, 3}*, the return codes are following:

- Mutant post-commit: **0**
- Mutant pre-commit: **1**
- Original post-commit: **1**

```
int func (int x[3], int y[3]) {
1.    int L, R, vL = 0, vR = 0;
2.    sort(x); sort(y);
3.    R = 2;
4.    if (x[R] > y[R]) {
5.        vR = 1;     // vR = 0;
6.    } else if (x[R] == y[R]) {
7. -      L = 1;
7. +      L = 0;
8.        if (x[L] > y[L])
9.            vL = 1;
10.   }
11.
12.   if (x[0] > y[2])
13.       return -1;
14.
15.   return vL + vR;
}
```

Mutant M_2 (Non-relevant)

No test can execute both the mutated statement (line 5) and the modification (line 7) in both pre and post commit versions

```
int func (int x[3], int y[3]) {
1.    int L, R, vL = 0, vR = 0;
2.    sort(x); sort(y);
3.    R = 2;
4.    if (x[R] > y[R]) {
5.        vR = 1;
6.    } else if (x[R] == y[R]) {
7. -      L = 1;
7. +      L = 0;
8.        if (x[L] > y[L])
9.            vL = 1;
10.   }
11.
12.   if (x[0] > y[2]) // if (x[0] >= y[2])
13.       return -1;
14.
15.   return vL + vR;
}
```

Mutant M_3 (Non-relevant)

Any test that kills the mutant post-commit must fulfil the condition $x[0] == y[2]$. Any test that fulfil the above condition will make the mutant output -1 for pre and post commit versions. Thus no test can make the mutation interact with the modification.

Figure 3.2: Example of relevant and non-relevant mutants. Mutant 1 is relevant to the committed changes. Mutants 2 and 3 are not relevant.

Figure 3.2 illustrates the concept of relevant mutants. The example function takes 2 arguments (integer arrays x and y of size 3), sorts them, makes some computations, and outputs an integer. The commit modification alters the statement at line *7* by changing the value assigned to the variable L from *1* to *0*, denoted with the pink-highlighted line (starting with '-') for the pre-commit version and green-highlighted line (starting with '+') for the post-commit version.

The sub-figure on the left side shows mutant M_1. M_1 is characterized by the mutation that changes the statement $R = 2$ into $R = 0$ in line 3 (the C language style comment represents the mutant's statement). We observe that, with an input t such that $t : x = \{0, 3, 4\}, y = \{0, 2, 3\}$, the original program post-commit has an output value of 1, the mutant M_1 pre-commit outputs 1 and the mutant M_1 post-commit outputs 0. Based on the definition of relevant mutants, M_1 is relevant to the commit modification.

The sub-figure in the center shows mutant M_2 (mutation changes the statement $vR = 1$ into $vR = 0$ in line 5). We observe that the mutated statement (in line 5) and the modification (in line 7) are located in two mutually unreachable nodes of the control-flow graph. Thus, no test can execute both the changed statement and M_2. M_2 is not relevant to the commit modification.

The sub-figure on the right side shows mutant M_3 (mutation changes the expression $x[0] > y[2]$ into $x[0] >= y[2]$ in line 12). We observe that some tests execute both the commit modification and the mutated statement. However, no test can kill M_3 in the post-commit version and at the same time differentiate between the outputs of the pre-commit and post-commit versions of mutant

M_3. The reason is that any test that kills M_3 in the post-commit version must fulfil the condition $x[0] == y[2]$. Any such test makes both the pre- and post-commit versions of M_3 to output -1, thus, not fulfilling the condition to be relevant. Since, there exists no such test M_3 is not relevant to the commit modification.

Note that in case a modification inserts statements, all killable mutants (in the post-commit version) located on these statements (new statements) are relevant to the modification. In case of deletion (modifications remove statements), the mutations located on these statement do not exist in the post-commit version, and thus, are not considered.

3.3 Experimental Setup

3.3.1 Research Questions

We start our analysis by recording the prevalence of commit-relevant mutants in code commits. Thus, we ask:

RQ1: (Mutant distributions) What ratio of mutants is relevant, is located on changed code, and is located on non-changed code?

Answering this question will help us understand the extent of "noise" included in the mutation score and will provide a theoretical upper bound on the application cost of commit-aware mutation testing.

As we shall show, the majority of the mutants are irrelevant to the committed code, indicating that using all mutants is sub-optimal in terms of application cost. Perhaps more interestingly, using such an unbalanced set could result in a score metric with low precision. Therefore, we need to check the extent to which mutation score is adversely influenced by irrelevant mutants. Thus, we investigate:

RQ2: (Metrics relation) Does the mutation score (MS), computed based on all mutants, on mutants located on the committed/modified code, and the delta of the pre- and post- commit MS correlate with the relevant mutation score (rMS)?

Knowing the level of these correlations can provide evidence in support (or not) of the commit-aware assessment (i.e., the extent to which mutation score reflects the level at which the altered code has been tested). In particular, in case there is a strong correlation, we can infer that the influence of the irrelevant mutants is minor. Otherwise, the effects of the irrelevant mutants may be distorting.

While the correlations reflect the influence of the irrelevant mutants on the assessment metric, they do not say much about the extent to which irrelevant mutants can lead to tests that are relevant to the changed behaviours (in case mutants are used as test objectives). In other words, it is possible that by killing random mutants (the majority of which is irrelevant), one can also kill relevant mutants. Such a situation happens when considering the relation between mutants and faults, where mutant killing ratios have weak correlation with fault detection rates but killing mutants significantly improves fault revelation [167]. Hence we ask:

RQ3: (Test selection) To what extent does the killing of random mutants result in killing commit-relevant mutants?

We answer this question by simulating a scenario where a tester analyses mutants and kills them. Thus, we are interested in the relative differences between the relevant mutation scores when testers aim at killing relevant and random mutants. We use the random mutant selection baseline as it achieves the current best results [34, 112]. We compare here on a best effort basis, i.e., the commit-relevant mutation score achieved by putting the same level of effort, measured by the number of mutants that require analysis. Such a simulation is typical in mutation testing literature [38, 112] and aims at quantifying the benefit of one mutant selection approach over another.

Answering the above question provides evidence that killing relevant mutants yields significant advantages over the killing of random mutants. While this is important and demonstrates the potential of killing commit-relevant mutants in terms of relevance, still the question of actual test effectiveness (actual fault revelation) remains. This means that it remains unclear what the fault revelation potential of killing commit-relevant mutants is when the commit is fault-introducing. Therefore we seek to investigate:

RQ4: (Fault Revelation) How does killing commit-relevant mutants compares with the killing of random mutants w.r.t. to (commit-introduced) fault revelation?

To answer this question we investigate the fault revelation potential of killing commit-relevant mutants based on a set of real fault-introducing commits. We follow the same procedure as in the previous research question (RQ3) in order to perform a best effort evaluation.

Overall, answering the above questions will improve the understanding of the potential of the cost-effectiveness application of commit-aware mutation testing.

3.3.2 Analysis Procedure

We performed mutation testing on the selected subjects using all the mutation operators supported by Mart [36] and Pitest [41] (the mutation testing tools we use). For the C programs, we then discarded all the trivially equivalent mutants (including the duplicated ones), using the TCE method [100] and applied our analysis on the resulting sets of mutants.

Identifying relevant mutants requires excessive manual analysis, thus we approximate them based on test suites (this is a typical experimental procedure [6, 112, 166]). To do so we composed large test pools, which approximate the input domain. The pools are composed of the post-commit version developer tests (mined from the related repository). For C programs we augment the pools with automatically generated tests, similarly to the process followed by Kurtz et al and Papadakis et al. [112, 166].

Using the test pools, we execute all the mutants (on both pre- and post-commit versions) and construct the mutation matrix that records the mutants killed by each test case of the pool. We also record the test execution output of each test on each mutants. For C programs, this output is the standard output produced when running the test, while for the Java programs it is the status (pass/fail) of the test run.

By using the test execution outputs and the mutant matrices, we approximate the relevant mutant set, from the post-commit mutants, based on Algorithm 1. In the algorithm, the function calls *postCommitOrigOutput*, *postCommitMutOutput* and *preCommitMutOutput* compute the output of the execution of test case 'test' on the post-commit original program, post-commit version of mutant 'mut' and pre-commit version of mutant 'mut', respectively.

Besides the relevant mutant set, we also extract the modification mutant set, made of mutants that are located on a statements modified or added by the commits. This set is computed by extracting the modified or added statements from the commit *diff* and collecting the mutants that mutate those statements. Note that, by definition, the killable modification mutants are also relevant mutants, as their pre-commit output is not defined, and thus different from their post-commit output.

We have three mutant sets: the post-commit, relevant and modification mutant sets. In RQ2, we want to know the correlations between the mutation scores of the aforementioned mutant sets. To do so, we select arbitrary test sets of various sizes and record the mutation scores on each mutant set and compute their correlations.

```
Algorithm 1: Approximate Relevant Mutants Set
Data: TestSuite, Mutants
Result: Relevant Mutants
$RelevantMuts \leftarrow \emptyset$;
for $mut \in Mutants$ do
    for $test \in TestSuite$ do
        $origV2 \leftarrow postCommitOrigOutput(test)$;
        $mutV2 \leftarrow postCommitMutOutput(test, mut)$;
        $mutV1 \leftarrow preCommitMutOutput(test, mut)$;
        if $origV2 \neq mutV2 \wedge mutV2 \neq mutV1$ then
            $RelevantMuts \leftarrow RelevantMuts \cup \{mut\}$;
            break;
        end
    end
end
return $RelevantMuts$ ;
```

In RQ2 we arbitrary pick sets of tests representing 10%, 20%, ..., 90% of the test pool. As these sets are randomly sampled we selected multiple sets (500 for C and 100 for Java) per size considered and per program commit (each subset of test can be seen as a testing scenario). For every test set, we computed the mutation score for each of the three mutant sets. We name as *MS*, *rMS* and *mMS* the mutation scores for the whole mutant set, relevant mutant set and modification mutant set, respectively. The mutation scores are computed on the post-commit versions and using the mutation matrix. Thus, for each commit and each test size, we have three statistical variables (*MS*, *rMS* and *mMS*), which instances are the corresponding mutation scores for each test set.

Having collected the data for the statistical variables *MS*, *rMS* and *mMS*, we compute the correlations between *rMS* and *MS* as well as the correlation between *rMS* and *mMS*. If the correlation between *rMS* and *MS* (*mMS*) is high, it means that *MS* (*mMS*) can be used as a proxy fo *rMS*. Otherwise, *MS* (*mMS*) is not a good proxy for *rMS* and thus, *rMS* should be targeted directly.

We also computed, for each test set, the mutation score in the pre-commit version. Then we compute the absolute change of mutation score (named *deltaMS*), on the analyzed mutant set, incurred by a commit modification (*delatMS* $= |MS_{post-commit} - MS_{pre-commit}|$), and we compute the correlation between *rMS* and *deltaMS*. A strong correlation would mean that the absolute change of mutation score between versions is a proxy for *rMS*. Weak correlation would mean that *rMS* cannot be represented by *delatMS*.

In RQ3, we simulate a scenario where a tester selects mutants and designs tests to kill them. This is a typical evaluation procedure [112, 166] where a test that kills a randomly selected mutant (from the studied mutant set) is selected from the test pool. This test is then used to determine the killed mutants, which are discarded from the studied mutant set. The process continues (by picking the next live mutant) until all mutants have been killed. If a mutant is not killed by any of the tests, we treat it as equivalent. This means that our effort measure is the number of mutants picked (either killable or not) and effectiveness measure is the relevant mutation score. Since we perform a best-effort evaluation we focus on the initial few mutants (up to 50) that the tester should analyse in order to test the commits under test. We repeat this process (killing all mutants) 100 times and compute the relevant mutation score.

For RQ4, we repeat the same procedure as in RQ3. However, instead of computing the relevant mutation score, we compute the fault revelation probability.

3.3.3 Statistical Analysis

We perform a correlation analysis to evaluate whether the mutation score, when considering all mutants, correlates with the relevant mutation score. To this end, we use two correlation metrics: *Kendall rank coefficient (τ)* (Tau-a) and *Pearson product-moment correlation coefficient (r)*. In all cases, we considered the 0.05 significance level.

The Kendall rank coefficient τ, measures the similarity in the ordering of the studied scores. We measure the mutation score MS and the relevant mutation score rMS when using test suites of size 10%, ..., 90% of the test pools. The Pearson product-moment correlation coefficient (r) measures the covariance(linear correlation) between the MS and rMS values. These two coefficients take values from -1 to 1. A coefficient of 1, or -1, indicates a perfect correlation while a zero coefficient denotes the total absence of correlation.

To evaluate whether the achieved mutation scores MS and relevant mutation scores rMS are significantly different (i.e., different data distribution), we use a Mann-Whitney U Test performed at the 1% significance level. This statistical test yields a probability called p-value which represents the probability that the MSs and rMS are equal. Thus, a p-value lower than 1% indicates that the two metrics are statistically different. We use paired and two-tailed U test, to account for the different commits and programs.

3.3.4 Program Versions Used

To answer RQs 1-3 we used the C programs of GNU Coreutils[1], used in many existing studies [29, 35, 107]. GNU Coreutils is a collection of text, file, and shell utility programs widely used in Unix systems. The whole code-base of Coreutils is made of approximately 60,000 lines of C code[2]. In order to obtain a commit benchmark of Coreutils programs we used to following procedure to mine recent commits from the Coreutils github repository. (1) We set the commit date interval from year 2012 to 2019. This resulted in 5,000 commits considered. (2) Next, we filtered out the commits that do not alter source code files. This resulted in 1,869 commit remaining. (3) Then, we only kept the commits that affect only the main source file of a single program (This enable better control of test execution, because other programs of Coreutils are often used to setup the test execution of a tested program). (4) After that, we filtered out commits that are very large (commits whose modification has an edit actions of more than 5 according to GumTree [53]). This resulted in 218 commits. (5) Due to the large execution time of the experiments, approx. 2 weeks of CPU time per commit, we randomly sampled 34 commits among the remaining commits for the experiments. This constitutes our Benchmark-1.

In order to further strengthen our experiment and answer RQ4, we also use 13 commits from the CoREBench [23] that introduce faults. We selected these commits to validate the fault revelation ability of relevant mutants. Since we approximate relevant mutants, we needed commits where automated tests generation frameworks could run. Thus, we limit ourselves to the 18 fault introducing commits of Coreutils that we can run with Shadow symbolic execution [107]. Among these faults, two were discarded due to technical difficulties in compiling the code (the build system uses very old versions of the build tools). Three faults were discarded due to the excessively high required execution time to run the mutants (we stopped after 45 days).

Table 4.1 summarizes the informations about the C language benchmarks used in the experiments.

To answer RQs 1-3, we also consider a set of commits from well-known and well-tested Java programs. We extract these commits from projects in the Apache Commons Proper repository[3], a set of

[1]https://www.gnu.org/software/coreutils/
[2]Measured with cloc (http://cloc.sourceforge.net/)
[3]`https://commons.apache.org/`

Table 3.1: C Test Subjects

Benchmark	#Programs	#Commits	# Mutants	#Test cases
CoREBench [23]	6	13	154,396	8,828
Benchmark-1	13	34	338,390	11,866

Table 3.2: Java Test Subjects

Project	# Commits	# Mutants	# Test cases
commons-cli	9	61,419	3,247
commons-collections	5	323,584	55,076
commons-io	3	105,181	3,972
commons-net	6	345,130	1,478
joda-time	5	561,782	20,962
jsoup	8	330,125	4,985

reusable Java component projects, from Joda Time[4], a time and date library, and Jsoup[5], an HTML manipulation library. For each of the projects, we manually gathered the most recent commits meeting the following conditions from the project's history: (1) only source code is modified, no modification to configuration files, (2) the commit introduces a significant change, not a trivial one such as a typo fix, (3) test contracts are not modified, in order to meaningfully compare pre- and post-commit outputs and (4) both pre- and post-commit versions of the project build successfully. Overall, we gathered 36 commits, table 3.2 summarises information about the commits used from each project.

3.3.5 Mutation Mapping Across Versions

As mutation testing tools generate mutants for a given program version instead of regression pairs, we need to identify the common mutants between the two versions. In other words, we need to map each mutant from its pre- to post-commit version of the program.

To establish such a mapping in the case of C programs, we unify the commit modifications into a single program, as done in the literature [107], and apply any standard (unmodified) mutation tool to generate the mutants. The code unification of the commit modification is done through annotation that has no side-effect. The annotations are made through a special function called *"change"* that takes 2 arguments/values (the arguments are the value of the pre-commit and post-commit versions, respectively) and return one of the two values.

The annotations are manually inserted in the program, according the semantics presented in previous studies [107].

Note that the statement insertion can be annotated by wrapping the inserted statement with $if(change(false, true))$; and a statement deletion can be annotated by wrapping the deleted statement with $if(change(true, false))$.

The choice of the version to use, for each mutant, is decided at runtime (by specifying the version to use through an environment variable recognizable by the *change* function).

For the Java programs, we perform the mapping of mutants from both sets of mutants and the commit diff. We first generate the mutants for both pre- and post-commit versions of the program

[4]`https://github.com/JodaOrg/joda-time/`
[5]`https://github.com/jhy/jsoup`

using the mutation tool. We then map pre- and post- commit line numbers by parsing the commit diff, and use this mapping to map pre- and post-commit mutants, using the line number, bytecode instruction number and mutation operator of the mutants to match both sets. We adopt this way for the Java programs in order to avoid making drastic changes on Pitest (the mutation testing tool we use).

3.3.6 Mutation Testing Tools and Operators

As test suites are needed in our experiment, we use the developer tests suites for all the projects that we studied. These were approximately 4,194 tests in total for C programs.

To strengthen the test suites used in our study, we augment them in two phases. First, we use KLEE [29], with a robust timeout of 2 hours, to perform a form of differential testing [51] called shadow symbolic execution [107], which generates 234 test cases. Shadow symbolic execution generates tests that exercise the behavioural differences between two different versions of a program, in our case the pre-commit and the post-commit program versions.

In order to also expose behavioural difference between the original program and the mutants, we used SEMu [35], with a robust timeout of 2 hours, to perform test generation to kill mutants in the post-commit program versions. SEMu generates 17,915 test cases.

These procedures resulted in large test suites of 22,343 test cases for C programs in total. Since we compare program versions, we use the programs output as an oracle. Thus, we consider as distinguished or killed, every mutant that results in different observable output than the original program.

We use Mart [36], a mutation testing tool that operates on LLVM bitcode, to generate mutants. Mart implements 18 operators (including those supported by modern mutation testing tools), composed of 816 transformation rules.

To reduce the influence of redundant and equivalent mutants, we enabled Trivial Compiler Equivalence (TCE) [78, 100] in Mart to detect and remove TCE equivalent and duplicate mutants. TCE detected 13,322 and 460,072 equivalent and redundant mutants.

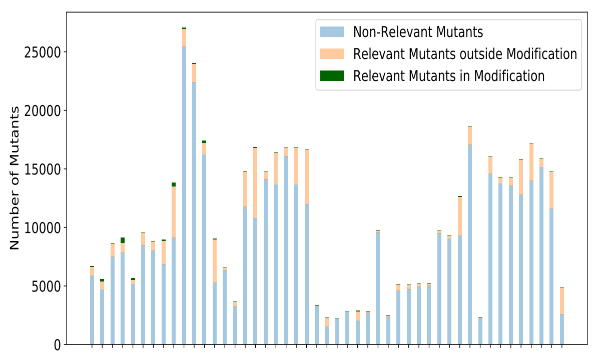
For the Java programs, we use the developper test suites available. We perform mutation analysis using Pitest[41], a state of the the art mutation testing tool that mutates JVM bytecode. We use all mutation operators available in Pitest, which are described in [114] and [40].

3.4 Results

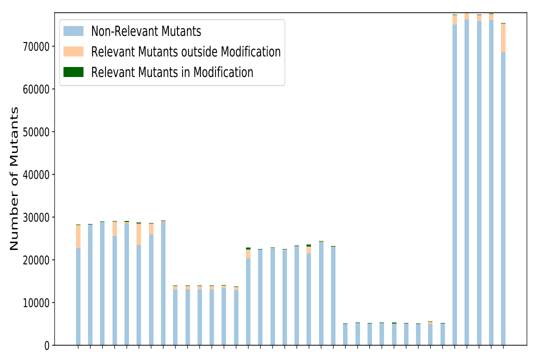
3.4.1 RQ1: Relevant mutant distribution

We start our analysis by examining the prevalence of commit-relevant mutants, i.e., mutants that affect the altered program behaviours. Figure 3.3 records the distribution of the relevant and non-relevant mutants among the studied commits. Based on these results we see that only a small portion of the mutant population produced by the selected mutation operators is actually relevant. This portion ranges from 0.5% to 47%, among which 3.6% is located on the changed program lines, while the rest is located on the rest of the code. For the large portion, it is possible to happen when the source code is not large, and the change is located in the crucial position.

Interestingly, the presence of so many "irrelevant" mutants, can have major consequences when performing mutation testing. Such consequences are a distorting effect on the accuracy of the mutation score, and a waste of resources when executing and trying to kill non-relevant to the commit mutants. We further investigate these two points in the following sections.



(a) C programs

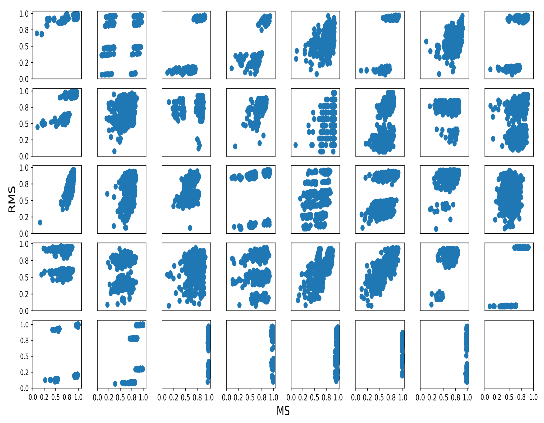


(b) Java programs

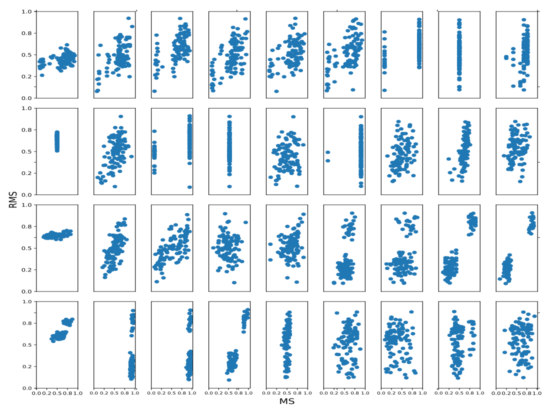
Figure 3.3: The distribution of killable, non-relevant, relevant outside the modification and relevant on the modification mutants among the studied commits.

3.4.2 RQ2: Relevant mutants and mutation score

Figure 3.4 visualizes our data; each data point represents the mutation score and relevant mutation score of a selected test suite. As can be seen from the scatter plots, there is no visible pattern or trend among the data. We can also see that there is a large variation between mutation scores and relevant mutants scores in almost all the cases. These observations indicate that the examined variables differ significantly. In other words, one cannot predict/infer one variable using the other one. To further explore the relationship between mutation score and relevant mutation score within our data we perform statistical correlation analysis.



(a) C programs



(b) Java programs

Figure 3.4: The relationship between Mutation Score and Relevant Mutation Score.

Finding a strong correlation would suggest that the two metrics have similar behaviours (an increase or decrease of one implies a relatively similar increase or decrease of the other). Figure 3.5 displays the results for the two correlation coefficients that have statistically significant values for randomly selected test suites (from our test suite pool) of different sizes. The first row shows the Kendall correlation. Interestingly, we observe that most of the correlations are relatively weak with their majority ranging from 0.15 to 0.35. Additionally, we see that both coefficients we examine are aligned, indicating a weak relationship when either ordering test suites or considering their score differences. We observe similar trends with Pearson correlation as show in the second row in Figure 3.5.

One may assume that the relevant mutation score may be well approximated by the mutants that are located on the modified code, assuming that mutants' location reflects their utility and relevance. Similarly, one may assume that the commit-relevant score could be approximated by the delta of the pre- and post-commit mutation scores. We investigate these cases and find that most of the correlations are relatively weak with their majority ranging from -0.1 to 0.1.

Overall, our results indicate that irrelevant mutants have a major influence on the mutation score calculation, and that using the overall mutation score does not reflect the actual value of interest, i.e., how well the altered behaviours are tested, which is represented by relevant mutation score (rMS). Approximating the rMS using either the deltaMS or the mutants of the altered lines is also not sufficient. Hence, our results suggest that MS and other direct metrics are not good indicators of commit-related test effectiveness. We envision that future research should develop techniques capable of identifying relevant mutants at testing time, i.e., prior to any test generation and mutant analysis, in order to support testers.

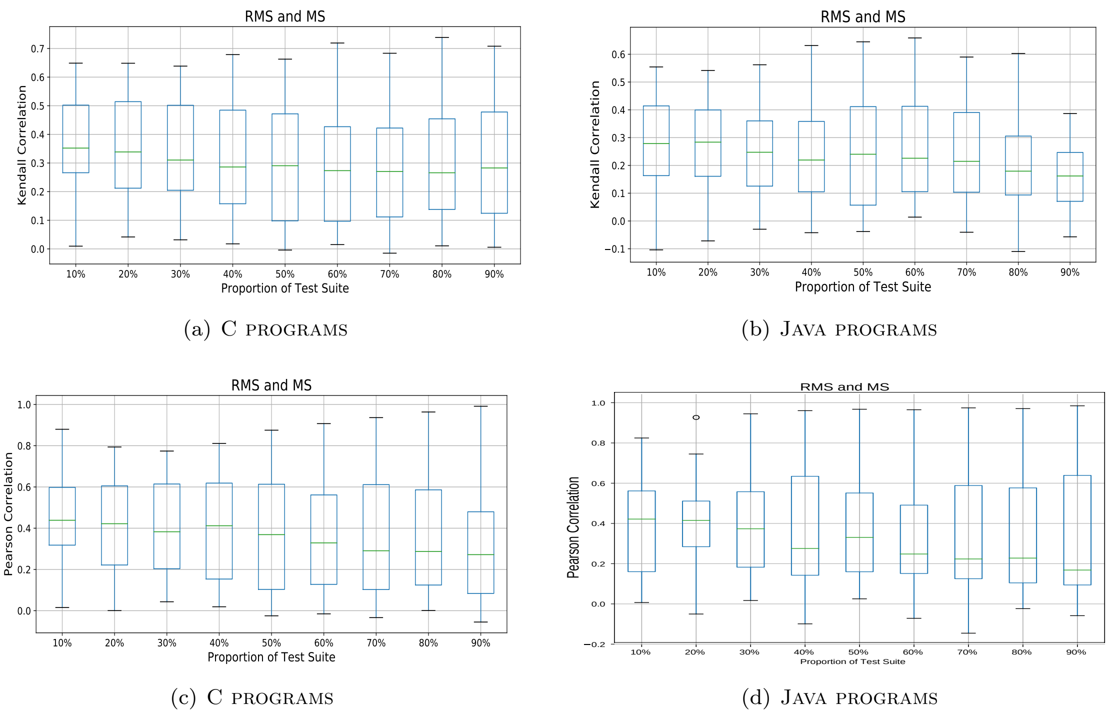
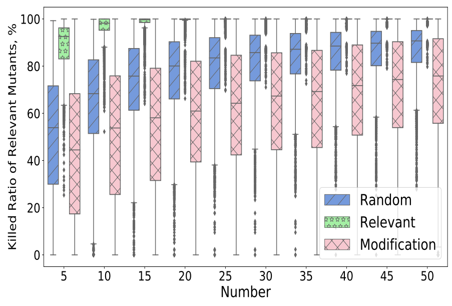


Figure 3.5: Correlation between Mutation Score and Relevant Mutation Score for different test suite sizes on different languages.

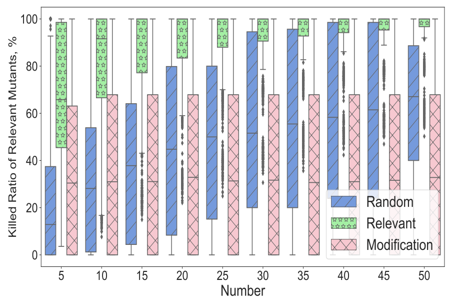
3.4.3 RQ3: Test Selection

Recent research has shown that mutation testing is particularly effective at improving test suites and revealing faults (guiding testers to design test cases that reveal faults), while at the same time mutation score is weakly correlated with fault detection [167]. In view of this, it is possible that despite the weak correlations we observe in our case, traditional mutation could successfully guide testers towards designing tests that collaterally kill relevant mutants.

Results are recorded in Figure 3.6 for the first 1-50 mutants to be analysed by the tester. We observe a large divergence (approximately 50%-60%) between the random, commit-based and relevant mutants. This difference is statistically significant and with large effect size (Effect Size values are recorded on Table 3.3). Taking together the weak correlations we found in the previous section with these results, we conclude that traditional mutation testing is suboptimal and cannot be used to assess or guide (in a best-effort basis) the testing of committed code. Therefore, to support practitioners, future research should aim at identifying and using commit-relevant mutants. Similarly, controlled experiments should be based on relevant mutants when aiming at assessing change-aware test effectiveness.



(a) C programs



(b) Java programs

Figure 3.6: Test suite improvement of mutation-based testing with random (traditional mutation) and relevant mutants.

Table 3.3: $\hat{A}_{12}$. rMS when aiming at Relevant, Random and Modification related mutants.

#Mutants	**5**	**10**	**20**	**30**	**40**	**50**
Relevant-Random	0.90	0.95	0.98	0.98	0.98	0.97
Relevant-Modification	0.89	0.96	0.99	0.99	0.99	0.99

Table 3.4: $\hat{A}_{12}$. Fault revelation when aiming at Relevant, Random and Modification related mutants.

% Relevant mutants analysed	**10%**	**20%**	**50%**	**75%**	**100%**
Relevant-Random	0.55	0.59	0.64	0.66	0.64
Relevant-Modification	0.57	0.59	0.69	0.73	0.70

3.4.4 RQ4: Fault Revelation

To demonstrate the importance of commit-aware mutation testing, we further compare the ability of the traditional mutants and commit-relevant mutants to reveal commit-introduced faults (real faults). We follow the same procedure as in the previous section but evaluate w.r.t. to the rate of faults revealed by the selected test suites.

The fault revelation results are depicted in Figure 3.7. From this data, we can see that a significant fault revelation difference (approximately 30-40%) between the compared approaches can be recorded. This difference is statistically significant with large effect size (Effect Size values are recorded on Table 3.4). Here it must be noted that these results can be achieved by an effort equivalent to analysing 0.4% of the mutants, which is 27 mutants per commit (on average).

Overall, our results demonstrate that by aiming at relevant mutants one can achieve significant fault revelation benefits (approximately 30%) over the traditional way of using mutation testing.

3.5 Threats to validity

External validity: We selected commits that do not modify test contracts. Such commits are common in industrial CI pipelines [117] but rare in open source projects. To mitigate this threat, we performed our analysis on a relatively large set of commits given the computational limits posed by mutation analysis. In C, our experiment required on average approximately 2 weeks of CPU time to complete, per commit studied (executions performed using Muteria [37]). In addition, we used an established research benchmark (CoREBench [23]) where we found similar results. Unfortunately, we consider

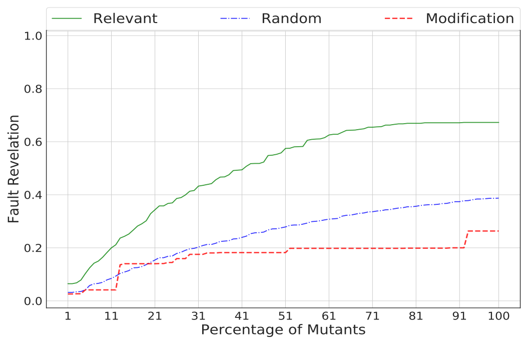


Figure 3.7: Fault revelation of mutation-based testing with random (traditional mutation) and relevant mutants.

fault introducing commits only in C as the Java datasets do not adhere to our non-changed test contract requirement.

Another threat may relate to the mutants we use. To reduce this concern we used a variety of operators covering the most frequently used language features including the operators adopted by the modern tools [114], in both C and Java.

Internal validity: Such threats lie in the use of automated tools, the way we treated live mutants and non-adequate test suites. To diminish these concerns, we used KLEE, a state of the art test generation tool and strong mature developer test suites. Nevertheless, the current state of practice [174] relies on non-adequate test suites, so our results should be relevant to at least a similar level of practice. To ensure our results, we carefully checked our implementation and performed a manual evaluation on a sample of our results. Moreover, we use established tools also employed by numerous studies.

To deal with randomness and minimize stochastic effects, we repeated our experiments 100 times and used standard statistical tests and correlations.

Construct validity: Our effort related measurement, number of analysed mutants, essentially captures the manual effort involved in test generation. Automated tools may reduce this effort and change our best-effort results. Still, we used the current standards, i.e., TCE [100] to remove all trivially equivalent mutants before conducting any experiment and KLEE (including a mutation-based test generation approach [35]). In test generation, we acknowledge that automated tools may generate test inputs that kill mutants, but we note that they fail to generate test oracles. Therefore, even if such tools are used, the test oracles will still require human intervention, i.e., introduce some effort. Here it should be noted that we consider the mutant execution cost as negligible since it is machine time and our focus is on the human time involved when performing mutant analysis. Moreover, existing advances [236] promises to reduce this cost to a practically negligible level.

Overall, we believe that our effort measurements approximate well (in relative terms) the human effort involved. All in all, we aimed at minimizing potential threats by using various metrics, well-known tools and benchmarks, real and artificial faults and following methodological guidelines [166]. Additionally, to enable reproducibility and replication we make our tools and data publicly available[6].

[6]Our data and results are openly accessible on the following Github link: `https://github.com/relevantMutationTesting`

3.6 Related Work

There are various methods aiming at identifying relevant coverage-based test requirements in the literature. For instance, it has been proposed to consider as relevant every test element that can be affected by the changes (by doing some form of slicing, i.e., following all control and data dependencies from the changed code) [17, 184]. As such, these methods aim at considering conservatively every test requirement affected by the change, resulting in sets with a large number of irrelevant requirements. Nevertheless, applying such an approach to mutation testing is equivalent to mutating the sliced program. This of course inherits all the limitations of program slicing such as scalability and precision [19], it is conservative (results in large number of false positives) and does not account for equivalent mutants located on potentially infected code.

To circumvent the problems of coverage, researchers have proposed the propagation-based techniques [10, 187, 188, 189], which aim at identifying the program paths that are affected by the program changes. They rely on dependence analysis and symbolic execution to form propagation conditions and decide whether changes propagate to a user-defined distance. Although promising, these techniques are complex and inherit the limitations from symbolic execution.

Researchers have also investigated techniques to automatically augment test suites by generating tests that trigger program output differences [180], increase coverage [230] and increase mutation score [196, 197]. Along the same lines differential symbolic execution [172], KATCH [141] and Shadow symbolic execution [107] aim at generating tests that exercise the semantic differences between program versions by incrementally searching the program path space from the changed locations and onwards. These methods are somehow complementary to ours as they can be used to create tests that satisfy the commit-relevant test requirements.

Interestingly, the problem of commit-relevant test requirements has not been investigated by the mutation testing literature [166]. Perhaps the closest work to ours is the regression mutation testing by Zhang et al. [240] and the predictive mutation testing by Zhang et al. and Mao et al. [139, 236]. Regression mutation testing aims at identifying affected mutants in order to incrementally calculate mutation score, while predictive mutation testing aims at estimating the mutation score without mutant execution. Apart from the different focus (we focus on commit-relevant mutants and refined score, while they focus on speeding up test execution and mutation score) and approach details, our fundamental difference is that we statically target killable mutants (both killed and live by the employed test suites) that are relevant to the changed code (we ignore irrelevant code parts and mutants).

3.7 Conclusion

We proposed commit-aware mutation testing, a mutation-based assessment metric capable of measuring the extent to which the program behaviours affected by some committed changes have been tested. We showed that commit-aware mutation testing has a weak correlation with the traditional mutation score and other regression testing approximations (such as the delta on mutation score between the pre- and post- commit versions and mutants located on modified code) indicating that it is a distinct metric. Our results also showed that traditional mutant selection is non-optimal as it loses approximately 50%-60% of the commit-relevant mutants when analysing 5-25 mutants and has 30% less chances of revealing commit-introducing faults.

4 MuDelta: Delta-Oriented Mutation Testing at Commit Time

To effectively test program changes using mutation testing, one needs to use mutants that are relevant to the altered program behaviours as shown in Chapter 3. We introduce *MuDelta*, an approach that identifies commit-relevant mutants; mutants that affect and are affected by the changed program behaviours. Our approach uses machine learning applied on a combined scheme of graph and vector-based representations of static code features. Our results, from 50 commits in 21 Coreutils programs, demonstrate a strong prediction ability of our approach; yielding 0.80 (ROC) and 0.50 (PR-Curve) AUC values with 0.63 and 0.32 precision and recall values. These predictions are significantly higher than random guesses, 0.20 (PR-Curve) AUC, 0.21 and 0.21 precision and recall, and subsequently lead to strong relevant tests that kill 45% more relevant mutants than randomly sampled mutants (either sampled from those residing on the changed component(s) or from the changed lines). Our results also show that *MuDelta* selects mutants with 27% higher fault revealing ability in fault introducing commits. Taken together, our results corroborate the conclusion that commit-based mutation testing is suitable and promising for evolving software.

Contents

4.1 Introduction

Mutation testing has been shown to be one of the strongest fault-revealing software test adequacy criteria available to software testers [38]. Nevertheless, although mutation testing has been widely studied for over four decades in the scientific literature, the formulation that underpins it has remained largely unchanged since its inception in the 1970s [25, 46]. In this unchanged formulation as described in Chapter 1, a program p is tested by a test suite, T, the adequacy of which is measured in terms of its ability to distinguish executions of p and a set of mutants M. Each mutant in M is a version of p into which a fault has been deliberately inserted, in order to simulate potential real faults, thereby assessing the ability of the test suite T to detect such faults.

The problem with this formulation is that it has not kept pace with recent software engineering practices. Most notably, the assumption of a fixed program p, set of mutants M, and test suite T, is unrealistic; modern software systems undergo regular change, typically in continuous integration environments [59, 80, 117]. In order to render mutation testing applicable to practising software engineers, a fundamentally new approach to finding suitable mutants is required in which p, T, and M are each continually evolving. Chapter 3 studies the commit-relevant mutants in Continuous Integration for the evolving systems. Specifically, we need a mutation testing formulation in which mutants can be found, on the fly, based on their relevance to specific changes to the system under consideration. In this 'evolving mutation testing' approach, both the set of mutants M and the tests that distinguished their behaviours T, are each able to change with each new commit. Such a mutation testing formulation is better suited to industrial practice, e.g., at Google [174], since mutation testing can be applied at commit time, to each code change as it is submitted, thereby keeping pace with the changes to p. More importantly, such an approach will focus the test effort deployed at commit time specifically to the changes in the commit, rather than wasting test effort on re-testing old code. In order to apply mutation testing on the fly in this manner, we need a fast lightweight approach to determine a priority ordering on a given set of mutants, where priority is determined by the relevance of a mutant to the change in hand.

This chapter introduces a machine learning-based approach to tackle this problem using a combined scheme of graph and vector-based representations of simple code features that aim at capturing the information (control and data) flow and interactions between mutants and committed code changes. We train the learner on a set of mutants from historical code changes that are labeled with respect to given test suites. The machine learner is subsequently used to predict the priority ordering of the set of mutants to identify those most likely to be relevant to a given change.

This way, once the learner has been trained, it can be used to quickly predict the priority order for the set of mutants in terms of their relevance to unseen changes, as they are submitted into the continuous integration system for review. This allows the tester (and/or some automated test design technology) to focus on those mutants that are most likely to yield tests that are fault revealing for the change in hand.

We implemented our approach in a system called *MuDelta*, and evaluated it on a set of 50 commits from Coreutils wrt a) prediction ability, b) ability to lead to relevant tests (tests killing commit-relevant mutants) and c) ability to reveal faults in fault introducing commits. Our results indicate s strong prediction ability; *MuDelta* yields 0.80 ROC-AUC value, 0.42 F1-score, 0.63 precision and 0.32 recall, while random guesses yield 0.20 F1-score, 0.21 precision and 0.21 recall. Killing the predicted mutants results in killing 45% more relevant mutants than random mutant sampling baselines.

Perhaps more importantly, our results show that our approach leads to mutants with 27% higher fault revealing ability in fault introducing commits. Taken together, our results corroborate the findings that *MuDelta* enables effective delta-relevant mutation testing, i.e., mutation testing targeting the specific code changes of the software system under test.

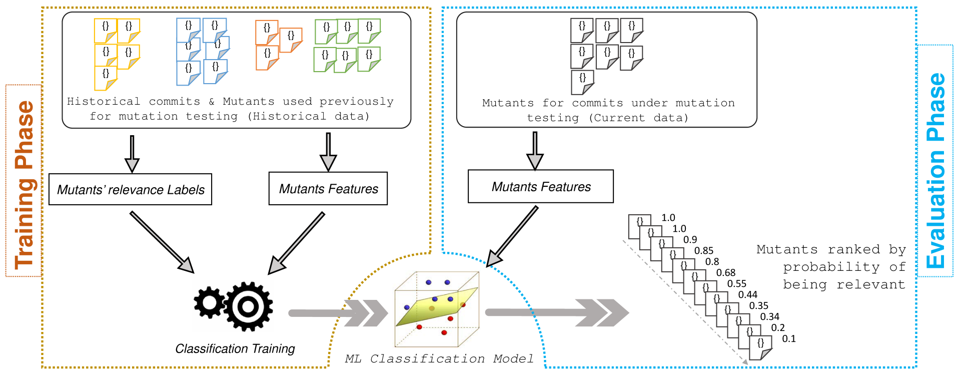

Training Phase
Historical commits & Mutants used previously for mutation testing (Historical data)
Mutants' relevance Labels
Mutants Features
Classification Training
ML Classification Model
Evaluation Phase
Mutants for commits under mutation testing (Current data)
Mutants Features
Mutants ranked by probability of being relevant
1.0
1.0
0.9
0.85
0.8
0.68
0.55
0.44
0.35
0.34
0.2
0.1

Figure 4.1: Overview of *MuDelta*. The learner is trained on a set of mutants from historical code changes that are labeled with respect to given test suites. The machine learner is subsequently used to predict the priority ordering of the set of mutants to identify those most likely to be relevant to a given change.

Chapter 3 has shown that mutants that resides in the changed code are not adequate in testing the change. This finding highlights the importance of locating mutants in the unchanged part of the program. This unchanged code that forms a contextual environment into which changes deployed. Such δ-relevant mutants in the context C, for some change, δ, tend to focus on (and reveal issues with) interactions between the change, δ, and the context C into which it is deployed. Developers are less likely to notice these since they are more likely to be familiar with their changes than the existing unchanged code. Such bugs may also be more subtle as they involve unforeseen interactions between parts of the system.

In summary, our primary contributions are:

- The empirical evidence that mutant relevance (to particular program changes) can be captured by simple static source code metrics.
- A machine learning approach, called *MuDelta*, that learns to rank mutants wrt to their utility and relevance to specific code changes.
- Empirical evidence suggesting that *MuDelta* outperforms the traditionally random mutant selection/prioritization method by revealing 45% more relevant mutants, and achieving 27% higher probability to reveal faults in these changes.

4.2 Context

4.2.1 Change-aware regression testing

Testing program regressions require test suites to exercise the adequacy of testing wrt to the program changes. In case the used test suites are insufficient, guidance should be given in order to help developers create test cases that specifically target the behaviour deviations introduced by the regressions.

One potential solution to this problem may be based on coverage; one can aim at testing the altered parts of the programs using coverage information. However, the strengths of coverage are known to be limited [10, 38]. Moreover, the most severe regression issues are due to unforeseen interactions between the changed code and the rest of the program [10, 188]. Therefore, we aim at using mutation testing using the commit-relevant mutants described in Chapter 3.

4.2.2 Motivation

Chapter 3 proposes commit-relevant mutants for the code change. Commit-relevant mutants are those that make observable any interaction between the altered code and the rest of the program under test. These mutants alter the program semantics that are relevant to the committed changes, i.e., they have behavioural effects on the altered code behaviour. This means that mutants are relevant to a commit when their behaviour is changed by the regression changes. Indeed, changed behaviour indicates a coupling between mutants and regressions, suggesting relevance. In essence, one can use relevant mutants to capture the 'observable' dependencies between changed and unchanged code [18, 101], which reflect the extent to which test suites are testing the altered program behaviours.

The virtue of commit-relevant mutation testing, as described in the study of Chapter 3 is the best-effort application of mutation testing. This gives the potential for improved fault revelation under the same (relatively low) user effort than using randomly sampled mutants, i.e., traditional mutation testing. However, in order to be useful, these mutants need to be identified in advance, prior to any mutant analysis performed. This is because relevant mutants form the objectives that developers will analyse. To achieve this, we develop a machine learning approach, which we describe in the following section.

Figure 4.2 presents a commit-relevant mutant on a fault-introducing commit of GNU Coreutils[1]. This is the commit with ID 8 from CoREBench [23]. The commit affects two functions of the program *seq* (*main* and *seq_fast*). The entry-point is the function *main*, which, calls the functions *print_numbers* and *seq_fast* to compute and print the results. The function *seq_fast* is an optimized implementation of the function *print_numbers*, used only when the inputs meet specific conditions. In Figure 4.2, the line 543 checks the condition to call *seq_fast*. If the condition is satisfied, *seq_fast* is called. Otherwise, *print_numbers* is called. Note that *print_numbers* may be called after *seq_fast* if the later fails (the condition at line 405 is not satisfied, i.e. $a > b$). In that case, the execution of *seq_fast* does not alter the program state or output.

The commit aims at relaxing the condition that guards the call to function *seq_fast*. In the pre-commit version, *seq_fast* is not called when the user specifies a separator. However, in the post-commit version, *seq_fast* is called whenever a) the user specifies a separator, and b) the separator string has a single character.

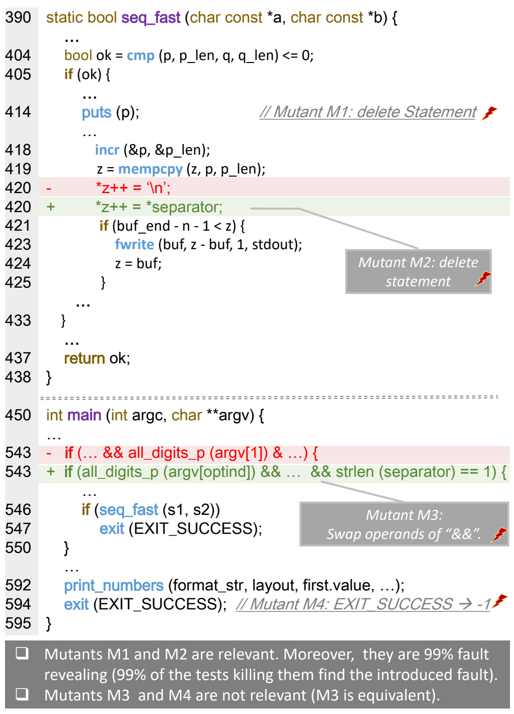
In the function *seq_fast*, the commit only replaces the hard coded separator ('\n') with separator's global *string* variable. In the function *main*, the commit relaxes the "if" condition at line 543, in a way that *seq_fast* is also called when the user specifies a separator, which can be any single "8-bits" character (it is not limited to '\n').

The program *seq* calls *seq_fast* to print all the integers from the first parameter a to the second parameter b, and using a given character (first character of *separator* in post-commit and '\n' in pre-commit) to separate the printed numbers.

Let four mutants such that: mutant M_1 deletes the statement at line 414, which prints the first number using *puts*. Mutant M_2 deletes the modified statement at line 420, which add the separator to the buffer to print. Mutant M_3 swaps the operands of the last "&&" operation at the modified line 543. Mutant M_4 replaces the exit value at line 595 by -1.

We observe that M_4 is not relevant to the commit. In fact, there is no test that can kill M_4 in the post-commit version, and create an output difference between pre- and post-commit versions of M_4. If a test kills post-commit M_4, it must avoid executing line 547, thus, *seq_fast* is either not called or its call does not succeed (does not print anything). Thus, the output of the execution of the pre- and post-commit versions of M_4 with such test will be same (both computed with *print_numbers*, which is not altered by the commit). Mutant M_3 is equivalent, because no clause has side effect that is controlled by another clause in the *if* condition.

[1]https://www.gnu.org/software/coreutils



```
390   static bool seq_fast (char const *a, char const *b) {
        …
404     bool ok = cmp (p, p_len, q, q_len) <= 0;
405     if (ok) {
          …
414       puts (p);                    // Mutant M1: delete Statement
          …
418         incr (&p, &p_len);
419         z = mempcpy (z, p, p_len);
420   -     *z++ = '\n';
420   +     *z++ = *separator;
421          if (buf_end - n - 1 < z) {
423            fwrite (buf, z - buf, 1, stdout);
424            z = buf;
425          }
         …
433     }
        …
437     return ok;
438   }
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
450   int main (int argc, char **argv) {
      …
543   -  if (… && all_digits_p (argv[1]) & …) {
543   +  if (all_digits_p (argv[optind]) && …  && strlen (separator) == 1) {
          …
546       if (seq_fast (s1, s2))
547         exit (EXIT_SUCCESS);
550     }
        …
592     print_numbers (format_str, layout, first.value, …);
594     exit (EXIT_SUCCESS);  // Mutant M4: EXIT_SUCCESS → -1
595   }
```

Mutant M2: delete statement
Mutant M3: Swap operands of "&&".

❑ Mutants M1 and M2 are relevant. Moreover, they are 99% fault revealing (99% of the tests killing them find the introduced fault).
❑ Mutants M3 and M4 are not relevant (M3 is equivalent).

Figure 4.2: Mutation testing in a fault introducing commit. The fault is triggered by the call to 'puts(p)', which automatically uses '\n' as the first separator, resulting in not using the user specified separator when this is a single character other than '\n'. This makes every test executing *seq_fast* with a separator other than '\n' to reveal the fault. Killing M_1 or M_2 can result in such tests, while killing M_4 does not (to kill M_4 a test must avoid executing line 547, which means that *seq_fast* is either not called or its call does not print anything, hence not making any observable difference). M_3 is equivalent.

However, M_1 is relevant to the commit. An execution of the test "seq -s, 1 2", which sets the separator to the comma (','), outputs "1,2\n" in pre-commit M_1 (*print_numbers* is called), "2," in post-commit M_1 ('puts(p)' is deleted and *seq_fast* is called), and "1\n2," in the post-commit original version (the first number is printed using 'puts(p)', which appends an '\n'). Similarly, M_2 is relevant to the commit. The execution of same test "seq -s, 1 2" outputs "1,2\n" in pre-commit M_2 (*print_numbers* is called), "1\n2" in post-commit M_1 (no comma separator printed and *seq_fast* is called).

Moreover, a fault introduced by the commit makes the program use '\n' instead of the user specified separator, after printing the first number, when the user separator is a single character other than '\n'. This happens because in such scenario, the program calls *seq_fast*, which calls 'puts(p)' (line 414) to print the first number. This automatically add an extra '\n' and do not use the specified separator.

Every test that executes *seq_fast*, with a separator other than '\n' reveal the fault. These are $(1 - \frac{2}{257}) \approx 99.2\%$ of all the tests that successfully execute *seq_fast*. The reason is that the separator is either not set in the test (defaults to '\n'), or set to one of the 256 '8-bits' characters (including '\n'). We observe that all tests that successfully execute *seq_fast* kill M_1 and M_2. Therefore, 99.2% of the tests that kill M_1 and M_2 reveal the fault.

4.3 Approach

We aim at testing commits using commit-relevant mutants; the subset of mutants on the post-commit program version that has a behaviour relevance to the committed changes.

We develop *MuDelta*, a technique that learns to rank mutants according to their commit-relevance potential (likelihood to be commit-relevant). Initially, *MuDelta* applies supervised learning on a mutant corpus from past data, and builds a prediction model. This model is then applied to predict the mutants that should be used to test the future commits of the program under test. This means that at commit time, testers can use and focus only on the most relevant mutants. This process is depicted in Figure 4.1.

4.3.1 *MuDelta* Feature Engineering

The mutant selection process in *MuDelta* is based on training of a predictor that is capable of identifying whether a mutant is commit-relevant with a certain confidence (probability). Consequently, we design a set of features to reflect specific code properties which may discriminate a commit-relevant mutant from another.

The study of Chekam et al. [34] found that fault revealing and killability mutant characteristics can be captured by simple code features. Therefore, we consider the features that they proposed in our machine learning model. Unfortunately, these features do not capture the interaction between mutants and the altered code. Hence, we design additional features capable of capturing the link between the mutant and the altered code (by the commit). These features also aim at capturing the characteristics of the altered code.

In the following subsections we describe the features we use in order to train a classifier. We consider a commit modification C associated with code statements $SC = \{S_{C_1}, S_{C_2}, ..., S_{C_n}\}$, and let $BC = \{B_{C_1}, B_{C_2}, ..., B_{C_k}\}$ the control-flow graph (CFG) basic blocks associated to the statements SC. Let us also consider a mutant M associated to a code statement S_M on which the mutation was applied. Let B_M be the CFG basic block associated to a mutated statement S_M containing the mutated expression E_M.

4.3.2 Contextual Features

In order to capture contextual information for each program statement, within a program version, we design features that leverage graph analysis technologies. We construct graph representations of the program, where the nodes are the statements of the program, and the edges are various types of relationships between statements. We consider the following four relationships (edge types): data dependency (direct data dependency, indirect data dependency) [33], control dependency, and control flow. Direct data dependency refers to variable value dependency, while indirect data dependency refers to pointer dereference value dependency (the data is accessed through dereferencing a pointer). In total we use the following 6 different graph representations, i.e., **1)** *Utility Graph* (UG) that includes all four edge types we discussed, **2)** *Dependency Graph* (DG) that includes all three dependency edges types, **3)** *Direct Data Dependency Graph* (DDDG) that includes only the direct data dependency edge type, **4)** *Indirect Data Dependency Graph* (IDDG) that includes only the indirect data dependency edge type, **5)** *Control Dependency Graph* (CDG) that includes only the control dependency edge type, and **6)** *Control Flow Graph* (CFG) which includes only the control flow edge type.

For each graph, we leverage graph analysis algorithms to compute a score for each node. We consider the following graph analysis algorithms: Rich-Club coefficient (RCC) [143, 147], Clustering coefficient (CC) [52, 157, 190], Square Clustering coefficient (SCC) [126], PageRank (PR) [160], and Hits Analysis (HA) [104].

Overall, we get a set of features F_S, for each statement S and for each graph G, by computing the score of the node corresponding to S, using all graph analysis algorithms on G. This gives us 6 * 5 (graphs * Metrics) features per program statement.

Complexity: Complexity of S_M, approximated by the number of mutants on S_M.
CfgDepth: Depth of B_M according to CFG.
CfgPredNum: Number of predecessor basic blocks, in CFG, of B_M.
CfgSuccNum: Number of successors basic blocks, in CFG, of B_M.
AstNumParents: Number of AST parents of E_M.
NumOutDataDeps: Number of mutants on expressions data-dependent on E_M.
NumInDataDeps: Number of mutants on expressions that E_M is data-dependent.
NumOutCtrlDeps: Number of mutants on statements control-dependents on E_M.
NumInCtrlDeps: Number of mutants on expressions that E_M is control-dependent
NumTieDeps: Number of mutants on E_M.
AstParentsNumOutDataDeps: Number of mutants on expressions data-dependent on E_M's AST parent statement.
AstParentsNumInDataDeps: Number of mutants on expressions that E_M's AST parent expression is data-dependent.
AstParentsNumOutCtrlDeps: Number of mutants on statements control-dependent on E_M's AST parent expression.
AstParentsNumInCtrlDeps: Number of mutants on expressions that E_M's AST parent expression is control-dependent.
AstParentsNumTieDeps: Number of mutants on E_M's AST parent expression.
TypeAstParent: Expression type of AST parent expressions of E_M.
TypeMutant: Mutant type of M, transformation rule. E.g., $a + b \rightarrow a - b$.
AstChildHasIdentifier: AST child of expression E_M has an identifier.
AstChildHasLiteral: AST child of expression E_M has a literal.
AstChildHasOperator: AST child of expression E_M has an operator.
OutDataDepNumStmtBB: Number of CFG basic blocks containing an expression data-dependent on S_M.
InDataDepNumStmtBB: Number of CFG basic blocks containing an expression on which S_M is data-dependent.
OutCtrlDepNumStmtBB: Number of CFG basic blocks containing an expression control-dependent on S_M.
InCtrlDepNumStmtBB: Number of CFG basic blocks containing an expression on which S_M is control-dependent.
AstParentMutantTypeNum: Number of each mutant type of E_M's AST parents.
OutDataDepMutantTypeNum: Number of each mutant type on expressions data-dependents on E_M.
InDataDepMutantTypeNum: Number of each mutant type on expressions on which E_M is data-dependent.
OutCtrlDepMutantTypeNum: Number of each mutant type on statements control-dependents on E_M.
InCtrlDepMutantTypeNum: Number of each mutant type on expressions on which E_M is control-dependent.

Figure 4.3: Mutant utility features

.

4.3.3 Mutant utility features

We used the features proposed by Chekam et al. [34]. These features relate to the complexity of the mutated statement S_M, the position of S_M in the control-flow graph, the dependencies with other mutants, and the nature of the code block B_M where S_M is located. The selected features are recorded in Figure 4.3. Note that for this study, we added the last 9 features (marked in the figure with italic), and the contextual features of S_M (Section 4.3.2). The first 4 features (with italic) are similar to the features *NumOutDataDeps, NumInDataDeps, NumOutCtrlDeps, NumInCtrlDeps* used by Chekam et al. [34], but, instead of the number of mutants, they count the number of basic blocks.

4.3.4 Mutant-Modification Interaction Features

To capture the interaction between mutant and altered code, we use features related to the information flow that the altered code C incur to the execution of mutant M. In this regard, we propose features that characterize the altered code and features that capture the information flow between C and M.

4.3.4.1 Modification Characteristics Features

We have features extracted from the commit diff and features extracted from the changed or added statements in the post-commit version of the program. Figure 4.4 describes the features extracted from the commit diff. The features extracted from the changed or added statements are: (a) The mean of the depth, according to CFG, of the basic blocks in BC (*modificationCfgDepth*). (b) The mean of the complexity of the statements in SC (*modificationComplexity*). (c) The contextual features (see Section 4.3.2) of the added or changed statements in the program. When the modification involves multiple statements, the mean of each feature value for all statements is computed.

NumConditional: Number of conditional statements in the modification.
NumHunks: Number of hunks (blocks) in the commit diff.
HasExit: The modification involves program termination commands.
ChangesCondition: The modification involves the condition of an *if* or a loop.
InvolesOutput: The modification involves a function call to *printf* or *error*.
IsRefactoring: The modification only does code refactoring.
NumUPDATE: Number of UPDATE operations from GumTree tool [53].
NumINSERT: Number of INERT operations from GumTree tool [53].
NumMOVE: Number of MOVE operations from GumTree tool [53].
NumDELETE: Number of DELETE operations from GumTree tool [53].
NumActionClusters: Number of action clusters from GumTree tool [53].
NumActions: Number of actions from GumTree tool [53].

ModificationCfgDepth:The mean of the depth according to CFG
ModificationComplexity: The mean of the complexity of the statements
Delta contextual features: The contextual features (see Section 4.3.2) of the added or changed statements in the program

Figure 4.4: Mutant-Modification Interaction Features

4.3.4.2 Information-flow Features

The first feature that we use, in this category, is a Boolean variable (*MutantOnModification*) that represents whether the mutant M mutates an altered code ($S_M \in SC$). Additionally, we consider the 6 graphs presented in section 4.3.2, and compute, for each graph, the set of shortest paths between S_M and SC.

For every set of paths, we compute the size (*NumPaths*), the maximum path length (*MaxPathLen*), minimum path length (*MinPathLen*) and mean path length (*MeanPathLen*). Our features are thus, the combination of each one of these metrics on every shortest path set.

4.3.5 Implementation

We implemented *MuDelta* in Python. For learning, we used stochastic gradient boosting [61] (decision trees), which has been found to work well in the context of mutation [34]. We used the XGBoost [39] framework and set the number of trees to 3,000 with a maximum trees depth to 10. We adopt early stopping during training to avoid over-fitting.

MuDelta uses both numerical or categorical features. The categorical features are: *TypeAstParent, TypeMutant*. In order to use the feature values with XGBoost, we pre-process them using a normalization of numerical and an encoding of categorical features. We normalize numerical features, between 0 and 1 using *Rescaling* (also known as min-max normalization).

We use *binary encoding* (binary encoding helps to keep a reasonably low feature dimension, when comparing to one-hot-encoding) for the categorical features. We also use NetworkX[2] in the graph representation in order to extract the contextual features that were described in section 4.3.2.

4.4 Research Questions

We start our analysis by investigating the prediction ability of our machine learning method. Thus, our first research question can be stated as:

RQ1 *(Prediction performance):* How well does *MuDelta* predict commit relevant mutants?

To answer this question we collect a set of commits from the subject programs where we apply mutation testing and identify relevant mutants. Then, we split the commits into training/validation (80% of the commits) and test sets (20% of the commits) based on the timeline of the project(older commits are used for training and newer for commits are used for evaluation), and perform our experiment.

[2]https://networkx.github.io/

After checking the performance of the predictions, we turn our attention to the primary problem of interest; mutant ranking. We investigate the extent to which our predictions can lead to strong and relevant tests (by using the predictive mutants as test objectives) in contrast to baseline mutants, i.e., randomly sampled mutants among those residing in the changed components (*Random*) or among those residing on the altered lines (*Modification*). Hence we ask:

RQ2 *(Test assessment):* How *MuDelta* compare with the baseline mutant sets with respect to killing commit-relevant mutants?

We answer this question following a simulation of a testing scenario where a tester analyse mutants in order to generate tests [9, 112]. We are interested in the relative differences between the subsumming relevant mutation score, denoted as $rMS*$, when test generation is guided by the predicted or the baseline mutants. We use the subsumming relevant mutation score to avoid bias from trivial/redundant mutants [163]. We also use the random mutant selection baseline since it performs comparably to the state-of-the-art [34, 73, 112]. We compare with random on a best effort basis, i.e., the $rMS*$ achieved by putting the same level of effort, measured by the number of mutants that require analysis. Such a simulation is typical in mutation testing literature [100, 112] and aims at quantifying the benefit of one method over the other. To further show the need for mutant selection out of the changed code, we also compute the extend to which mutants on modification are sufficient in killing commit-relevant mutants.

Answering the above question provides evidence that using our approach yields significant advantages over the baselines. While this is important and demonstrates the potential of our approach, still the question of actual test effectiveness (actual fault revelation) remains. This means that it remains unclear what the fault revelation potential of our approach when the commit is fault-introducing. Therefore, we seek to investigate:

RQ3 *(Fault Revelation):* How *MuDelta* compare with the baseline mutant sets with respect to (commit-introduced) fault revelation?

To answer this question, we investigate the fault revelation potential of the mutant selection techniques based on a set of real fault-introducing commits. We follow the same procedure as in the previous research questions.

4.5 Experimental Setup

4.5.1 Benchmarks Used

We selected C programs from the GNU Coreutils[3], a collection of text, file and shell utility programs widely used in software testing research [23, 29, 107]. The whole code-base of Coreutils comprises approximately 60,000 lines of C code[4]. To perform our study on commits we used the benchmark[5] introduced by Chapter 3 that is composed of two parts and includes *Benchmark-1*, a set of commits mined from the Coreutils' Github repository from year 2012 to 2019 and *CoREBench* [23] that has fault introducing commits.

The benchmark contains a) mutants generated by Mart [36], a state-of-the-art tool that supports a comprehensive set of mutation operators and TCE[6] [100, 164] on both pre- and post-commit program versions of each commit, b) the mutant labels (whether they are commit-relevant), and c) large test pools created using a combination of test generation tools [29, 35, 107]. It is noted that the mutant test executions involved require excessive computational resources, i.e., require roughly 100 weeks

[3]https://www.gnu.org/software/coreutils/

[4]Measured with cloc (http://cloc.sourceforge.net/)

[5]https://github.com/relevantMutationTesting

[6]Compiler-based equivalent and duplicate mutant detection technique

Table 4.1: Test Subjects

Benchmark	#Programs	#Commits	#Mutants	#Relevant	#Tests
CoREBench	6	13	154,396	21,597	8,828
Benchmark-1	17	37	412,060	65,982	14,785

of computation. Details about the data we used are recorded in Table 4.1. The column *#Relevant* records the number of commit-relevant mutants.

4.5.2 Experimental Procedure

To account for our working scenario, we always train according to time, i.e, we use the older commits for training and the newer for evaluation. This ensured that we follow the historical order of the commits.

Following the stated RQs, our experiment is composed of three parts. The first part evaluates the prediction ability (performance) of *MuDelta*, answering RQ1. The second at evaluating the ability of *MuDelta* to rank commit-relevant mutants, answering RQ2, and the third part at evaluating the fault revealing potential, answering RQ3.

First experimental part: We evaluate the trained classifiers using five typically adopted metrics, namely, the Area Under the Receiver Operating Characteristic Curve (ROC-AUC), the Area Under the Precision-Recall Curve (PR-AUC), the precision, the recall and the F1-score.

The Receiver Operating Characteristic (ROC) curve records the relationship between true and false positive rates [246]. The Precision-Recall (PR) Curve records the decrease in true positive classifications when the predicted positive values increase. In essence, the PR curve shows the trade-off between precision and recall [246].

Precision is defined as the number of items that are truly relevant among the items that predicted to be relevant. *Recall* is defined as the number of items that are predicted to be relevant among all the truly relevant ones. The F1-score or F-measure of a classifier is defined as the weighted harmonic mean of the precision and recall. These assessment metrics measure the general classification accuracy of the classifier. Higher values denote a better classification.

To reduce the risk of over-fitting, we split our commit data into three mutually exclusive sets (training, validation and test data). We also use early stopping during training to overwhelm over-fitting. We use the following procedure:

1. Chronologically order the commit (from older to newer).
2. Select the newest 20% of commits as test data.
3. Randomly shuffle all the mutants from the remaining 80% of commits (oldest commit), then, select 20% of them as validation data and the rest as training data.

Thus, the training, validation and test data represent 64%, 16% and 20% of the data-set, respectively. The model evaluation is performed on the test data. This experiment part was performed on both CoREBench and *Benchmark-1.*

Second experimental part: We simulate a scenario where a tester selects mutants and designs tests to kill them. This typical procedure [34, 38, 112, 151] consists of randomly selecting test cases, from the test pools of the benchmark, that kill the selected mutants. Specifically, we rank the mutants and then we follow the mutant order by picking test cases, from the test pool, that kill them. We then remove all the killed mutants and pick the next mutant from the list. If the mutant is not killed by any of the tests, we discard it without selecting any test. We repeat this process 100 times for all the approaches. *MuDelta* ranks all the mutants by the predicted commit-relevance probability, *Random* randomly ranks all the mutants in the changed components, and *Modification* randomly ranks the mutants located on the altered code.

Our effectiveness metrics are the relevant subsuming mutation score ($rMS*$) achieved by the test suites when analysing up to a certain number of mutants. Subsuming score metrics allows reducing the influence of redundant mutants [111, 162, 163]. We also compute the Average Percentage of Faults Detected (APFD) [82] that represents the average relevant subsuming mutation score when analysing any number of mutants within a given range.

Our effort metric is the number of mutants picked (analysed by the tester). This includes the mutants, killable or not, that should be presented to testers for analysis (either design a test to kill them or judge them as equivalent) when applying mutation testing [100, 112]. In the spirit of the best-effort evaluation, we focus on few mutants (up to 100) that testers need to analyse. This evaluation aims at showing the benefits of *MuDelta* over *Random* under the same relative testing effort. The contrast with the *Modification* shows whether there is a need for mutant selection outside of the modified code, i.e., whether mutants on modification are sufficient leading to tests that kill commit-relevant mutants. This part of the experiment was performed on both CoREBench and *Benchmark-1*.

Third experimental part: To evaluate the fault revealing ability of *MuDelta*, we used the CoREBench commits. We adopted a chronological ordering for training, validation and testing when splitting the commits similar to what we did in previous experimental parts. We use the same process and effort metric as in the the second part of the experiment and report results related to fault revelation and the average percentage of commit-introduced faults revealed (APFD) within the range, 1-100, of analysed mutants.

To account for the stochastic selection of test cases and mutant ranking, we used the Wilcoxon test to determine whether there is a statistically significant difference between the studied methods. To check the size of the differences we used the Vargha Delaney effect size $\hat{A}_{12}$ [210], which quantifies the differences between the approaches. A value $\hat{A}_{12} = 0.5$ suggests that the data of the two samples tend to be the same. Values $\hat{A}_{12} > 0.5$ indicate that the first data-set has higher values, while values $\hat{A}_{12} < 0.5$ indicate the opposite.

4.6 Results

4.6.1 Assessment of the Prediction Performance (RQ1)

To evaluate the performance of *MuDelta*, we check the model's convergence. During training and after each iteration of the training process, we check the model performance on both the training and validation data we used for training. Figure 4.5 shows the ROC-AUC and PR-AUC values wrt the number of training iterations. We observe that the model performance on both the training and validation data increase with the number of iteration and stabilizes at specific values, suggesting that our model is able to learn the characteristics of commit-relevant mutants.

We then evaluate the performance of our model to predict commit-relevant mutants on the future commits that appear in the test set. To compute the precision, recall and F1-score, we set the prediction threshold probability to 0.1, which we obtained by applying the geometric mean [12, 106] on the validation dataset. The precision, recall and F1-score of our classifier are 0.63, 0.32 and 0.42, respectively. These values are higher than those that one can get with a random classifier (0.21, 0.21 and 0.20, respectively). Figure 4.6 shows the ROC and PR curves of our classifier (strong lines) and a random classifier (dashed lines). We observe that the ROC-AUC of our classifier is 0.80 indicating a strong prediction ability. Similarly, we see that the PR-AUC of our classifier is 0.50 while the random classifier PR-AUC is 0.20.

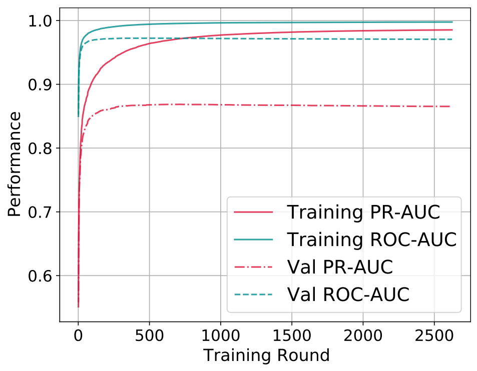


Figure 4.5: Training and Validation Curves from the Training phase.

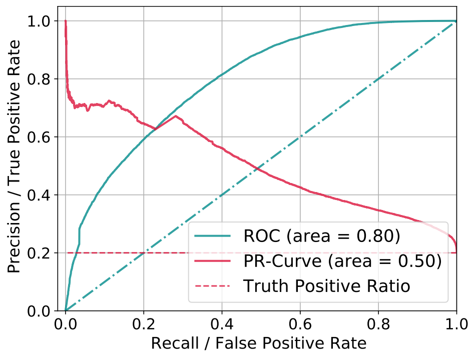


Figure 4.6: Precision-Recall and ROC Curves on test data.

In this context [174] it is important to give few mutants to developers for analysis. To evaluate the performance of *MuDelta* with lower thresholds, we also study the performance of *MuDelta* with thresholds ranging from the 10 to 100 mutants. We observe that the median precision of *MuDelta* ranges from 0.76 to 0.90 when the threshold goes from 10 to 30 mutants. These values are significantly higher than the random classifier, which has a precision of 0.15.

These results provide evidence that *MuDelta* provides a good discriminative ability for assessing the utility of mutants to test particular code changes.

4.6.2 Mutant Ranking for Tests Assessment (RQ2)

Figure 4.7 shows the median $rMS*$ achieved by the mutant ranking strategies, when the number of analysed mutant budget range from 1 to 100 mutants. In other words, the figure shows test effectiveness (measured with $rMS*$, y-axis) that is achieved by a developer when analysing a number of mutants, representing the cost factor (recorded in x-axis). Each sub-figure is a commit taken from the test data. We observe that the curve for *MuDelta* is always higher than the curves of *Random* and *Modification*, and *Random* is above *Modification*.

To further visualize the differences, Figure 4.8 shows the distribution of the $rMS*$ of the mutant ranking strategies for budget thresholds 10, 30, 50 and 100 mutants. As can be seen from the plots, *MuDelta* outperforms both *Random* and *Modification*. Interestingly, *Random* outperforms *Modification*. With threshold 10 mutants, the difference of the median values is 22% and 26% for *Random* and *Modification*, respectively. This difference is markedly increased when analysing more mutants, i.e., it becomes 45% and 50% for the thresholds of 30 and 50 mutants, for *Random*.

To check whether the differences are statistically significant we performed a Wilcoxon rank-sum test and computed the Vargha Delaney $\hat{A}_{12}$ effect size and found that *MuDelta* outperforms both *Random*

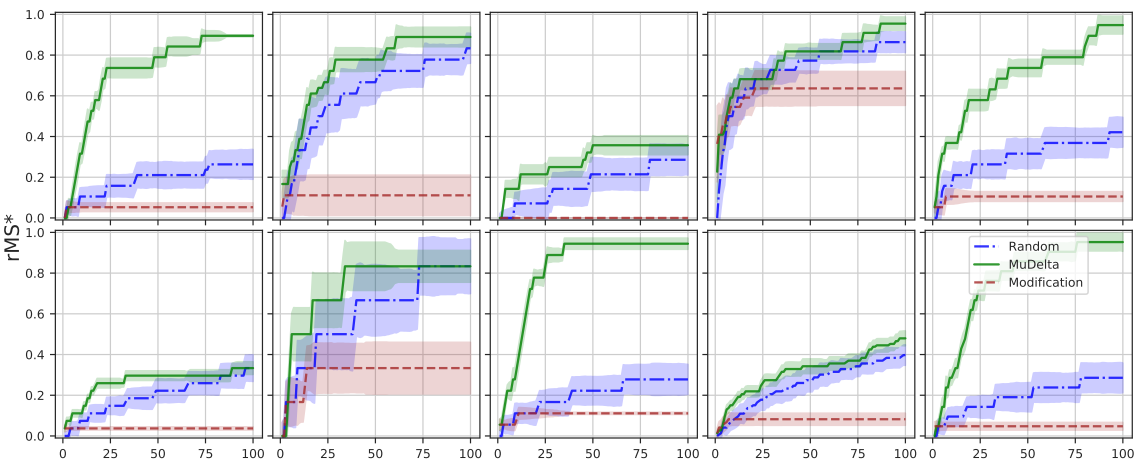


Figure 4.7: $rMS*$ achieved when analysing up to 100 mutants.

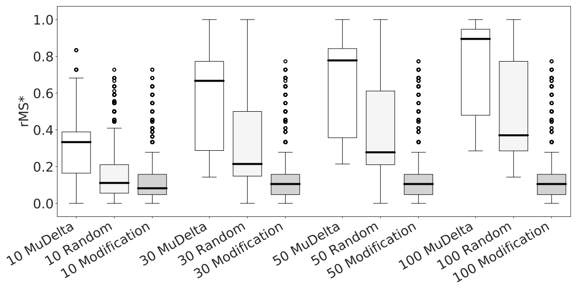


Figure 4.8: $rMS*$ values when analysing up to 10, 30, 50 and 100 mutants.

and *Modification* with statistically significant difference (at 0.01 significant level). *Random* has also statistically significant differences with *Modification*.

Figure 4.9 shows the Vargha Delaney $\hat{A}_{12}$ values between *MuDelta* and both *Random* and *Modification*. We observe that the median value is between 77% and 83% for threshold between 10 and 100 mutants, for *Random*. Suggesting that *MuDelta* is better than *Random* in 77% to 83% of the cases for these thresholds. The differences are larger for *Modification*.

We further validate our approach by considering the distributions of APFD (Average Percentage of Faults Detected) values for all possible thresholds (for 1-100 mutants). Figure 4.10 depicts these results and shows that *MuDelta* yields an APFD median of 71%, *Random* and *Modification* reach median APFD values of 26% and 11% respectively, confirm the superiority of our approach.

To account for the stochastic nature of the compared approaches and increase the confidence on our results, we further perform a statistical test on the APFD values. The Wilcoxon test results yielded p-values much lower than our significance level for the compared data, i.e., samples of *MuDelta* and *Random*, *MuDelta* and *Modification*, *Random* and *Modification*, respectively. Therefore, we conclude that *MuDelta* outperforms *Random* with statistically significance, while *Modification* is not sufficient for testing the deltas.

4.6.3 Mutant Ranking and Fault Revelation (RQ3)

Figure 4.11 shows the distributions of APFD (Average Percentage of Faults Detected) values for the CoREBench fault introducing test commits, using the three approaches under evaluation. While *MuDelta* yields an APFD median of 52%, *Random* and *Modification* reach median APFD values

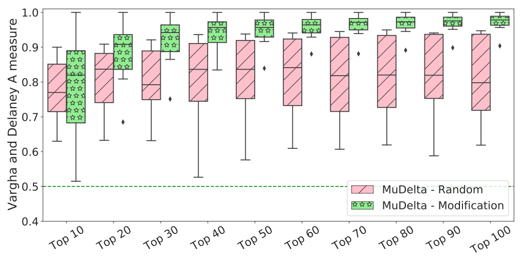


Figure 4.9: Vargha and Deianey $\hat{A}_{12}$ (*MuDelta* VS Random, *MuDelta* VS Modification) about $rMS*$

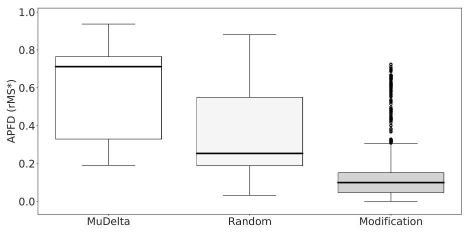


Figure 4.10: APFD $rMS*$ (up to 100 mutants).

of 25% and 0% respectively. The improvement over *Random* and *Modification* are 27% and 52%, respectively. These results confirm the superiority of our approach wrt to fault revelation.

The Wilcoxon test yielded p-values much lower than our significance level for the compared data, i.e., samples of *MuDelta* and *Random*, *MuDelta* and *Modification*, *Random* and *Modification*. Therefore, we conclude that *MuDelta* outperforms *Random* and *Modification* with statistically significance while *Random* outperforms *Modification*.

Figure 4.12 shows the distribution of fault revelation for the ranking strategies and for mutant set size thresholds up to 100 mutants. We observe that the curve for *MuDelta* is above the curves of *random* and *Modification*, and *Random* is above *Modification*. Specifically, we observe that *MuDelta* reaches a fault revelation of 60% and 100% when analysing the top 30 and 61 mutants, while *Random* 7% and 12%, respectively.

4.7 Discussion

4.7.1 Comparison with other models

To further assess the effectiveness of our model, we contrast it with the prediction ability of five other models (on the same training, validation and test data-sets) that are typically used in prediction modelling studies. In particular, we used three families of models (Ensemble model classifiers, Logistic classifiers and Neural Networks) and built five models; namely Adaboost, Random Forest, Logistic Regression, Multilayer Perceptron (MLP) and Mixed MLP. MLP and Mixed MLP were inspired by the work of Li et al. [121], their architecture is shown in Figure 4.13 and 4.14. To train and evaluate the models we used the Sklearn library[7]. Since our data are imbalanced we also used class weighting strategies that are commonly used to tackle this issue. To avoid bias from improper setting of the learners, in all the cases we used Grid Search & Cross Validation on the validation set to tune our hyperparameters.

Table 4.2 reports the ROC-AUC, PR-AUC, MCC, and precision on top 100 ranked mutants of the prediction results of all different learners we built. The results show that the XGBoost model, that

[7]`https://scikit-learn.org/stable/`

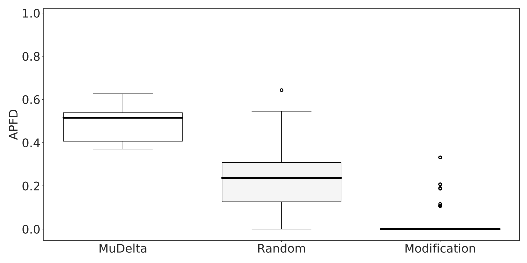


Figure 4.11: APFD Fault-revelation (up to 100 mutants).

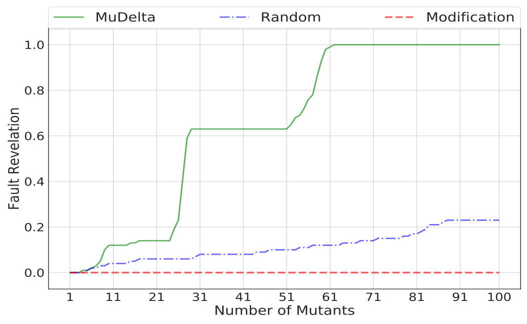


Figure 4.12: Median fault-revelation in fault introducing commits.

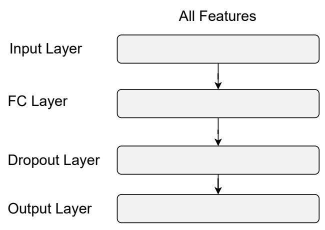


Figure 4.13: MLP - Neural Network Architecture

Table 4.2: Model Comparison

AdaBoost	0.6	0.35	0.26	0.55
Random Forest	0.66	0.31	0.24	0.57
Logistic	0.58	0.26	0.13	0.21
MLP	0.51	0.19	0.1	0.2
Mixed MLP	0.68	0.45	0.31	0.2
XGBoost	*0.80*	*0.50*	*0.36*	*0.61*

we use, perform best in all cases. The general prediction metrics (ROC-AUC,PR-AUC, MCC) show that Mixed MLP model is the second best case though it falls behind the Ensemble models wrt to the top-100 mutants. Nevertheless, the results provide clear indications that the XGBoost model we use is indeed the best choice.

4.7.2 Feature Importance

To evaluate the importance of our features we used the SHapley Additive exPlanations (SHAP)[8] method [129], i.e., a game theory method that explains individual predictions based on the game theoretically optimal Shapley Values. In particular, we aim at explaining our predictions by assuming that each feature value we use is a "player" in a game where the prediction is the payout. Shapley

[8]https://github.com/slundberg/shap

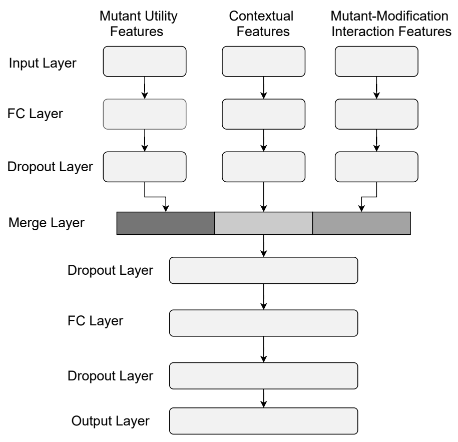


Figure 4.14: Mixed MLP - Neural Network Architecture

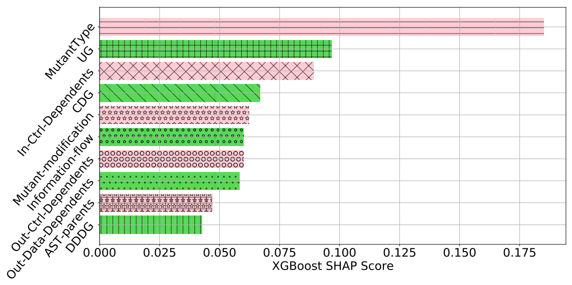


Figure 4.15: Feature Importance, SHAP Score of top-10 feature sets. The 10 most important features are "Mutant Type", "Utility Graph", "Incoming Control Dependencies", "Mutant-modification features", "Control Dependency Graph", "Information-flow", "Outgoing Control Dependencies", "Outgoing Data Dependencies","AST parents" and "Directed Data Dependency Graph".

values – a method from coalitional game theory – tells us how to fairly distribute the "payout" among the features. We thus measure and report the feature importance (Shapley values) of the feature categories we use. Results are depicted on Figures 4.15 and show that "Mutant Type", "Utility Graph", "Incoming Control Dependencies", "Mutant-modification features", "Control Dependency Graph" and the "Information-flow" are the top 6 feature sets and that all three types of features we use are important. Additional results related to the feature importance of the individual features we used can be found on the accompanied website.

4.8 Threats to Validity

A possible threat to external validity could be due to our test subjects. Our target was commits that do not alter test contracts and make small modifications, similar to those observed in industrial CI pipelines. Such commits are usually hard to test and typically result in subtle faults. Large commits that add new features, should be anyway tested by using a mutation testing approach that involves (almost) all the relevant mutants residing on the added code. To reduce this threat, we sampled a commit set where we could reasonably perform our experiments. At the same time, to diminish potential selection bias, we also used the Coreutils commits of CoREBench [23], which are frequently used in testing studies.

We are confident on our results since the relevance properties of the mutants reside on the context of the committed code, which includes the area around the dependencies to the committed code (where we draw our feature values), that is small and its characteristics should be as representative as our subjects. Moreover, our predictions converge well, do not have significant variance wrt to the baselines and consistently outperform the baselines in all test subjects we used. Additionally, the statistical significance we observe indicates the sufficiency of our data analysis [11]. Future work should validate our findings and analysis to larger programs.

Another threat may relate to the mutants we use. To mitigate this threat, we selected data from a mutation testing tool [36] that has been used in several studies [34, 35, 136] that supports the most commonly used operators [114] and covers the most frequent features of the C language.

Threats to internal validity may be due our features. We use a large number of features, selected either based on previous studies [34] or by using our intuition, which are automatically filtered by gradient boosting. To further reduce this concern, we split our data in three parts, training, validation and test data. During training (using training data) we measure the model convergence on training and validation data. As demonstrated in Figure 4.5, our model converges both on the training and validation data, showing that there are low chances for over- or under-fitting because in these cases, the model would not converge on the validation data.

The test-based approximation of relevant and killable mutants may introduce additional threats. To reduce it, we used test suites generated by KLEE [107] and SeMu [35], together with developer test suites.

A possible threat to construct validity could be due to the effort metric, i.e., the number of analysed mutants, we use. This is a typical metric for this kind of studies [112] aiming at capturing the manual effort involved when analysing mutants or asserting automatically generated tests. Since, our data have been filtered by TCE [100, 164], a state-of-the-art equivalent mutant detection technique, this threat should be limited.

Overall, we tried to reduce threats by using various evaluation metrics, i.e., prediction performance, relevant mutation score and fault revelation, and established procedures. Furthermore, to enable replication and future research we will make our tools and data publicly available.

4.9 Related Work

The problem of determining the set of mutants that are most relevant to particular code changes might resemble a dependence analysis problem. One natural solution involves forming a program slice on the set of changed statements. Any mutant that lies in the slice should be considered relevant. Unfortunately, this approach does not scale well for several reasons. Firstly, as have been previously observed [20, 21], even a single static slice of a program tends to occupy between one and two thirds of the program from which it is constructed. Therefore, the union of a set of such slices, will be large, and thereby fail to exclude many mutants. Secondly, the dependence analysis would need to be incremental, which raises further challenges. Although there have been incremental dependence analyses in the literature [158], many well-developed slicing systems are not incremental. In general, the problem of incremental program analysis at scale remains challenging [80]. Thirdly, it is hard to use dependence analysis to provide the priority ordering we need, where priority is based on degree of relevance. Potentially, unions of dynamic slices or some form of observation-based slicing [18] could achieve this, but such approaches have a prohibitive computational cost in comparison to our method.

Change impact analysis [118] aims at determining the effects of changes on the other parts of the software. Similar to program slicing, such approaches are conservative, therefore they result in large number of false positives, does not account for equivalent mutants located on potentially infected code and is hard to provide the mutant ranking (prioritizes mutant types and location) we need. Other attempts aim at testing the potential propagation flows of the changes [10, 187, 188, 189]. Similarly to change impact analysis their purpose is to identify the program paths (flows) that may be impacted by the changes. They rely on symbolic execution to check for the feasibility of the flows, form test requirements (conditions to be fulfilled) and decide on relevance. Unfortunately, such techniques inherit most of the issues of symbolic execution, are complex to implement and test the propagation of the changes. In contrast our technique scales since it relies on static code features, does not require any complex analysis techniques and applies mutation testing that is known for capturing the fault-revealing properties of test suites [34, 38].

Automatic test case generation aims at producing test inputs that a) make observable the code differences of two program versions [180], b) increase and optimize coverage [230] and kill mutants [35, 60, 196]. Among these techniques, the most relevant to our study are the are the ones related to patch testing, i.e., differential symbolic execution [172], KATCH [141] and Shadow symbolic execution [107]. These techniques generate tests exercising the semantic differences between program versions guided by coverage. All these techniques do not propose any test requirements as done by *MuDelta* and thus, they are complementary to our goal. This means that they can be used to generate tests to kill the commit-relevant mutants proposed by *MuDelta*.

Related to continuous integration, Google [174] is using a mutation testing tool that is integrated with the code review process (reviewers select mutants). This tool proposes mutants to developers in order to design test cases. The key basis of this approach is to choose some mutants from the lines of the altered code. We share a similar intent, though we aim at making an informative selection of mutants among all project mutants. According to our results mutants residing on non-altered code tend to be powerful at capturing the interactions between the altered and non-altered code.

Regression mutation testing [240] and the predictive mutation testing [139, 236] also focus on regression testing. Similarly, Pitest [41], a popular mutation testing tool, implements an incremental analysis that computes which mutants are killed or not by a regression test suite. This means that the goal of the above techniques is to estimate the mutation score achieved by regression test suites thereby not making any distinction between commit-relevant and non-relevant mutants, not making any mutant ranking and not proposing any live mutant to be used for test generation.

Fault revealing mutant selection [34] aims at selecting mutants that are likely to expose faults. While powerful, that technique targets the entire program functionality and not the changed/delta one. Since it is unaware of the deltas it selects many irrelevant mutants, while missing many delta-relevant mutants related to the delta-context interactions.

Perhaps the closest work to ours is the commit-aware mutation testing study [136] that defines the notion of mutant relevance and demonstrates its potential. In essence that work describes the fundamental aspects of relevant mutants but does not define any way to identify them at the testing time. We therefore built on top of this notion by providing a static technique that identifies relevant mutants.

Overall, there is a fundamental difference on the aims of our approach and previous research since we statically produce relevant, to code changes, mutants and rank them to provide a best effort testing application.

4.10 Conclusion

We presented *MuDelta* a delta-oriented mutation testing approach that selects delta-relevant mutants; mutants capturing the program behaviours affected by specific program changes. Experiments with *MuDelta* demonstrated that it identifies delta-relevant mutants with 0.63 and 0.32 precision and recall. Interestingly, killing these mutants leads to strong tests that kill 45% more relevant mutants than killing randomly selected mutants. Our results also show that *MuDelta* selects mutants with a 27% higher fault revealing ability than randomly selected mutants.

Acknowledgement

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C17/IS/11686509/CODEMATES. Mark Harman is part supported by European Research Council Advanced Fellowship grant number 741278; Evolutionary Program Improvement (EPIC).

5 GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses

Code embedding is a keystone in the application of machine learning on several Software Engineering (SE) tasks. To effectively support a plethora of SE tasks, the embedding needs to capture program syntax and semantics in a way that is *generic*. To this end, we propose the *first self-supervised pre-training* approach (called GraphCode2Vec) which produces task-agnostic embedding of lexical and program dependence features. GraphCode2Vec achieves this via a synergistic combination of *code analysis* and *Graph Neural Networks*. GraphCode2Vec is *generic*, it *allows pre-training*, and it is *applicable to several SE downstream tasks*. We evaluate the effectiveness of GraphCode2Vec on four (4) tasks (method name prediction, solution classification, mutation testing and overfitted patch classification), and compare it with four (4) similarly *generic* code embedding baselines (Code2Seq, Code2Vec, CodeBERT, GraphCodeBERT) and 7 *task-specific*, learning-based methods. In particular, GraphCode2Vec is more effective than both generic and task-specific learning-based baselines. It is also complementary and comparable to GraphCodeBERT (a larger and more complex model). We also demonstrate through a probing and ablation study that GraphCode2Vec learns lexical and program dependence features and that self-supervised pre-training improves effectiveness.

Contents

Figure 5.1: Motivating example showing (a) an original method (`LowerBound`), and two behaviorally equivalent clones of the original method, namely (b) a renamed method (`findLowerBound`), and (c) a refactored method (`getLowerBound`).

```java
public static int lowerBound(int[] array,
                int length, int value) {
        int low = 0;
        int high = length;
        while (low < high) {
            final int mid = (low + high) / 2;
            if (value <= array[mid]) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return low;
}
```

(a) Original Method

```java
public static int findLowerBound(int[] inputs,
                                int size, int v) {
        int bounder = 0;
        int l = size;
        int mindex = 0;
        while (bounder < l) {
            mindex = (bounder + l) / 2;
            if (v <= inputs[mindex]) {
                l = mindex;
            } else {
                bounder = mindex + 1;
            }
        }
        return bounder;
}
```

(b) Renamed Method

```java
public static int getLowerBound(int v,
                int size, int[] inputs) {
        int h = size;
        int mindex = 0;
        int check = 0;
        while (check < h) {
            mindex = (check + h) / 2;
            if (v > inputs[mindex]) {
                check = mindex + 1;
            } else {
                h = mindex;
            }
        }
        return check;
}
```

(c) Refactored Method

5.1 Introduction

Applying machine learning to address software engineering (SE) problems often requires a vector representation of the program code, especially for deep learning systems. A naïve representation, used in many SE applications, is one-hot encoding that represents every feature with a dedicated binary variable (a vector including binary values) [192]. However, this type of embedding is usually a high-dimensional sparse vector because the size of vocabulary is very large in practice, which results in the notorious *curse of dimensionality* problem [14]. Besides, one-hot encoding has *out-of-vocabulary* (OOV) problem, which decreases model generalization capability such that it cannot handle new type of data [208].

To deal with these issues, researchers use dense and reasonably concise vectors to encode program features for specific SE tasks, since they generalise better [92, 218, 222, 242]. More recently, researchers apply natural language processing (NLP) techniques to learn the universal code embedding vector for general SE tasks [3, 4, 5, 15, 26, 27, 43, 57, 76, 83, 95, 170, 179, 213, 219]. The resulting code embedding represents a mapping from the "program space" to the "latent space" that captures the different code-used semantics, i.e., the semantic similarities between program snippets. The aim is that similar programs should have similar representations in the latent space.

State-of-the-art code embedding approaches focus either on *syntactic features* (*i.e.*, tokens/AST), or on *semantic features* (*i.e.*, program dependencies) ignoring the importance of combining both features together. For example, Code2Vec [5] and CodeBERT [57]) focus on syntactic features, while PROGRAML [43] and NCC [15]) focus on program semantics. There are few studies using both program semantics and syntax, e.g., GraphCodeBERT [76]. However, these approaches are not *precise*, they do not obtain or embed the entire program dependence graph. Instead, they estimate program dependence via string matching (instead of static program analysis), then augment AST trees with sequential data flow edges.

To address these challenges, we propose the *first approach (called GraphCode2Vec) to synergistically capture syntactic and semantic program features with Graph Neural Network (GNN) via self-supervised pretraining.*

The *key idea* of our approach is to use *static program analysis* and *graph neural networks* to effectively represent programs in the latent space. This is achieved by combining lexical and program dependence analysis embeddings. During lexical embedding, GraphCode2Vec embeds the syntactic features in the latent space via *tokenization*. In addition, it performs dependence embedding to capture program semantics via static program analysis, it derives the program dependence graph (PDG) and represents it in the latent space using Graph Neural Networks (GNN). It then concatenates both lexical embedding and dependence embedding in the program's vector space. This allows GraphCode2Vec to be effective and applicable on several downstream tasks.

Table 5.1: Cosine Similarity of three behaviorally/semantically similar program pairs from our motivating example, using GraphCodeBERT, CodeBERT and GraphCode2Vec

Program Pairs	**Graph-CodeBERT**	**CodeBERT**	**GraphCode2Vec**
`searchLowerBound` & `lowerBound`	1	0.99	1
`findLowerBound` & `lowerBound`	0.70	0.61	0.99
`getLowerBound` & `lowerBound`	0.70	0.51	0.99
Average of 91 pairs	-0.05	-0.06	-0.03

To demonstrate the importance of semantic embedding, we compare the similarity of three pairs of programs using our approach, in comparison to a syntax-only embedding approach – CodeBERT, and GraphCodeBERT, which embeds both syntax and semantic, albeit without program dependence analysis. Consider the example of three program clones in Figure 5.1. This example includes three behaviorally or semantically equivalent programs, that have low syntactic similarity (i.e., different tokens), but with similar semantic features, i.e., program dependence graphs (PDGs). To measure the similarity distance in the latent space, in addition to the example code clones (Figure 5.1), we randomly select 10 other different code methods (from GitHub) without any change to establish a baseline for comparing all approaches. To this end, we compute the average cosine similarity distance for all 91 program pairs ($\frac{14\times 13}{2}$) for reference to show that all approaches report similar scores for all randomly selected 91 pairs (Table 5.1).[1] For all three approaches, the similarity between the "original program" and a direct copy of the program with only method name renaming to "searchLowerBound", is well captured with an almost perfect cosine similarity score for all approaches (1 or 0.99). Likewise, the cosine similarity of the original program and the "renamed" program (`findLowerBound`) is mostly well captured by all approaches, since they all embed program syntax, albeit with lower cosine similarity scores for CodeBERT (0.61) and GraphCodeBERT (0.70), in comparison to our approach (0.99).

Meanwhile, CodeBERT fails to capture the semantic similarity between the "original program" and the "refactored program" (`getLowerBound`), even though they are behaviorally similar and share similar program dependence. This is evidenced by the low cosine similarity score (0.51), because it does not account for semantic information in its embedding, especially the similar program dependence graph shared by both programs. Lastly, GraphCodeBERT performs slightly better than CodeBERT (0.70 vs. 0.51), but lower than our approach (0.99). This is due to lack of actual static program analysis in the embedding of GraphCodeBERT, since it only applies a heuristic (string matching) to estimate program dependence, it is *imprecise.* This example demonstrates the importance and necessity of embedding precise dependence information.

A key ingredient of GraphCode2Vec is *self-supervised pretraining. Even though task-specific learning based approaches (e.g., CNNSentence [156]) learn the vector representation of code without pre-training, they are non-generic and less effective. Applying their learned vector representation to other (SE) tasks requires re-tuning model parameters, and the lack of pretraining reflects in their performance. As an example, our evaluation (in RQ1 section 5.5) showed that our self-supervised pretraining approach improves effectiveness when compared to 7 task-specific approaches (i.e., without pretraining) addressing two (SE) tasks (solution classification and patch classification). To further demonstrate the importance of self-supervised pretraining, we compare the effectiveness of GraphCode2Vec with and without pretraining using two downstream tasks. Overall, we demonstrate that our self-supervised pretraining improves effectiveness by 28% (see RQ3).*

To evaluate GraphCode2Vec, we compare it to four generic code embedding approaches, and 7 task-specific learning-based applications. We also investigate the stability and learning ability of

[1]The purpose of computing the average cosine similarity of all 91 code pairs is to establish a meaningful reference for comparing embeddings and to serve as a sanity check. We expect the mean of the cosine similarity of a set of randomly selected pairs of code clones and non-clones to lie around zero for all approaches (range -1 to 1).

our approach through sensitivity, ablation and probing analyses. Overall, we make the following *contributions*:

Task-specific learning-based applications. We introduce the automatic application of GraphCode2Vec to solve specific downstream SE tasks, without extensive human intervention to adapt model architecture. In comparison to the state-of-the-art task-specific learning-based approaches (*e.g.*, ODS [232]), our approach does not require any effort to tune the hyper-parameters to be applicable to a downstream task (Section 5.3). Our evaluation on two downstream tasks, solution classification and patch classification, showed that GraphCode2Vec outperforms the state-of-the-art task-specific learning-based applications: For all tasks it outperforms all task-specific applications (RQ1 in Section 5.5).
Generic Code embedding. We propose a novel and generic code embedding learning approach (*i.e.,* GraphCode2Vec) that captures the lexical, control flow and data flow features of programs through a novel combination of *tokenization*, *static code analysis* and *graph neural networks* (GNNs). To the best of our knowledge, GraphCode2Vec is the first code embedding approach to precisely capture syntactic and semantic program features with GNNs via self-supervised pretraining. We demonstrate that *GraphCode2Vec is effective* (RQ2 in Section 5.5): *It outperforms all syntax-only generic code embedding baselines.* We provide our pre-trained models and generic embedding for public use and scrutiny.[2]
Further Analyses. We extensively evaluate the *stability* and *interpretability* of our approach by conducting *sensitivity*, *probing* and *ablation* analyses. We also investigate the impact of configuration choices (i.e., pre-training strategies and GNN architectures) on the effectiveness of our approach on downstream tasks. Our evaluation results show that GraphCode2Vec *effectively learns lexical and program dependence features*, it is *stable* and insensitive to the choice of GNN architecture or pre-training strategy (RQ3 in Section 5.5).[3]

5.2 Related Work

5.2.1 Generic code embedding

We introduce methods that learn general-purpose code representations to support several downstream tasks. These approaches are not designed for a specific task. There are three major types of generic code embedding approaches, namely *syntax-based*, *semantic-based* and *combined semantic and syntactic* approaches (*see* Table 5.2).

Syntax-based Generic Approaches: These approaches encode program snippets, either by dividing the program into *strings*, lexicalizing them into *tokens* or parsing the program into a *parse tree or abstract syntax tree (AST).* Syntax-only generic embedding approaches include Code2Vec [5], Code2Seq [4], CodeBERT [57], C-BERT [27], InferCode[26], CC2Vec [83], AST-based NN [234] and ProgHeteroGraph [219] (*see* Table 5.2). Notably, these approaches use neural models for representing code (snippets), *e.g.*, via code vector (*e.g.*, Code2Vec [5]), machine translation (*e.g.*, Code2Seq [4]) or transformers (*e.g.*, CodeBERT [57]). Code2Vec [5] is an AST-based code representation learning model that represents code snippets as single fixed-length code vector. It decomposes a program into a collection of paths using an AST and learns the atomic representation of each path while simultaneously learning how to aggregate the set of paths. Code2Seq [4] is an alternative code embedding approach that uses Sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), to encode code snippets. CodeBERT [57] is a bimodal pre-trained model for programming language (PL) and natural language (NL) tasks, which uses transformer-based neural

[2] `https://github.com/graphcode2vec/graphcode2vec`
[3] In the rest of this work, we interchangeably use the terms “lexical” and ”syntactic” interchangeably, as well as “(program) dependence” and “semantic”. Such that the terms “lexical embedding” and “syntactic embedding” refer to the embedding of program syntax, and the terms “dependency embedding” and “semantic embedding” refer to the embedding of program dependence information.

architecture to encode code snippets. Besides, CodeBERT [57], C-BERT [27] and Cu-BERT [95] are BERT-inspired approaches, these methods adopt similar methodologies to learn code representations as BERT [47].

GraphCode2Vec in Chapter 5 is similar to the aforementioned generic code embedding methods, it is also a general-purpose code embedding approach that captures syntax by lexicalizing the program into tokens (*see* Table 5.2). However, all of the aforementioned generic approaches are syntax-based, none of these approaches account for program semantics (i.e., data and control flow). Unlike these approaches, GraphCode2Vec additionally captures program semantics via static analysis. In this work, we compare our approach (GraphCode2Vec) to the three (3) most popular and recent syntax-based generic code embedding approaches, namely Code2Vec [5], Code2Seq [4] and CodeBERT [57] (*see* section 5.5).

Semantic-based Generic Approaches: This refers to code embedding methods that capture *only* semantic information such as control and data flow dependencies in the program. *Semantic-only generic approaches* include NCC [15] and PROGRAML [43]. On one hand, NCC [15] extracts the contextual flow graph of a program by building an LLVM intermediate representation (IR) of the program. It then applies word2vec [145] to learn code representations. On the other hand, PROGRAML [43] is a language-independent, portable representation of whole-program semantics for deep learning, which is designed for data flow analysis in compiler optimization. It adopts message passing neural networks (MPNN) [68] to learn LLVM IR representations. In contrast to these approaches, GraphCode2Vec captures both semantics and syntax.

Combined Semantic and Syntactic -based Approaches: There are generic approaches that capture both syntactic and semantic features such as IR2Vec [15], OSCAR [170], ProgramGraph [3], ProjectCodeNet [179] and GraphCodeBERT [76]. IR2Vec [15] and OSCAR [170] use LLVM IR representation of a program to capture program semantics. Meanwhile, ProgramGraph [3] uses GNN to learn syntactic and semantic representations of code from ASTs augmented with data and control edges. ProgHeteroGraph leverages abstract syntax description language (ASDL) grammar to learn code representations via heterogeneous graphs [219]. Finally, GraphCodeBERT [76] is built upon CodeBERT [57], but in addition to capturing syntactic features it also accounts for semantics by employing data flow information in the pre-training stage.

Similar to these approaches, our approach (GraphCode2Vec) learns both syntactic and semantic features. In Chapter 5, we compare GraphCode2Vec to GraphCodeBERT because it is the most recent state-of-the-art and closely related approach to ours, since it captures both syntax and semantics (*see* RQ2 section 5.5).

5.2.2 Task-specific learning-based applications

Researchers have proposed specialised learning-based techniques to tackle specific (SE) downstream tasks, e.g.. *patch classification* [128, 232] and *solution classification* [67, 156, 177]. In the experiments of Chapter 5 , we consider specialised learning approaches for both tasks. This is because these tasks have several software engineering applications, especially during software maintenance and evolution [128, 156, 232].

Table 5.2 highlights details of our task-specific learning methods.

Solution classification: Let us describe the state-of-the-art learning-based approaches for solution classification. Most of these approaches are syntax-based and adopt convolution neural networks (CNNs) to classify programming tasks. SequentialCNN [67] applies a CNN to predict the language/tasks from code snippets using lexicalized tokens represented as a matrix of word embeddings. CNNSentence [156] is similar to SequentialCNN since it also uses CNNs, except that it classifies source code without relying on keywords, *e.g.*, variable and function names. It instead considers the structural features of the program in terms of tokens that characterize the process of arithmetic

Table 5.2: Details of the state-of-the-art Code Embedding approaches. "Semantic" or "Sem" means program dependence, and "Syntactic" or "Syntax" refers to strings, tokens, parse tree or AST-tree. Symbol "✓" means the approach supports a feature, and "×" means it does not support the feature.

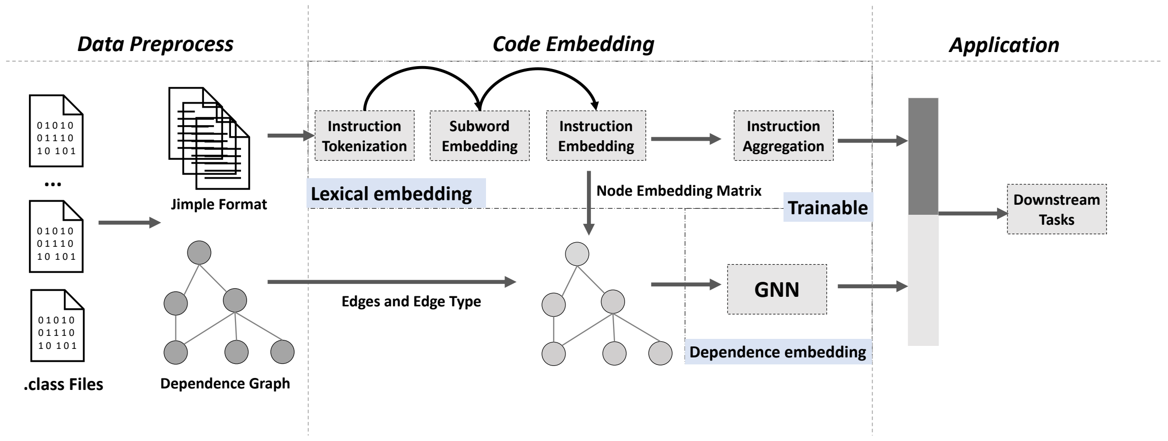
Type		**Approaches**	**Syntactic**	**Semantic**	**Granularity**	
					Method	**Class**
Task-specific	*Syntax*	CNNSentence [156]	✓	×	×	✓
		OneCNNLayer [177]	✓	×	×	✓
		SequentialCNN [67]	✓	×	×	✓
	Both	SimFeatures [217]	✓	✓	×	✓
		Prophet [128]	✓	✓	×	✓
		PatchSim [228]	✓	✓	×	✓
		ODS [232]	✓	✓	×	✓
Generic	*Syntax-only*	CodeBERT [57]	✓	×	✓	×
		Code2Vec [5]	✓	×	✓	×
		Code2Seq [4]	✓	×	✓	×
		C-BERT [27]	✓	×	✓	✓
		InferCode [26]	✓	×	✓	✓
		CC2Vec [83]	✓	×	✓	✓
		AST-based NN [234]	✓	×	×	✓
		ProgHeteroGraph [219]	✓	×	✓	×
	Sem.	NCC [15]	×	✓	✓	✓
		PROGRAML [43]	×	✓	✓	✓
	Both	IR2Vec [15]	✓	✓	✓	✓
		OSCAR [170]	✓	✓	✓	✓
		ProgramGraph [3]	✓	✓	✓	✓
		ProjectCodeNet [179]	✓	✓	×	✓
		GraphCodeBERT [76]	✓	✓	✓	×
		GraphCode2Vec	✓	✓	✓	✓

processing, loop processing, and conditional branch processing. Finally, OneCNNLayer [177] also uses CNN for solution classification. It firstly pre-processes the program to remove unwanted entities (*e.g.*, comments, spaces, tabs and new lines), then tokenizes the program to generate the code embedding using word2vec. The resulting embedding includes the token connections and their underlying meaning in the vector space.

Patch Classification: These are techniques designed to determine the correctness of patches (i.e., identify correct, wrong or over-fitting patches). These learning-based techniques can be static (e.g., ODS [232]), dynamic (e.g., Prophet [128]), heuristic-based (e.g., PatchSim [228]) or hybrid (e.g., SimFeatures [217]). Table 5.2 provides details of these approaches. Notably, they all capture both syntactic information (e.g. via AST) and program dependence information (e.g., via execution paths or control flow information). For instance, PatchSim [228] is a *heuristic approach* that leverages the behavioral similarity of test case executions to determine patch correctness by leveraging the complete path spectrum of test executions. Meanwhile, Wang *et al.* [217] proposed (SimFeatures –) a *hybrid strategy that identifies correct patches by integrating static code features with dynamic features or (test) heuristics.* SimFeatures combines a learned static code model with dynamic or heuristic-based information (such as the dependency similarity between a buggy program and a patch) using majority voting. More recently, Y.He *et al.* [232] proposed a supervised learning approach (called ODS) that employs static code features of patched and buggy programs to determine patch correctness, specifically to classify over-fitting patches. It uses supervised learning on extracted static code at the AST level to learn a probabilistic model for determining patch correctness. ODS also tracks program dependencies by tracking control flow statements.

In Chapter 5, we compare GraphCode2Vec to the aforementioned seven (7) learning-based methods for solution classification and patch classification(*see* Section 5.5 in Chapter 5).

Figure 5.2: Overview of GraphCode2Vec



5.2.3 Graph Neural Networks

Graph Neural Network (GNN) [191] can process structural data consisting of nodes, edges and their attributes. During training, a node representation is updated iteratively by aggregating the features of its neighbors [229, 247]. The process is defined by two important functions, i.e., the *aggregation function* and the *combining function*. The aggregation function defines how to aggregate the features from the node neighbors. The combining function specifies how to update the node representation. Different aggregation and combining functions lead to different GNN variants. there are several popular GNN types: Graph Convolutional Network (GCN; [102]), GraphSAGE [77], Graph Attention Network (GAN; [212]), Graph Isomorphism Network (GIN; [229]), and Variational Graph Auto-Encoder (VGAE; [103]). GCN defines a graph convolutional operator on the Laplacian matrix as the aggregation function, similar to the convolutional operator in Convolutional Neurnal Network (CNN; [116]). GraphSAGE [77] directly aggregates the neighbor features instead of using Laplacian matrix. GIN uses the sum operation as aggregation function that makes GNN can well distinguish different graphs. GAN utilizes the self-attention mechanism to assign different weight values to the neighbors during updating a node representation. VGAE uses the encoder-decoder architecture to get the node representation. It assumes that the node representation follows a Gaussian distribution so that the encoder learns parameters of the distribution. Then, the decoder reconstruct the graph from the output of the encoder.

5.3 Approach

5.3.1 Overview

Figure 5.2 illustrates the steps and components of our approach. First, GraphCode2Vec takes as input a Java program (i.e. a set of class files) that is converted to a Jimple intermediate representation. Secondly, GraphCode2Vec employs Soot [209] to obtain the program dependence graph (PDG) by feeding the class files as input. From the resulting Jimple representation and PDG, GraphCode2Vec learns two program embeddings, namely a lexical embedding and a dependence embedding. These two embeddings are ultimately concatenated to form the final code embedding.

To achieve *lexical embedding*, our approach first tokenizes the Jimple instructions obtained from our pre-processing step into sub-words. Next, given the sub-words, our approach learns sub-word embedding using word2vec [144]. Then, it learns the instruction embedding by representing every Jimple instruction as a sequence of subwords embeddings using a bi-directional LSTM (BiLSTM, Section 5.3.2). The forward and backward hidden states of this BiLSTM allows to build the instruction embeddings. GraphCode2Vec employs a BiLSTM since it learns context better: BiLSTM can learn both past and future information while LSTM only learns past information. Finally, it aggregates multiple instruction embeddings using element-wise addition, in order to obtain the overall lexical program embedding.

To learn the *dependence embedding*, GraphCode2Vec applies a Graph Neural Network (GNN) [191] to embed Jimple instructions and their dependencies. Each node in the graph corresponds to a Jimple instruction and contains the (dependence) embedding of this instruction. Node attributes are from lexical embeddings. The edges of the graph represent the dependencies between instructions. Our approach considers the following program dependencies: data flow, control flow and method call graphs. GraphCode2Vec uses intra-procedural analysis [58] to extract data-flow and control-flow dependencies by invoking Soot [209]. Then, it builds method call graphs via class hierarchy analysis [45].

The training of GNNs is an iterative process where, at each iteration, the embedding of each node n is updated based on the embedding of the neighboring nodes (i.e., nodes connected to n) and the type of n's edges [229, 247]. The *message passing function* determines how to combine the embedding of the neighbors – also based on the edge types – and how to update the embedding n based on its current embedding and the combined neighbors' embedding. The dependence embedding of an instruction is the embedding of the corresponding node at the end of the training process.

Finally, after obtaining lexical embedding and dependence embedding, our approach concatenates both embeddings to obtain the overall program representation.

5.3.2 Lexical embedding

Step 1 - Jimple code tokenization: The first crucial step of GraphCode2Vec is to properly tokenize Jimple code into meaningful "tokens", to learn the vector representations. The traditional way to tokenize code is to split it on whitespaces. However, this manner is inappropriate for two reasons. First, whitespace-based tokenization often results in long tokens such as long method names (e.g., "getFunctionalInterfaceMethodSignature"). Long sequences often have a low frequency in a given corpus, which subsequently leads to an embedding of inferior quality. Second, whitespace-based tokenization is not able to process new words that do not occur in the training data – these out-of-vocabulary words are typically replaced by a dedicated "unknown" token. This is an obvious disadvantage for our approach, whose goal is to support practitioners to analyze diverse programs – which may then include words that did not occur in the programs used to learn the embedding.

To address this challenge, we tokenize the Jimple code into *subwords* [108, 193, 225], which are units shorter than words, e.g., morphemes. Subwords have been widely adopted in representation learning systems for texts [47, 81, 181, 245] as they solve the problem of overly long tokens and out-of-vocabulary words. New code programs can be smoothly handled using short tokens representation, by limiting the amount of long, but different tokens. Subwords get rid of the almost-infinite character combinations that are common in many program codes. For example, this is the reason why BERT uses wordpiece subwords [225], and XLNet [231] and T5 [181] use sentence-piece subwords. Similarly, GraphCode2Vec uses sentence-piece subwords. When using subwords, the long token "getFunctionalInterfaceMethodSignature" is split into "get", "Functional", "Interface", "Method" and "Signature". It is worth noting that most of the subwords are in fact words, e.g., "get" [93].

Step 2 - Subword embedding with word2vec: Given a subword-tokenized Jimple code corpus $\mathcal{C}$ with vocabulary size $|\mathcal{C}|$, our approach learns a subword embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{C}| \times d}$ where d is a hyperparameter referring to the embedding dimension (d is usually set to 100). It uses the popular Skip-gram with negative sampling (SGNS) method in word2vec [144] to produce $\mathbf{E}$. And $\mathbf{E}$ is utilized as the subword embedding matrix [144].

Step 3 - Instruction embedding: After forming the subword embeddings, GraphCode2Vec represents every Jimple instruction as a sequence of subword embeddings $(\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_n)$, by using a bidirectional LSTM (BiLSTM). The role of BiLSTM is to learn the embedding of the instruction from the subword sequence of the instruction. Let $\overrightarrow{\mathbf{h}_t}$ and $\overleftarrow{\mathbf{h}_t}$ be the forward hidden state and backward hidden state of LSTM after feeding the final subword. Then, it forms the instruction embedding by concatenating $\overrightarrow{\mathbf{h}_t}$ and $\overleftarrow{\mathbf{h}_t}$, denoted as $\mathbf{x} = (\overrightarrow{\mathbf{h}_t}, \overleftarrow{\mathbf{h}_t})$.

Step 4 - Instruction embedding aggregation: The last step in the process of forming lexical embedding is the aggregation of the instruction embeddings in order to form the overall program lexical embedding. The reason why we aggregate instruction-level embedding as opposed to learning an embedding for the whole program is that LSTMs work with sequences of limited length and thus, truncate the instructions into small sequences (not exceeding the maximal length). After tokenization, a program can have many subwords and if one directly consider all subwords in the program, one needs to cut these subwords into the limited sequence length for LSTM and result in information loss.

Our approach uses element-wise addition as the token aggregation function. This operation allows the aggregation of multiple instruction embeddings while keeping a limited vector length.

5.3.3 Dependence embedding

Step 1 - Building method graphs: A method graph is a tuple $G = (V, E, \mathbf{X}, \mathbf{K})$, where V is the set of nodes (i.e. Jimple instructions), E is the set of edges (dependence relations between the instructions), $\mathbf{X}$ is the node embedding matrix (which contains the embedding of the instructions) and $\mathbf{K}$ is the edge attribute matrix (which encodes the dependencies that exist between instructions). For each node n there is a column vector $\mathbf{x_n}$ in $\mathbf{X}$ such that $\mathbf{x_n} = (\overrightarrow{\mathbf{h}}_t, \overleftarrow{\mathbf{h}}_t)$ (instruction embedding).

To define E and $\mathbf{K}$, our approach extracts data-flow and control-flow dependencies by invoking Soot [58, 209]. Then, GraphCode2Vec introduces an edge between two nodes if and only if the two corresponding instructions share some dependence.

Step 2 - Building program graphs: A program graph consists of a pair $\mathcal{P} = (\mathcal{G}, \mathcal{R})$ where $\mathcal{G} = \{G_0, G_1, ..., G_m\}$ is a set of method graphs and where $\mathcal{R} \subseteq \mathcal{G}^2$ is the call relation between the methods, that is, $(G_i, G_j) \in \mathcal{R}$ if and only if the method that G_i represents calls the method that G_j represents. To represent this relation in the GNN, GraphCode2Vec introduces an entry node and an exit node for each method and edges linking those nodes with caller instructions.

Step 3 - Message passing function: The exact definition of the message passing function depends on the used GNN architecture. We choose the widely-used GNN architectures with linear complexity [226] that has been successfully applied in various application domains. GraphCode2Vec employs four GNN architectures, namely Graph Convolutional Network (GCN [102]), GraphSAGE [77], Graph Attention Network (GAN [212]), Graph Isomorphism Network (GIN [229]).

Step 4 - Learning the dependence embedding: The dependence embedding of each instruction is obtained by running the message passing function on all nodes for a pre-defined number of iterations, i.e., the number of GNN layers. Once these instruction embeddings have been produced, GraphCode2Vec aggregates them using the global attention pool operation [123] in order to produce the program-level dependence embedding. Attention mechanism can make program-level dependence embedding consider more important nodes (instructions).

The dependence embeddings that GNN produces depend on the learnable parameters of (a) the message passing function and (b) bidirectional LSTM. These parameters can be automatically set to optimize the effectiveness of GraphCode2Vec either directly on the downstream task or on some pre-training objectives, as described hereafter.

In the end, our approach uses a concatenation operator to get the program embedding vector. Concatenation has been shown to be an effective method to fuse features without information loss when using DNN [66, 89, 113, 159, 201, 202]. Although the dependence embedding inherently encodes the lexical embedding, the importance of lexical inherently fades away as the semantic representation is learnt. Our ablation study (see RQ3 in Section 5.5) later reveals the benefits of concatenating an explicit lexical embedding with the dependence embedding.

5.3.4 Pre-training

Self-supervised learning has been applied with success for pre-training deep learning models [50, 127, 183]. It allows a model to learn how to perform tasks without human supervision [150, 248] by learning a universal embedding that can be fine-tuned to solve multiple downstream tasks. In this work, we employed three (3) self-supervised learning strategies to pre-train the BiLSTM and GNN in GraphCode2Vec, namely *node classification*, *context prediction* [87], and *variational graph encoding* (VGAE) [103]. Node (or Instruction) classification trains the model to infer the type of an instruction, given its embedding. Context prediction requires the model to predict a masked node representation, given its surrounding context. Variational graph encoding (VGAE) learns to encode and decode the code dependence graph structure. Note that these pretraining procedures *do not* require any human-labeled datasets. The model learns from the raw datasets without any human supervision.

5.4 Experimental Setup

Research Questions: Our research questions (RQs) are designed to evaluate the *effectiveness* of GraphCode2Vec. In particular, we compare the *effectiveness* of GraphCode2Vec to the state-of-the-art in *task-specific* and *generic* code embedding methods (*see* RQ1 and RQ2). This is to demonstrate the utility of GraphCode2Vec in solving downstream tasks, in comparison to specialised learning-based approaches tailored towards solving specific SE tasks (RQ1) and other general-purpose code embedding approaches (RQ1). We also examine if GraphCode2Vec effectively embeds lexical and program dependence features in the latent space, and how this impacts its effectiveness on downstream tasks (*see* RQ3). The first goal of RQ3 is to demonstrate the validity of our approach, i.e., analyse that it indeed embeds lexical and dependence features as intended via *probing analysis.* In addition, we analyse the contribution of lexical embedding and dependence embedding to its effectiveness on downstream tasks by conducting an *ablation study.* We also investigate the *sensitivity* of our approach to the choices in GraphCode2Vec's framework, e.g., *model pre-training (strategy)* and *GNN configuration.* These experiments allow to evaluate the influence of these choices on the effectiveness of GraphCode2Vec.

Specifically, we ask the following research questions (RQs):

RQ1 Task-specific learning-based applications: Is our approach (GraphCode2Vec) effective in comparison to the state-of-the-art *task-specific* learning-based applications? What is the benefit of capturing semantic features in our code embedding?

RQ2 Generic Code embedding: How effective is our approach (GraphCode2Vec), in comparison to the state-of-the-art syntax-only generic code embedding approaches? What is the impact of capturing both syntactic and semantic features (i.e., program dependencies) in code embedding? How does GraphCode2Vec compare to GraphCodeBERT, a larger and more complex model?

RQ3 Further Analyses: What is the impact of model pre-training on the effectiveness of GraphCode2Vec? Does our approach effectively capture lexical and program dependence features? What is the contribution of lexical embedding or dependence embedding to the effectiveness of our approach on downstream tasks? Is our approach *sensitive* to the choice of GNN?

Baselines: We compare the effectiveness of GraphCode2Vec to several state-of-the-art code embedding approaches (aka *generic baselines*), and specialised or *task-specific learning-based applications.* On one hand, *generic baselines* refers to code embedding approaches that are designed to be general-purpose, i.e., they provide a code embedding that is amenable to address several downstream tasks. On the other hand, *task-specific* baselines refers to learning-based approaches that address a specific downstream SE task, e.g., patch classification.

Table 5.2 provides details about these baselines for solution classification and patch classification. Specifically, we evaluated GraphCode2Vec in comparison to four (4) generic code embedding approaches, namely Code2Seq [4], Code2Vec [5], CodeBERT [57] and GraphCodeBERT [76] (*see* RQ2 in section 5.5). We have selected these generic baselines because they have been evaluated against several well-known state-of-the-art code embedding methods and demonstrated considerable improvement over them. Besides, these approaches are recent, popularly used and have been applied on many downstream (SE) tasks.

For task-specific learning-based approaches, we consider solution classification, and patch classification. These are popular SE downstream tasks that have been studied using learning-based approaches.

We utilised three (3) specialised learning-based baseline for the solution classification task, namely CNNSentence [156], OneCNNLayer [177] and SequentialCNN [67]. We also used all four patch classifiers (Prophet [128], PatchSim [228], SimFeatures [217] and ODS [232]). These task-specific baselines have been selected because they have been shown to outperform other proposed learning-based approaches for these tasks. For instance, SequentialCNN [67] has been evaluated against five other learning-based approaches and demonstrated to be more effective. ODS [232] has also been shown to be more effective and efficient than the three other patch classifiers.

Subject Programs: In our experiments, we employed eight (8) subject programs written in `Java`. Table 5.3 provides details about each of our subject programs and their experimental usage. Notably, we employ four (4) publicly available programs for the downstream tasks, namely Defects4J [94], Java-Small [5], and Java250 [179]. These datasets were employed for our comparative evaluation (*see* RQ1 and RQ2). We chose these datasets because they are popular and have been employed in the evaluation of our downstream tasks in previous studies [4, 179, 232, 242]. Besides, we employed Java-Small and Java250 in our ablation study where we evaluate the contribution of lexical and dependence embedding to the effectiveness of GraphCode2Vec (RQ3). We chose these two datasets for this task because they correspond to tasks that require lexical and semantic information to be effectively addressed. To further analyze GraphCode2Vec (*see* RQ3), we employed the Concurrency dataset [48, 63] and collected two (2) subject programs (named LeetCode-10 and M-LeetCode) from LeetCode[4]. We use these programs to investigate the difference between capturing lexical and dependence information. In particular, the Concurrency dataset contains different concurrent code types, which have similar syntactic/lexical features but different structure information. We mutated LeetCode-10 to create M-LeetCode dataset. Our mutation preserves lexical features, but modifies semantic or program dependence features such that LeetCode-10 and M-LeetCode have the same lexical features, but different semantics. For example, a simple dependence mutant involves switching outer and inner loops. We utilize LeetCode-10, M-LeetCode and Concurrency for the probing analysis of our approach (GraphCode2Vec).

Downstream Tasks: In our evaluation, we considered four (4) major software engineering tasks, namely, *mutant prediction*, *patch classification*, *method name prediction*, and *solution classification.* These are popular downstream SE tasks that have been investigated in the community for decades. For these four tasks, we evaluated GraphCode2Vec in comparison to four *generic baselines*, namely

[4]https://leetcode.com/

Table 5.3: Details of Subject Programs

Subject Program	#Progs.	Tasks/Analyses
`Java-Small`	11	Method Name Prediction and Ablation Studies
`Java250`	75000	Solution Classification and Ablation Studies
`Defects4J`	15 & 5	Mutant Prediction and Patch Classification
`LeetCode-10`	100	Probing Analysis
`M-LeetCode`	100	Probing Analysis
`Concurrency`	46	Probing Analysis
`Jimple-Graph`	1976	Model Pre-training

Code2Seq [4], Code2Vec [5], CodeBERT [57] and GraphCodeBERT [76]. Table 5.3 provides details on the subject programs employed for each downstream tasks. In the following, we provide further details about the experimental setup for each task evaluated in this work.

Method Name Prediction: This refers to the task of predicting the method name of a function in a program, given a set of method names and the body of the function as inputs [26]. This task is useful for automatic code completion during programming. In our experiment, all four generic baselines were evaluated for this task. We evaluated this task using the Java-Small dataset, since it was designed for this task in previous studies [5] (*see* Table 5.3).

Solution Classification: This refers to the classification of source code into a predefined number of classes, e.g., based on the task it solves [177], or programming languages [67]. It has 250 problems(labels) and each problem has 300 solutions[5]. This is useful to assist or assess programming tasks and manage code warehouse. We evaluated all four generic baselines on this task, as well as three specialised learning-based approaches for this task, namely CNNSentence [156], OneCNNLayer [177], SequentialCNN [67] (Table 5.2). We evaluated this task using the Java250 dataset, which was designed for this task in previous studies [179] (*see* Table 5.3).

Patch Classification: For this task, the aim is to identify the correctness of patches, i.e., if a patch is (in)correct, wrong or over-fitting [228, 232]. In our experiment, we compare the performance of GraphCode2Vec to the four generic baselines, as well as the current state-of-the-art learning-based approach for patch classification, i.e, ODS [232]. We employed the Defects4J [94] dataset (*see* Table 5.3) which has also been used by previous studies for this task [228, 232]. The goal of this task is to identify over-fitting APR patches. We used five (5) programs and 890 APR patches[6] containing 643 over-fitting patches and 247 correct patches.

Mutant Prediction: The goal of this task is to predict different types of mutants employed during mutation testing. Mutation testing is an important SE task that is typically deployed to determine the adequacy of a test suite to expose injected faults in a program [165]. In this work, we predict if a mutant is *killable* or *live*. To this end, we employ the Defects4J [94] dataset (*see* Table 5.3) which has been popularly employed for several SE tasks, including mutation testing [165]. We curated a mutant prediction dataset containing 15 Java programs, and 16,216 mutants.

Pre-training Setup: For model pre-training, we curated the `Jimple-Graph` dataset from the Maven repository[7], it contains 1,976 Java libraries with about 3.5 millions methods in total. We randomly sample around 10% data for the pre-traning purpose. These Java libraries are from 42 application domains, this ensures a reasonable program diversity, these domains include math and image processing libraries. For the BiLSTM component (Section 5.3.2), we use one layer with hidden

[5]The dataset description can be found in the link, `https://dax-cdn.cdn.appdomain.cloud/dax-project-codenet/1.0.0/readme.html?_ga=2.202687321.954561633.1651654500-214066389.1651654500`

[6]We exempted 12 patches out of the 902 patched programs used by ODS, since they deleted complete functions, and there is no code representation for deleted functions.

[7]https://mvnrepository.com/

Table 5.4: Effectiveness of GraphCode2Vec vs. Syntax-only Generic Code Embedding approaches. The best results are in **bold** text, the results for the best-performing baseline are in *italics*. We report the improvement in effectiveness between GraphCode2Vec and the best-performing baseline in "% Improvement", improvements above five percent (>5%) are in **bold** text.

Generic Code	**Method Name Prediction**			**Solution Classification**			**Mutant Prediction**			**Patch Classification**		
Embedding	**F1**	**Preci**	**Recall**	**F1**	**Preci**	**Recall**	**F1**	**Preci**	**Recall**	**F1**	**Preci**	**Recall**
Code2Seq	*0.4920*	*0.5963*	0.4187	0.7542	0.7678	0.7536	0.5911	0.6423	0.5881	0.8901	0.8355	*0.9541*
Code2Vec	0.3309	0.3779	0.2943	0.8034	0.8081	0.8028	0.6398	0.6632	0.6320	0.8787	0.8806	0.8782
CodeBERT	0.3963	0.3295	*0.4969*	*0.8783*	*0.8747*	*0.8878*	*0.7106*	*0.7305*	*0.6995*	*0.9275*	*0.9099*	0.9473
GraphCode2Vec	**0.5807**	**0.6150**	**0.5502**	**0.9746**	**0.9753**	**0.9746**	**0.7542**	**0.7569**	**0.7524**	**0.9359**	**0.9145**	**0.9602**
% Improvement	**18.03%**	3.14%	**10.73%**	**10.96%**	**11.50%**	**9.78%**	**6.14%**	3.61%	**7.56%**	0.91%	0.51%	0.64%

dimension size 150. We pre-train sub-tokens using the `Jimple` text for each program, the sub-token embedding dimension is set to 100 (*see* Section 5.3). We fine-tune the downstream tasks using the obtained pre-trained weights after one epoch. All GNNs use five (5) layers with dropout ratio 0.2. We use Adam [99] optimizer with 0.001 learning rate. In our experiment, we evaluated all three (3) pre-training strategies (Section 5.3.4).

Metrics and Measures: For all tasks, we report F1-score, precision and recall. We discuss most of our results using F1-score since it is the harmonic mean of precision and recall. Besides, it is a better measurement metric than accuracy, especially when the dataset is imbalanced (e.g., Java-Small). Hence, we do not report the accuracy for imbalanced datasets, e.g., mutant data is imbalanced with about 30% live mutants and 70% killable mutants. We provide the code details in the Github repository[8].

Probing Analysis: The goal of our probing analysis is to ensure that lexical and dependence features are indeed learned by GraphCode2Vec's code embedding. Probing is a widely used technique to examine an embedding for desired properties [42, 183, 243]. To this end, we trained diagnostic classifiers to probe GraphCode2Vec's code embedding for our desired properties (i.e., lexical and/or program dependence features). Concretely, we train a simple classifier with one MLP layer fed with the learned code embedding (e.g. lexical) to examine if our code embedding encodes the desired property. To achieve this, we curated a dedicated dataset for training and evaluating our probing classifiers. Specifically, we employ three probing datasets, namely LeetCode-10, M-LeetCode and Concurrency (Table 5.3). We have employed these datasets because they require lexical or dependence embedding to address their corresponding tasks.

Probing Task Design: We design four probing tasks. The first three (Task-1, Task-2 and Task-3) use LeetCode-10 and M-LeetCode, and the last one (Task-4) uses Concurrency. *Task-1* classifies what problem the solution code solves on LeetCode-10. LeetCode-10 shares lexical token similarities within one problem group, and some solutions from the different problem groups may have the same semantic structure, e.g., using one for-loop. Therefore, we hypothesize that the lexical embedding is more informative than the semantic embedding for Task-1. *Task-2* mixes LeetCode-10 and M-LeetCode, and then judges which dataset the input code is from (binary classification). LeetCode-10 and M-LeetCode share lots of similar lexical tokens but the code semantic structures are different. Hence, the semantic embedding should be more informative than the code lexical syntactic embedding. *Task-3* also mixes the two datasets but uses all the 20 labels instead of a binary classification. Task-3 integrates Task-1 and Task-2, requiring both lexical and semantic information. *Task-4* is a concurrency bug classification task. The code with same label can have the high lexical similarity but the code semantic structure should be different.

GraphCode2Vec's Configuration: We employ three (3) pre-training strategies, namely node classification, context prediction and VGAE. Our approach supports four (4) GNN architectures for dependence embedding (*see* Section 5.3), namely GCN [102], GraphSAGE [77], GAN [212] and GIN [229]. In total, we have 12 possible configurations. However, the *default configuration is context*

[8]`https://github.com/graphcode2vec/graphcode2vec`

Table 5.5: Effectiveness of GraphCode2Vec (aka "Graph.") vs. Task-Specific learning-based approaches for two SE tasks. The best results are in **bold** text, the results for the second best-performing approach are in *italics*. The improvement in effectiveness between GraphCode2Vec and the best-performing baseline is reported in "Graph. *(% Improv.)*".

	Solution Classification				Patch Classification				
	CNN Sen.	One CNN.	Seq.-CNN	Graph. *(% Improv.)*	SimFea-tures	Prop-het	Patch-Sim	ODS	Graph. *(% Improv.)*
F1-Score	*0.690*	0.540	0.470	**0.970 (40.6%)**	0.881	0.892	0.881	*0.900*	**0.915 (1.7%)**
Recall	*0.690*	0.540	0.470	**0.970 (40.6%)**	0.895	0.891	0.389	*0.950*	**0.960 (2.1%)**
Precision	*0.700*	0.550	0.480	**0.970 (38.6%)**	0.870	0.889	0.830	*0.924*	**0.936 (1.3%)**

Table 5.6: Effectiveness of GraphCode2Vec vs. GraphCodeBERT. Lower complexity, the best results and higher improvements (above five percent (>5%)) are in **bold** text.) are in **bold** text.

Generic Code	**Model**	**Pretrain**	**Method Name Prediction**			**Solution Classification**			**Mutant Prediction**			**Patch Classification**		
Embedding	**Size**	**Data**	**F1**	**Preci**	**Recall**	**F1**	**Preci**	**Recall**	**F1**	**Preci**	**Recall**	**F1**	**Preci**	**Recall**
GraphCodeBERT	124M	2.3M	0.5761	**0.7261**	0.4775	**0.9850**	**0.9868**	**0.9843**	**0.7649**	**0.768**	**0.7623**	0.9317	0.9108	0.9557
GraphCode2Vec	**2.8M**	**314K**	**0.5807**	0.6150	**0.5502**	0.9746	0.9753	0.9746	*0.7542*	*0.7569*	*0.7524*	**0.9359**	**0.9145**	**0.9602**
% Improvement	**50X**	**7X**	**7.99%**	-15.30%	**15.23%**	-1.07%	-1.17%	-0.18%	-1.40%	-1.45%	-1.30%	0.45%	0.41%	0.47%

prediction for pre-training and dependence embedding with GAT architecture. In our experiments, we evaluate the effect of each configuration on the effectiveness of our approach (*see* Section 5.5).

Implementation Details and Platform: GraphCode2Vec was implemented in about 4.8 KLOC of Python code, using the Pytorch ML framework. Our data processing and evaluation code is about 3 KLOC of Java code. We use Soot [209] to extract the program dependence graph (PDG). We reuse the code from the public repository of each baseline in our experiments.[9] However, we adapt each baseline to our downstream tasks, e.g., by replacing the classifier but using the same performance metrics. All experiments were conducted on a Tesla V100 GPU server, with 40 CPUs (2.20 GHz) and 256G of main memory. The implementation of GraphCode2Vec is available online[10].

5.5 Experimental Results

RQ1 Task-specific learning-based applications: This experiment examines how GraphCode2Vec compares to seven (7) state-of-the-art task-specific learning-based techniques for *solution classification* and *patch classification*. We selected these two tasks for this experiment due to their popularity, availability of ML-based baselines and their application to vital SE tasks, e.g., automated program repair, patch validation, code evolution, and software warehousing. We evaluated against three solution classifiers, namely CNNSentence [156], OneCNNLayer [177], SequentialCNN [67]. We also compare GraphCode2Vec to four patch classifiers – Prophet [128], PatchSim [228], SimFeatures [217] and ODS [232].

Our evaluation results show that *GraphCode2Vec outperforms the state-of-the-art task-specific learning based approaches for the tested tasks, i.e., patch classification, and solution classification.* Table 5.5 highlights the effectiveness of GraphCode2Vec in comparison to learning-based approaches for patch classification and solution classification, respectively. In particular, GraphCode2Vec outperforms all seven task-specific baselines in our evaluation. GraphCode2Vec outperforms all three baselines for solution classification, it is almost twice as effective as SequentialCNN and OneCNNLayer, and 40% more effective than the best baseline – CNNSentence (*see* Table 5.5). In addition, GraphCode2Vec outperforms all four state of the art patch classifiers, i.e., ODS [232], Prophet [128]), PatchSim [228] and SimFeatures [217]. It is at least twice as effective as PatchSim (in terms of recall) and slightly (up to 2%) more effective than the best baseline, i.e., ODS (*see*

[9]https://github.com/tech-srl/code2vec,https://github.com/tech-srl/code2seq, https://github.com/microsoft/CodeBERT, https://github.com/hukuda222/code2seq

[10]https://github.com/graphcode2vec/graphcode2vec

Table 5.5). This result demonstrates the utility of our approach in addressing both downstream tasks. Furthermore, it highlights the effectiveness of generic code embedding in comparison to specialised learning-based approaches. This superior performance can be attributed to the fact that GraphCode2Vec is generic, and it employs self-supervised model pre-training.

GraphCode2Vec is up to two times (2x) more effective than the seven (7) state-of-the-art task-specific approaches, for both tasks.

RQ2 Generic Code embedding: In this experiment, we demonstrate how GraphCode2Vec compares to the state-of-the-art generic code embedding approaches. We thus, compare the effectiveness of GraphCode2Vec with three (3) syntax-only generic baselines, namely CodeBERT, Code2Seq and Code2Vec. Additionally, we compare the effectiveness of our approach to a a larger and more complex state-of-the-art generic approach that captures both syntax and semantics, specifically, GraphCodeBERT. We used four (4) downstream SE tasks – method name prediction, solution classification, mutant prediction and patch classification.

Syntax-only Generic Embedding: In our evaluation, we found that *our approach (GraphCode2Vec) outperforms all syntax-based generic baselines for all tasks*. Table 5.4 highlights the effectiveness of GraphCode2Vec in comparison to the baselines (i.e., Code2seq, Code2Vec and CodeBERT). As an example, consider method name prediction, GraphCode2Vec is twice as effective as some baselines, e.g., Code2Vec. For all (four) tasks, GraphCode2Vec clearly outperforms all baselines across all metrics. It is up to 12% and 18% more effective than the best baselines, CodeBERT and Code2Seq, respectively. We observed CodeBERT is the best baseline on three tasks. We attribute the performance of CodeBERT on these tasks to its much higher complexity (i.e., huge number of trainable parameters, more than 124M) and the size of the pre-training dataset (8.5M) [90]. Overall, our results demonstrate that including semantic program features improves the performance of code representation across these downstream tasks. Thus, emphasizing the importance of semantic features in addressing SE tasks, especially the need to capture program dependencies in code representation.

For all (four) tasks, GraphCode2Vec is (up to 18%) more effective than (the best) syntax-only baselines.

Complementarity with GraphCodeBERT: We also observe that despite the lower complexity of our approach (GraphCode2Vec), *it is comparable and complementary to GraphCodeBERT* across tested tasks. GraphCodeBERT captures both syntactic and semantic program features but, it is significantly larger and complex than GraphCode2Vec. Table 5.6 highlights the complexity and effectiveness of GraphCodeBERT in comparison to GraphCode2Vec. For instance, GraphCodeBERT has at least 50 times (50x) as many trainable parameters as GraphCode2Vec (124 million versus 2.8 million parameters), and seven times (7x) as much pre-training data (2.3M versus 314K methods). Despite the difference in size and complexity, GraphCodeBERT has a comparable performance to GraphCode2Vec. Specifically, GraphCode2Vec outperforms GraphCodeBERT on two tasks (method name prediction and patch classification) and it is comparable on the other two tasks (solution classification, and mutant prediction). Notably, GraphCodeBERT has a negligible improvement over GraphCode2Vec for these two tasks (about 1%). These results demonstrate that although simpler and trained on 7 times less data, GraphCode2Vec is complementary to GraphCodeBERT. This disparity in size and complexity implies that precise program dependence information is important. Nevertheless, our results show that both GraphCode2Vec and GraphCodeBERT are more effective than syntax-only approaches, e.g., CodeBERT (*cf.* Table 5.5 and Table 5.6).

GraphCode2Vec is complementary to GraphCodeBERT despite being simpler and trained on seven times (7x) less data. It is more effective on two tasks, and comparable on the other two tasks.

Table 5.7: Probing Analysis results showing the accuracy for all pre-training strategies and GNN configurations. Best results for each sub-category are in bold, and the better results between syntactic (lexical) embedding and semantic embedding is in *italics*. "syn+sem" refers to GraphCode2Vec's models capturing both syntactic and semantic features.

Pre-training	**Captured**	**Task-1** (syntax-only)				**Task-2** (semantic-only)				**Task-3** (syntax and semantic)				**Task-4** (semantic-only)			
Strategy	**Feature**	**GCN**	**GIN**	**GSAGE**	**GAT**	**GCN**	**GIN**	**GSAGE**	**GAT**	**GCN**	**GIN**	**GSAGE**	**GAT**	**GCN**	**GIN**	**GSAGE**	**GAT**
Context	**semantic**	0.822	0.674	0.842	0.886	***0.684***	*0.614*	***0.704***	*0.741*	0.513	0.381	*0.543*	***0.612***	***0.654***	***0.666***	***0.657***	***0.594***
	syntactic	***0.934***	***0.938***	*0.942*	*0.928*	0.615	0.602	0.617	0.602	*0.529*	*0.527*	*0.528*	0.527	0.580	0.525	0.524	0.449
	syn+sem	0.918	0.928	**0.95**	**0.942**	0.641	**0.641**	0.688	**0.797**	**0.559**	**0.546**	**0.587**	0.6	0.605	0.592	0.608	0.592
Node	**semantic**	0.758	0.820	0.802	0.840	***0.651***	***0.667***	***0.741***	***0.686***	0.426	***0.514***	***0.625***	***0.563***	***0.647***	***0.664***	***0.659***	***0.670***
	syntactic	***0.904***	*0.884*	***0.876***	***0.916***	0.584	0.587	0.606	0.593	*0.516*	0.504	0.490	0.513	0.484	0.476	0.420	0.550
	syn+sem	0.872	**0.9**	**0.876**	0.902	0.624	0.618	0.691	0.67	**0.522**	0.508	0.572	0.545	0.519	0.522	0.451	0.57
VGAE	**semantic**	0.856	0.812	0.868	0.866	***0.594***	***0.653***	0.583	***0.617***	0.403	*0.532*	0.407	0.477	***0.673***	***0.680***	***0.674***	***0.656***
	syntactic	*0.916*	***0.932***	***0.928***	***0.950***	0.591	0.572	***0.594***	0.599	*0.485*	0.494	*0.492*	***0.495***	0.523	0.617	0.584	0.591
	syn+sem	**0.92**	0.926	**0.928**	0.938	0.59	0.63	0.591	0.596	**0.498**	**0.548**	**0.508**	0.492	0.627	0.658	0.531	0.586
Best Config.		**Syntactic = 8/12**				**Semantic = 9/12**				**Syntactic + Semantic = 7/12**				**Semantic = 12/12**			

Table 5.8: Effectiveness (F1-Score) of GraphCode2Vec on all GNN configurations and Pre-training Strategies, for all downstream tasks. For each subcategory, the best results for each category are in **bold** text.

GNN	**Method Name Prediction**					**Solution Classification**				
	No Pre-	**Pre-training Strategies**				**No Pre-**	**Pre-training Strategies**			
	training	**Context**	**Node**	**VGAE**	**Average**	**training**	**Context**	**Node**	**VGAE**	**Average**
GCN	0.4494	0.5018	0.4859	0.5337	0.4930	0.9679	0.9710	0.9710	0.9751	0.9712
GIN	0.4347	0.4684	0.4037	0.5266	0.4584	0.9645	0.9711	0.9700	0.9710	0.9692
GraphSage	0.3998	0.5006	0.4531	0.5412	0.4736	0.9675	0.9712	0.9721	0.9727	0.9709
GAT	0.4246	0.5807	0.6194	0.5890	0.5534	0.9647	0.9746	0.9703	0.9735	0.9708
Average	0.4271	0.5129	0.4905	**0.5476**		0.9662	0.9720	0.9718	**0.9731**	
Variance	0.0003	0.0017	**0.0064**	0.0006		2.2e-6	**2.2e-6**	7.1e-7	2.3e-6	
SD	0.0180	**0.0413**	0.0800	0.0244		**0.0015**	**0.0015**	0.0008	**0.0015**	

Table 5.9: Ablation Study results showing the F1-Score of GraphCode2Vec. Best results are **bold**.

Pre-training	**Captured**	**Method Name Prediction**				**Solution Classification**			
Strategy	**Feature**	**GCN**	**GIN**	**GSAGE**	**GAT**	**GCN**	**GIN**	**GSAGE**	**GAT**
Context	**semantic**	**0.5454**	**0.4674**	**0.5038**	**0.6082**	**0.9698**	**0.9649**	**0.9682**	**0.9740**
	syntactic	0.4575	0.4500	0.4644	0.4381	0.9614	0.9560	0.9588	0.9610
Node	**semantic**	**0.4843**	**0.4136**	**0.4404**	**0.5888**	**0.9738**	**0.9711**	**0.9696**	**0.9704**
	syntactic	0.3800	0.3845	0.3660	0.3560	0.9563	0.9562	0.9572	0.9595
VGAE	**semantic**	**0.5988**	**0.4786**	0.3675	**0.5464**	**0.9725**	**0.9663**	**0.9671**	**0.9711**
	syntactic	0.3922	0.4053	**0.3936**	0.4058	0.9711	0.9659	0.9626	0.9705
Best config.		**Semantic = 11/12**				**Semantic = 12/12**			

RQ3 Further Analyses: The goal of this research question is to examine the impact of *model pre-training* on improving GraphCode2Vec's effectiveness on downstream tasks. We also investigate if GraphCode2Vec effectively captures lexical and/or semantic program feature(s). We employ *probing analysis* to analyze if pre-trained GraphCode2Vec models learn the lexical and semantic features required for feature-specific tasks, i.e, that require capturing either or both features to be well-addressed. For instance, Task-4 is the concurrency classification task requiring semantic features. In addition, we conduct an *ablation study* to investigate how the syntactic and semantic information captured by GraphCode2Vec influence its effectiveness on downstream tasks. Finally, we evaluate the *sensitivity of our approach* to the selected *GNN*.

Model Pre-training: We examine if the three pre-training strategies improve the effectiveness of GraphCode2Vec on downstream tasks, using two downstream tasks and all three pre-training strategies (node, context and VGAE) (*see* Table 5.8).

We found that *model pre-training improves the effectiveness of GraphCode2Vec across all tasks*. Pre-training improves its effectiveness by up to 28%, on average. For instance, consider model pre-training with VGAE strategy for method name prediction (*see* Table 5.8). This result implies that model pre-training improves the effectiveness of GraphCode2Vec on downstream SE tasks.

Model pre-training improves the effectiveness of
GraphCode2Vec (by up to 28%, on average) across all tasks.

Probing Analysis: Let us examine if our pre-trained code embedding indeed encodes the desired lexical and semantic program features. To achieve this, we use the lexical embedding and semantic embedding from GraphCode2Vec's pre-training as inputs for probing. In this probing analysis, only the classifier is trainable and GraphCode2Vec is frozen and non-trainable. We use one MLP-layer classifier to evaluate these models on four tasks, Task-1 requires only lexical/syntactic information. However, Task-2 and Task-4 require only semantic information (program dependence). Finally, Task-3 subsumes tasks one and two, such that it requires both syntactic and semantic information.

Our evaluation results show that *GraphCode2Vec's pre-trained code embedding mostly captures the desired lexical and semantic program features for all tested tasks, regardless of the pre-training strategy or GNN configuration.* Table 5.7 highlights the effectiveness of each frozen pre-trained model for each task, configuration and pre-training strategy. Notably, the frozen pre-trained model performed best for the desired embedding for each task in three-quarters (36/48=75%) of all tested configurations. As an example, for tasks requiring semantic information (Task-2 and Task-4), our pre-trained model encoding only semantic information performed best for 88% of all configurations (21/24 cases). This result demonstrates that GraphCode2Vec effectively encodes either or both syntactic and semantic features, this is evidenced by the effectiveness of models encoding desired feature(s) for feature-specific tasks.

GraphCode2Vec effectively encodes the syntactic and/or semantic features, feature-specific models performed best in 75% of cases.

Ablation Study: We investigate the impact of syntactic/lexical embedding and semantic/dependence embedding on addressing downstream tasks. Using method name prediction and solution classification, we examine how removing lexical embedding or dependence embedding during the fine-tuning of GraphCode2Vec's pre-trained model impacts the effectiveness of the approach.

Our results show that *GraphCode2Vec's dependence embedding is important to effectively address our downstream SE tasks.* Table 5.9 presents the ablation study results. In particular, results show that models fine-tuned with only semantic information outperformed those fine-tuned with syntactic features in almost all (23/24 = 96% of) cases. This result demonstrates the effectiveness of dependence embedding in addressing downstream SE tasks.

Results show that dependence/semantic embedding is vital to the effectiveness of GraphCode2Vec on downstream SE tasks.

GNN Sensitivity: This experiment evaluates the sensitivity of our approach to the choice of GNN. Table 5.8 provides details of the GNN sensitivity analysis, tasks and GNN configurations. To evaluate this, we compute the *variance* and *standard deviation (SD)* of the effectiveness of GraphCode2Vec when employing different GNNs.

Our evaluation results show that *GraphCode2Vec is stable, it is not highly sensitive to the choice of GNN.* Table 5.8 shows the details of the SD and variance of our approach for each GNN configuration. Across all tasks, the variance and SD of the GraphCode2Vec is mostly low, it is maximum 0.0064 and 0.0413, respectively.

GraphCode2Vec is stable across GNN configurations, the variance and SD of its effectiveness are very low for all configurations.

5.6 Threats to Validity

External Validity: This refers to the generalizability of our approach and results, especially beyond our data sets, tasks and models. For instance, there is a threat that GraphCode2Vec does not generalize to other (SE) tasks and other Java programs. To mitigate this threat, we have evaluated

GraphCode2Vec using mature Java programs with varying sizes and complexity (*see* Table 5.3), as well as downstream tasks with varying complexities and requirements.

Internal Validity: This threat refers to the correctness of our implementation, if we have correctly represented lexical and semantic features in our code embedding. We mitigate this threat by evaluating the validity of our implementation with probing analysis and ablation studies (*see* Section 5.5). We have also compared GraphCode2Vec to 7 baselines using four (4) major downstream tasks. In addition, we have conducted further analysis to test our implementation using different pre-training strategies and GNN configurations. We also provide our implementation, (pre-trained) models and experimental data for scrutiny, replication and reuse.

Construct Validity: This is the threat posed by our design/implementation choices and their implications on our findings. Notably, our choice of intermediate code representation (i.e., Jimple) instead of source code implies that our approach lacks natural language text (such as code comments) in the (pre-)training dataset. Indeed, GraphCode2Vec would not capture this information as it is. However, it is possible to extend GraphCode2Vec to also capture natural language text. This can be achieved by performing lexical and program dependence analysis at the source code level.

5.7 Conclusion

In this work, we have proposed GraphCode2Vec, a novel and generic code embedding approach that captures both syntactic and semantic program features. We have evaluated it in comparison to the state-of-the-art generic code embedding approaches, as well as specialised, task-specific learning based applications. Using seven (7) baselines and four (4) major downstream SE tasks, we show that GraphCode2Vec is stable and effectively applicable to several downstream SE tasks, e.g., patch classification and solution classification. Moreover, we show that it indeed captures both lexical and dependency features, and we demonstrate the importance of generically embedding both features to solve downstream SE tasks. We also provide our experimental code for replication and reuse:

`https://github.com/graphcode2vec/graphcode2vec`

6 Test Selection for Deep Learning Systems

Testing of deep learning models is challenging due to the excessive number and complexity of the computations involved. As a result, test data selection is performed manually and in an ad hoc way. This raises the question of how we can automatically select candidate data to test deep learning models. Recent research has focused on defining metrics to measure the thoroughness of a test suite and to rely on such metrics to guide the generation of new tests. However, the problem of selecting/prioritising test inputs (e.g. to be labelled manually by humans) remains open. In this work, we perform an in-depth empirical comparison of a set of test selection metrics based on the notion of model uncertainty (model confidence on specific inputs). Intuitively, the more uncertain we are about a candidate sample, the more likely it is that this sample triggers a misclassification. Similarly, we hypothesise that the samples for which we are the most uncertain, are the most informative and should be used in priority to improve the model by retraining. We evaluate these metrics on 5 models and 3 widely-used image classification problems involving real and artificial (adversarial) data produced by 5 generation algorithms. We show that uncertainty-based metrics have a strong ability to identify misclassified inputs, being 3 times stronger than surprise adequacy and outperforming coverage related metrics. We also show that these metrics lead to faster improvement in classification accuracy during retraining: up to 2 times faster than random selection and other state-of-the-art metrics, on all models we considered.

Contents

6.1 Introduction

Deep Learning (DL) systems [115] are capable of solving complex tasks, in many cases equally well or even better than humans. Such systems are attractive because they learn features by themselves and thus require only minimum human knowledge. This property makes DL flexible and powerful. As a result, it is increasingly used and integrated with larger software systems and applications.

Naturally, the adoption of this technology introduces the need for its reliable assessment. In classical, code-based software engineering, this assessment is realised by means of testing. However, the testing of DL systems is challenging due to the complexity of the tasks they solve. In order to effectively test the DL system, we need to identify corner cases that challenge the learned properties. In essence, DL system testing should focus on identifying the incorrectly learned properties and lead to data that can make the systems deviate from their expected behaviour.

To this end, recent research [98, 131, 169, 203] focuses on designing test coverage metrics to measure how thoroughly a set of inputs tests the model. Most of them (e.g. those inspired by neuron coverage [131, 169, 203]) focus on activating various combinations of neurons and on generating new test inputs to increase the proportion of those combinations. Others (e.g. [98]) argue that DL models reflect particular properties of the training data and their behaviour is determined based on the knowledge they acquired during the training phase. As such, they promote coverage metrics that measure the differences across the inputs rather than within the model.

In this work, we focus on the problem of *selecting* test inputs. In DL, test input selection adresses a practical concern: which subset of unlabelled data one should label to discover faults in DL models. This goal differs from previous methods that either measure test thoroughness or generate (artificial) test inputs. Our aim is to help with the manual effort involved when labelling test data (deciding the class of an input). Evidently, data labelling involves extensive manual analysis (due to the large amounts of data required by DL systems), which could be reduced by using only the most likely fault revealing test data. Therefore, to reduce this burden, we aim to identify metrics that help selecting the most interesting (likely to trigger misclassifications) test data.

In the recent years, the scientific community has come up with metrics to support the testing of deep learning models (read more in Section Section 2.2). However, these metrics were studied in different contexts (e.g. adversarial example generation and detection) or testing scenarios (e.g. measuring test thoroughness). All in all, the capability of existing metrics to pinpoint misclassified inputs remains unclear. Thus, our contribution is the evaluation of these metrics from a new perspective; the test input selection. That is serving the purpose of selecting inputs that maximise the chances to trigger misclassifications. To our knowledge, our work is the only one to study this test selection goal.

We postulate that effective metrics should select inputs that are more likely to trigger misclassifications by the model. Experience has shown that classification mistakes are incorrectly learned properties that happen due to overlapping and closely located regions of the feature space. Therefore, the cases residing between the learned categories and their boundaries are the most likely to be the incorrectly learned ones. In view of this, rather than aiming at the coverage of specific neurons [169] or test data diversity [98], we aim at the data with properties that are close to the model boundaries. In other words, we argue that test selection should be directed towards the boundaries of the learned classes.

Accordingly, we consider test selection metrics based on the notion of model uncertainty (low confidence on specific inputs). Intuitively, the more uncertain a model is about a candidate sample, the more likely the sample will trigger misclassification. Similarly, the samples for which the model is the most uncertain are also the most informative ones for learning (should be used to improve the models by retraining). As suggested by Gal and Ghahramani [62], we approximate uncertainty by the variance in prediction probabilities observed by randomly dropping neurons in the network multiple times [200]. We also use the actual model's output probabilities as a certainty measure, which we also combine with dropout variance.

We evaluate these metrics using image classifiers on three datasets, i.e., MNIST, Fashion-MNIST and CIFAR-10, and compare them with respect to previously proposed metrics, i.e., the surprise adequacy metrics [98] and several metrics based on neuron coverage [131, 169]. In particular, we investigate the correlation between the metrics and misclassification on both real and artificial (adversarial) data. We show that uncertainty-based metrics have medium to strong correlations with misclassification when considering real data, and strong correlations when considering a mix of real and adversarial data. In contrast, metrics based on coverage have weak or no correlation, while surprise adequacy has weak correlation.

Interestingly, our results reveal that when considering misclassifications, the prediction probabilities (a simple certainty metric overlooked by previous work) is among the most effective metrics, significantly outperforming surprise adequacy and coverage metrics. However, in the case of retraining, a combination of the dropout variance with prediction probabilities outruns the other metrics in terms of faster improvements in classification accuracy.

Our contributions can be summarised by the following points:

- We propose to perform test input selection based on a set of metrics measuring model uncertainty, i.e., the confidence in classifying unseen inputs correctly. We consider the variance caused by multiple dropouts (i.e. the distribution of the model's output under dropouts), the model's prediction probabilities, and metrics combining the two.
- We perform the first study on the fault revealing ability (misclassification triggering ability) of test selection metrics. We demonstrate that the uncertainty-based metrics challenge the DL models and have medium to strong correlations with misclassification (correlations of approximately 0.3 on real data and 0.6 on real plus adversarial ones). Furthermore, we show that these metrics are significantly stronger than the surprise adequacy and coverage related metrics.
- We also show that model uncertainty can guide the selection of informative input data, i.e., data that are capable of increasing classification accuracy. In particular, when retraining the DL model based on the selected data, the best performing uncertainty metrics achieve up to 2 times faster improvement over random selection and coverage metrics.

6.2 Motivation and Problem Definition

DL systems are known for their capability to solve problems with large input space, by learning statistical patterns from the available data. This is typically the case in computer vision applications (the use case we consider in our experiments), where the goal is to classify images correctly between two classes or more. An interesting characteristics of such problems is that data (i.e. images) usually proliferate. However, to be useful these data also need to be labelled (associated with their correct class). They can then be used either to test the model (check that the DL model predicts the correct label of the image) to (re-)train it (feed new labelled data into the model to improve its predictions). Data labelling is typically performed by manual or non-systematic procedures. This means that DL system developers have to put a lot of effort to produce a DL model of acceptable quality. Our key motivation is to support them in this task by optimizing the effort-reward ratio.

We formulate this problem as follows. Let us assume that developers have access to an arbitrarily large number of inputs (i.e. data without label) and that they can afford to label only an arbitrary number of k inputs. We name *test input selection* the problem of selecting the k most effective inputs to label. Here, effectiveness is measured differently depending on the considered DL development phase:

- When *testing* the trained DL model, effectiveness is measured in terms of fault-revealing ability. Misclassifications being the most obvious defects that occur in DL models, selection methods should maximize the *number of inputs misclassified by the model.*
- When *re-training* the model, the inputs to label should be selected in order to *maximize the performance gain* (e.g. maximize accuracy).

Our goal is to address the test input selection problem by providing objective and measurable ways of identifying effective test inputs. Thus, We thus aim at answering the following two questions:

- *How can we select test inputs to challenge (trigger misclassifications in) a Deep Learning model?*

- *How can we select additional training inputs to improve the performance (increase classification accuracy) of a Deep Learning model?*

We answer these questions by conducting an empirical study evaluating the adequacy of different selection metrics. Based on our results, practitioners can identify which metrics they should use given their goal (testing or retraining their model) and their context (e.g. with or without adversarial data).

6.3 Test Selection Metrics

Our overall goal is to consider a range of metrics related to misclassification and study how effective they are in selecting misclassified inputs. Particularly, we hypothesize that model uncertainty is strongly correlated to misclassification, that is, the more uncertain a model is towards a certain input, the more it is likely to misclassify this input. Dropout variance is the most concrete and simple way to estimate the prediction uncertainty [62], instead of dealing with Bayesian models whose training can often be quite demanding. Other metrics we consider come from the machine learning testing literature (e.g. neuron coverage [169], surprise adequacy [98] and etc.) – please refer to Section 6.3.1 for details. Although they were not meant for test input selection (as we define in this work), they relate to the general concept of test adequacy and remain commonly used to drive test generation. As such, they appear as natural baselines worth of consideration. Since we aim at experimenting with the test input selection problem (our new perspective), we should use the most relevant metrics. The remaining metrics represent complementary properties: neuron boundary coverage is a generalization of neuron coverage; silhouette coefficient is an alternative to surprise adequacy; Kullback-Leiber divergence is another way of measuring dropout variance that estimates the uncertainty of the model, therefore we consider them as well.

6.3.1 Metrics Derived from the Machine Learning Testing Literature

The metrics we retain from the literature were initially directed to measuring test thoroughness (“coverage”) rather than test input selection. Still, they can be used for the latter purpose by selecting the test inputs that reach the highest coverage.

6.3.1.1 Neuron Coverage

Neuron coverage was first proposed in [169] to drive the generation of artificial inputs. It is simply defined as the percentage of neurons that were activated by at least one input of the test set. Accordingly, we define the coverage of a single input as follows.

Definition 1 *Let D be a trained DL model composed of a set N of neurons. The **Neuron Coverage** (NC) of the input x wrt. D is given by*

$$NC(x) = \frac{|\{n \in N \mid activate(n, x)\}|}{|N|}$$

where $activate(n, x)$ holds true if and only if n is activated when passing x into D.

The above definition determines the coverage of an input independently of the other inputs. One can instead define the *additional* neurons covered by x that were not covered during training.

Definition 2 *Let D be a DL model composed of a set N of neurons and trained on a set T of inputs . The **Additional Neuron Coverage** (ANC) of the input x wrt. D is given by*

$$ANC(x) = \frac{|\{n \in N \mid activate(n, x) \wedge \forall x' \in T : \neg activate(n, x')\}|}{|N|}.$$

We also consider the other test thoroughness metrics that extend the concept of NC: **K-Multisection Neuron Coverage** (KMNC), **Neuron Boundary Coverage** (NBC) and **Strong Neuron Activation Coverage** (SNAC). A detailed description of those metrics can be found in their original paper [131].

6.3.1.2 Surprise Metrics

The next two test selection methods are based on surprise adequacy [98]. In their recent paper, Kim et al. [98] proposed two metrics to measure the surprise of a DL model D when confronted to a new input x. The first one is based on kernel density estimation and aims at estimating the relative likelihood of x wrt. the training set in terms of the activation values of D's neurons. To reduce computational cost, only the neurons of a specified layer are considered [98].

Definition 3 *Let D be a DL model trained on a set T of inputs. The **Likelihood-based Surprise Adequacy** (LSA) of the input x wrt. D is given by*

$$LSA(x) = \frac{1}{|A_{N_L}(T_{D(x)})|} \sum_{x_i \in T_{D(x)}} K_H(\alpha_{N_L}(x) - \alpha_{N_L}(x_i))$$

where $\alpha_{N_L}(x)$ is the vector recording the activation values of the neurons in layer L of D when confronted to x, $T_{D(x)}$ is the subset of T composed of all inputs of the same class as x, $A_{N_L}(T_{D(x)}) = \{\alpha_{N_L}(x_i) \mid x_i \in T_{D(x)}\}$, and K_H is the Gaussian kernel function with bandwidth matrix H.

As an alternative, Kim et al. proposed a second metric that relies on Euclidean distance instead of kernel density estimation. The idea is that inputs that are closer to inputs of other classes and farther from inputs of their own class are considered as more surprising. This degree of surprise is measured as the quotient between the distance of the closest input x_a of the same class as x and the distance of the closest input x_b from any other class. Like the LSA metric, all these distances are considered in the activation value space of the inputs.

Definition 4 *Let D be a DL model trained on a set T of inputs. The **Distance-based Surprise Adequacy** (DSA) of the input x wrt. D is given by*

$$DSA(x) = \frac{||\alpha_N(x) - \alpha_N(x_a)||}{||\alpha_N(x_a) - \alpha_N(x_b)||}$$

where

$$x_a = \underset{\{x_i \in T_{D(x)}\}}{\text{argmin}} ||\alpha_N(x) - \alpha_N(x_i)||$$
$$x_b = \underset{\{x_j \in T \setminus T_{D(x)}\}}{\text{argmin}} ||\alpha_N(x_a) - \alpha_N(x_j)||$$

and where $D(x)$ is the predicted class of x by D and $\alpha_N(x)$ is the activation value vector of all neurons of D when confronted to x.

LSA and DSA rely on measuring the density or distance of the clusters formed by the different classes. In essence, any metrics that can discriminate the consistency of clusters might be also candidate metrics for test selection. For instance, we propose to use **Silhouette Coefficient** [185] as another metric. Its advantages are its stability and a limited range of output, i.e. $[-1, 1]$ (whereas LSA and DSA have a priori no upper bound).

Definition 5 *Let D be a DL model trained on a set T of inputs.* ***Silhouette Coefficient*** *(Si) of the input x wrt. D is given by*

$$Si(x) = \begin{cases} 1 - \dfrac{a_x}{b_x}, & a_x < b_x \\ 0, & a_x = b_x \\ \dfrac{b_x}{a_x} - 1, & a_x > b_x \end{cases}$$

where

$$a_x = \frac{1}{|T_{D(x)} \setminus \{x\}|} \sum_{x_i \in T_{D(x)} \setminus \{x\}} ||\alpha_N(x) - \alpha_N(x_i)||,$$
$$b_x = \min_{D(x) \neq D(x_i)} \frac{1}{|T_{D(x_i)}|} \sum_{x_i \in T_{D(x_i)}} ||\alpha_N(x) - \alpha_N(x_i)||.$$

6.3.2 Model Uncertainty Metrics

The starting point for suggesting the use of other selection metrics lies in the hypothesis that test inputs are more challenging (i.e. more likely to be misclassified) as they engender more uncertainty (as opposed to surprise) in the considered DL model.

The prediction probabilities of the classes returned by the model are immediate metrics that can indicate how challenging a particular input is. Indeed, one can intuitively state that more challenging inputs are classified with lower probability, that is, the highest prediction probability output by the model is low. Using prediction probabilities as metrics is mostly overlooked by the literature but remains efficient, as our experiments show.

Definition 6 *Let D be a trained DL model. The* ***maximum probability score*** *of the input x wrt. D is given by*

$$MaxP(x) = \max_{i=1:C} p_i(x)$$

where C is the number of classes and $p_i(x)$ is the prediction probability of x to class i according to D.

Recently, it has been mathematically proven that neuron dropout [200] can be used to model uncertainty [62, 96]. Dropout was initially proposed as a technique to avoid overfitting in neural networks by randomly dropping (i.e. deactivating) neurons during training [200]. This is achieved by incorporating so-called *dropout* layers into the network, which dynamically simulate the deactivation of neurons during a forward pass.

Dropout can be used to estimate the uncertainty of a trained model wrt. a new input x [62]. More precisely, the uncertainty is estimated by passing k times the input x into the model (wherein dropout layers were added) and computing the variance of the resulting prediction probabilities over x. Intuitively, while prediction probabilities can be visualized as the distances of x from the class boundaries estimated by the model, dropout variance represents the variance of these distances induced by the uncertainty of our knowledge about the exact locations of the class boundaries. The motivation towards using dropout variance rather than classification probabilities stems from the observation that some modern deep neural networks are poorly *calibrated* [75], i.e. their prediction probabilities do not correlate with their likelihood of correct classification.

In addition to being a good estimate of model uncertainty [62, 198], dropouts are cheap to compute thanks to their implementation as dropout layers (as opposed to really generating k altered models from the original one, which would be more expensive given the high number of neurons that models include). Formally, let D be an original, well-trained model equipped with dropout layers to simulate random dropping, such that each neuron is dropped out with probability (i.e. dropout rate) r. Given an input x, we pass x into the network k times and denote by $p_i^j(x)$ the prediction probability of x to class i output on the j-th pass. We also denote by $P_i(x) = \{p_i^j(x) | 1 \leq j \leq k\}$ the multiset of prediction probabilities of x assigned to class i resulting from the k passes. Then, the variance of $P_i(x)$ is a good estimate of the uncertainty of D when classifying x in class i [62].

Following our hypothesis that uncertain inputs are more likely to be misclassified, we define a metric derived from dropout variance to assess how much an input x is challenging to D. This *variance score* is a macroscopic view of dropout variance in that it averages the uncertainty of D wrt. x over all classes.

Definition 7 *The* ***variance score*** *of the input x is given by*

$$Var(x) = \frac{1}{C} \sum_{i=1}^{C} \mathrm{var}(P_i(x))$$

where C is the number of classes and var *denotes the variance function.*

A drawback of this metric is that it does not consider the prediction probabilities (thus, the actual distance to class boundaries). To overcome this, we propose a relative metric that normalizes the variance score with the highest probability output by D.

Definition 8 *The* ***weighted variance score*** *of the input x is*

$$Var_w(x) = \big(MaxP(x)\big)^{-1} \cdot Var(x)$$

While variance and weighted variance scores of x can be regarded as quantitative measures of the uncertainty of the model wrt. x, we also propose a nominal alternative. Instead of the variance of prediction probabilities, we focus on the actual class predictions produced by the different mutant models, that is, the classes with the highest probability scores. We construct a normalized histogram of these k class predictions and we compare their distribution with that of a theoretical, worst-case, completely uncertain model, where the class predictions are uniformly distributed over all classes. Thus, in this worst case, the number of mutants predicting that an input x belongs to class i is approximately given by $\frac{k}{C}$.

To compare the actual class prediction distribution with the worst-case distribution, we rely on the discrete version of Kullback-Leibler (KL) divergence. When the uncertainty of D is high (i.e. the mutants often disagree), the KL divergence is low.

Table 6.1: Datasets and DNN models used in our experiments.

Dataset	Model	Optim. method	# layers (convolution, dense)	# neurons (kernels)	Accuracy (%)
MNIST / Fashion-MNIST	MLP	RMSprop	3	1034	98.51 / 89.33
	LeNet	SGD + Nesterov	5	236	99.05 / 89.99
	WLeNet	Adam	5	310	99.57 / 92.88
CIFAR-10	NetInNet	SGD + Nesterov	9	1418	90.77
	VGG10	SGD + Momentum	10	1674	91.99

Definition 9 *The **Kullback-Leibler score** of the input x is*

$$KL(x) = \sum_{i=1}^{C} H_i \ln \frac{H_i}{Q_i}$$

where i is the class label, H is the normalized histogram, or frequencies, of the class predictions for x resulting from the k dropouts and Q is the uniform distribution (i.e. $Q_i = \frac{1}{C}$).

6.4 Experimental Setup

6.4.1 Datasets and Models

We consider three image recognition datasets. *MNIST* [116] contains handwriting number data of 10 classes and is composed of 70,000 images (60,000 for training and 10,000 for testing). *Fashion-MNIST*[1] has clothing images classified into 10 classes and is also composed of 70,000 images, 60,000 and 10,000 for training and testing. *CIFAR-10* [1] has 10 categories of images (cats, dogs, trucks etc.). The dataset has 50,000 images for training and 10,000 for testing.

The three datasets are widely used in research and considered as a good baseline to observe key trends, in addition to requiring affordable computation cost. Furthermore, the diversity of these datasets (in terms of classes and domain concepts) and the used models provides confidence about the generality of our results.

Thanks to the efforts of the research community, these classification problems can today be solved with high accuracy. This characteristic makes these datasets challenging and relevant for us; triggering misclassifications in accurate models is much harder than in inaccurate ones. Indeed, test selection is more beneficial as interesting tests (i.e. misclassified inputs) are rarer within the set of test candidates (thus, when the model has high accuracy). Considering models with low accuracy is not relevant, as in this case it is more likely to select misclassified examples.

Table 6.1 shows the characteristics of the models we use in our experiment. For MNIST and Fashion-MNIST, we use three simple networks, Multi-Layer Perceptron (MLP), LeNet [116] and a modified version LeNet with more kernels (WLeNet). For CIFAR-10, we use tow complex networks, NetInNet[125] and 10-layer *VGG16* [195] – named VGG10 – obtained by removing the top layers and inserting a batch normalization layer after each convolutional layer. The models were trained for 50 epochs (MNIST), 150 epochs (Fashion-MNIST) and 300 epochs (CIFAR-10). The last column in Table 6.1 shows the best accuracy of the models (over the epochs) when trained on the whole training set.

[1]https://github.com/zalandoresearch/fashion-mnist

6.4.2 Objectives and Methodology

6.4.2.1 Test Selection with Real Data

Our first step is to assess the abilities of the studied metrics to select test inputs that can challenge a given DL model D. To achieve this, we determine the correlation between the metrics and misclassification. We encode the 'correctness' of the prediction of D for one particular input x as a binary variable b_x (well- or miss-classified). For each metric, we compute the Kendall correlation between the score given by the metric to all test inputs and their corresponding binary variables. We used Kendall correlation because, being an ordinal association metric, it focuses on how well the metrics rank the misclassified inputs first (irrespective of the actual score values). Thus, if one has to select a limited number of inputs to label and test, one should select the inputs of higher ranking (irrespective of their score). This is important for the test input selection problem, as the metrics should allow an effective selection regardless of the actual budget of inputs to label. Given that we study the correlation between the numeric values returned by the selection metrics and misclassification, random selection is not a relevant baseline in this experiment, as it is not a metric. Actually, since we consider high-accuracy models which yield few misclassifications, random selection is inherently ineffective for test input selection.

6.4.2.2 Test Selection with Adversarial Data

To investigate the fault revealing ability with a larger number of data, we augment our test data with adversarial samples. Adversarial data result from the successive application of minor perturbations to original data with the aim of deceiving a classifier. Adversarial samples have been of major concern [110] and test selection metrics should be robust against them. Moreover, previous research [98, 131] also used adversarial data. To craft adversarial data, we use five established adversarial data generation algorithms: Fast Gradient Sign Method (FGSM) [71], Jacobian-based Saliency Map Attack (JSMA) [168], DeepFool (DF) [148], Basic Iterative Method(BIM) [109] and Carlini-Wagner (CW) [32]. We apply each algorithm separately and add its generated images to the original test set. Thus, we obtain five new datasets. All algorithms except CW generated 10,000 images and thus doubled the size of the test set. Regarding CW, we made it generate 1,000 adversarial images as it is much slower, which necessitated more than one day of computing on our HPC infrastructure. Still, CW remains interesting to study as it is known to apply less perturbation to the original image. Nevertheless, we use the procedure mentioned previously to compute the Kendall correlation between test selection metrics and misclassification, in the five datasets using both original and adversarial images generated by five algorithms.

To further investigate the sensitivity of the metrics on adversarial data, we apply FGSM and CW on 600 images randomly picked from the datasets. As these algorithms iteratively generate adversarial images (by altering them, introducing noise), until they succeed, most (66% to 100%) of the intermediate images are well-classified. Thus, we store 3,603 intermediate images generated by FGSM and 18,148 generated by CW over the iterations and compute their score according to the studied metrics. Since we start from well-classified images and the adversarial generation algorithms work incrementally (at each iteration they generate images that are closer to misclassification) the number of the iteration at which an input was generated reflects its distance from the starting point (a later iteration step signifies a higher chance for misclassification). Therefore, a monotonic relation between the metrics and the iterations signifies a good capability to quantify the likelihood of misclassification (caused by the adversarial images ultimately produced at the end of the process). Hence, we compute the Spearman correlation (statistical test measuring the monotonicity between the studied variables) between the score of the metrics, of these intermediate images, and the iteration number that they were produced.

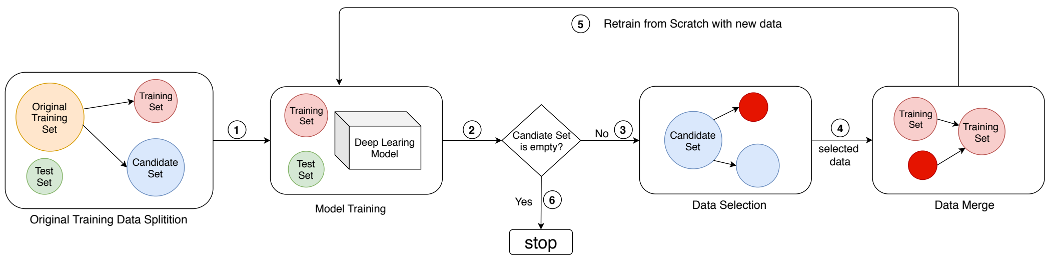


Figure 6.1: Flow process of Data Selection for Retraining

We also study the correlation between the metrics and misclassification when using only adversarial inputs. We pick the intermediate images and the final adversarial images and associate them with a binary variable (well- or miss-classified) and compute the Kendall correlations between the binary variables and the metrics.

6.4.2.3 Data Selection for Retraining

Having studied the adequacy of the metrics to select challenging test inputs, we focus next on how much the metrics can help selecting additional training data effectively. That is, we study whether augmenting the training set with data selected based on the metrics can lead to faster improvement. To do this, we set up an iterative retraining process as shown in Figure 6.1.

At first, we randomly split the original training set into an initial training set of 10,000 images and a candidate set that contains the remaining images. The test set remains untouched. In the first round, we train the model using only the initial training set and compute its accuracy on the test set. After finishing training, we use the best model that we get (over the training epochs) to compute the test selection metrics on the remaining candidate data.

Then, we add (without replacement) a batch of $5,000$ new images (selected by the metrics) from the candidate set to the current training set. The selected images are those that have the highest uncertainty (i.e. lowest score for Si, KL and MaxP, highest score for Var and Var_w) or surprise (LSA or DSA) or coverage (i.e. higher NC, ANC, KBNC, NBC and SNAC). We retrain the models from scratch using the whole augmented training set for a sufficient number of epochs to guarantee convergence (150 epochs for MNIST and Fashion-MNIST, 300 epochs for CIFAR-10) so that we can fairly analyze the different methods. Although incremental training (which re-applies the training algorithms on the current model using the new data) is more efficient computation-wise, current implementations (e.g. within scikit-learn[2]) have shown that incremental training creates biases towards the oldest data, as training algorithms (like stochastic gradient descent) give less importance to new examples over time (due to a decreasing learning rate). This difference can be significant if the new data follow a different distribution than the old data. Thus, incremental training is used when assuming minor concept drift, while training from scratch is used in cases where such assumption cannot be made (or may not hold). This is actually the reason why many companies retrain from scratch [65]. Nevertheless, the purpose of our experiments is not to find the computationally optimal way to incorporate additional training samples, but to make sure that by incorporating additional training samples in the most exhaustive way (to make sure that the model has been trained well enough) yields the best possible (even by a small difference) results. Thus, to avoid making such assumptions, we followed the conservative approach of retraining from scratch to make sure that the old and new training data are treated equally.

We repeat the process for multiple rounds, until the candidate set is empty. We ensure that test data are never used during training or retraining. To account for random variations in the training process,

[2]`https://scikit-learn.org/0.15/modules/scaling_strategies.html#id2`

we repeat the experiments three times and report, for each obtained model, the average (over the three repetitions) of the best accuracy obtained (over all epochs).

To assess the effectiveness of each metric, we observe the evolution of the validation loss and accuracy wrt. the independent test data over the retraining process. Effective metrics should yield fast increases in validation accuracy and fast decreases in validation loss. Although validation loss has usually a strong negative correlation with validation accuracy, it is still important to study both, e.g. to detect overfitting models.

Previous studies have shown that increasing the accuracy of models that already have a high accuracy ($> 90\%$) may result in decreasing their robustness to adversarial attacks [206]. Thus, it is possible that some test selection metrics increase the accuracy during retraining but reduce robustness. To assess this, we also compute the empirical robustness [148] (based on FGSM and 100 randomly picked images) of all models at all retraining rounds. This allows us to check whether there exists a compromise between the metrics (i.e. privileging accuracy or robustness).

6.4.3 Implementation

We tooled our approach on top of Keras and Tensorflow, and used the library Foolbox [182] to generate adversarial images. Our tool, together with our replication package, is available online.[3] The Model training phase was performed on GPU K80 and GPU Volta V100.

When the considered DL model (e.g. VGG16) does not use standard dropout for training, we implement it as Lambda layers. When the DL model includes Dropout layers for training (e.g. WLeNet), we simply keep those Dropout layers working during testing. Thus, we do not alter the model computations per se but rather alternating the model behaviour through the dropout layers.

In every case, the hyperparameters of our method are the dropout rate r (probability of dropping-out neurons) and the number k of forward passes of any input into the network (while randomly dropping neurons on the fly). If r is too large, the original model competence will be significantly degraded, which will result in poor quality. On the contrary, a small r results in too small variations for our method to perform well. For VGG10, we experimentally set the drop rate to 0.25. On the other models, we keep the drop rate as 0.35. We also set $k = 50$ as it appeared as a good trade-off between the estimation of the variance and computation cost.

For Surprise Adequacy and Neuron Coverage, we use the source code available on Github[4]. We re-implemented NC and Neuron-Level Coverage (NLC) based on the source code in an efficient batch-computing way. Finally, we use the IBM robustness framework[5] to compute the empirical robustness of the retrained models.

6.5 Results

6.5.1 Test Selection with Real Data

Table 6.2 shows the Kendall correlation between the metrics and misclassification. We observe that KL, Var, Var_w, MaxP and DSA have a medium degree of correlation, meaning that they can lead to valuable test data, i.e., those causing misclassification. Conversely, we observe that both LSA and Si have a weak correlation to misclassification. All these correlations are statistically significant with a p-value lower than 10^{-05}. Metrics based on neuron coverage have weak to very weak correlations. In

[3] `https://github.com/TestSelection/TestSelection`
[4] `https://github.com/coinse/sadl`, `https://github.com/ARiSE-Lab/deepTest`
[5] https://github.com/IBM/adversarial-robustness-toolbox/

Table 6.2: Kendall correlation between misclassification and the metrics on the *original (real) test data.* Overall, KL and MaxP achieve the strongest correlations.

Model (Dataset)	KL	Var	Var$_w$	MaxP	DSA	LSA	Si	NC	ANC	KMNC	SANC	BNC
MLP (MNIST)	**0.3556**	0.1626	0.1627	0.2804	0.1606	0.1396	0.1326	0.0454	N/A	-0.0181	0.0006	0.0006
LeNet (MNIST)	0.1253	0.1263	0.1282	**0.2076**	0.1273	0.1157	0.1143	-0.0430	0.0017	-0.0464	0.0144	0.0105
WLeNet (MNIST)	**0.2774**	0.0903	0.0905	0.0990	0.0900	0.0766	0.0872	0.0722	N/A	0.0308	0.0047	0.0068
MLP (Fashion)	**0.4107**	0.3222	0.3339	0.3519	0.322	0.0414	0.1639	0.2112	0.0113	0.1876	0.0171	0.0177
LeNet (Fashion)	0.3048	0.2784	0.3103	**0.3369**	0.3059	0.1542	0.2601	-0.0234	0.0058	0.0598	0.0057	0.0067
WLeNet (Fashion)	**0.4156**	0.2896	0.2962	0.3133	0.2941	0.0949	0.2551	0.2429	-0.0139	0.1760	0.0205	0.0188
VGG10 (CIFAR)	0.3404	0.2818	0.2911	**0.3647**	0.2661	0.1964	0.2560	-0.0101	-0.0087	-0.1092	0.0404	0.0329
NetInNet (CIFAR)	**0.4366**	0.3337	0.3371	0.339	0.3208	0.2076	0.3302	-0.0236	N/A	-0.0228	0.0132	0.0228

particular, we could not compute the correlation of ANC for three models because, in these models, the test set does not cover new neurons that the training set did not cover already.

Overall, these results indicate that KL, Var, Var_w, $MaxP$ and DSA correlate better with misclassifications. More precisely, KL and $MaxP$ appear as the best metrics for test selection, being up to 3 times more correlated to misclassification than all the other metrics. In the particular case of the MNIST dataset models, these two metrics are the only ones to achieve a medium correlation, while the other metrics reach only weak or very weak correlations. Nevertheless, the best correlations we found are only moderate, meaning that none of the metrics can perfectly distinguish between well-classified and misclassified inputs.

KL and $MaxP$ are the best metrics to discriminate misclassified real data from well-classified ones. They correlate with misclassification up to 3 times more than the others.

6.5.2 Test Selection with Adversarial Data

6.5.2.1 Mix of Real and Adversarial Data

Figure 6.2 records the Kendall correlation between the metrics and misclassification when the original test data set is augmented with each adversarial dataset (separately). Interestingly, the correlations of all metrics are stronger than they were with real data only. KL, Var, Var_w, MaxP, Si and DSA now have a strong degree of correlation with misclassifications in most cases, while the correlations of LSA reach only moderate levels. Overall, MaxP achieves the stronger correlations regardless of the algorithm used, except for the WLeNet-MNIST where KL gets even stronger correlations. As for metrics based on neuron coverage, their correlations remain weak overall and can be positive or negative. Even on a model-by-model basis, no general tendency tends to appear. Quite surprisingly, NC performs better than KMNC, NBC and SNAC, and even achieves moderate/strong correlations on the two WLeNet models.

Nonetheless, the overall strengthening of the correlations can be explained by the fact that adversarial images have some form of artificial noise that the classifier never experienced during training. This noise makes the classifier less confident on how to deal with them, a fact reflected by the metrics. We also infer that adversarial data do not form a challenging scenario to evaluate test selection methods (as performed by related work [98, 131, 169]). Given that the adversarial images are misclassified, it is possible that the metrics are even more appropriate to distinguish adversarial and real data than they are to differentiate well- and miss-classified data.

Uncertainty- and surprise-based metrics can discriminate well-classified real inputs from all (real and adversarial) misclassified inputs. They can achieve this with more ease as misclassified adversarial inputs are added. Overall, MaxP reaches the strongest correlations (between 0.64 and 0.78).

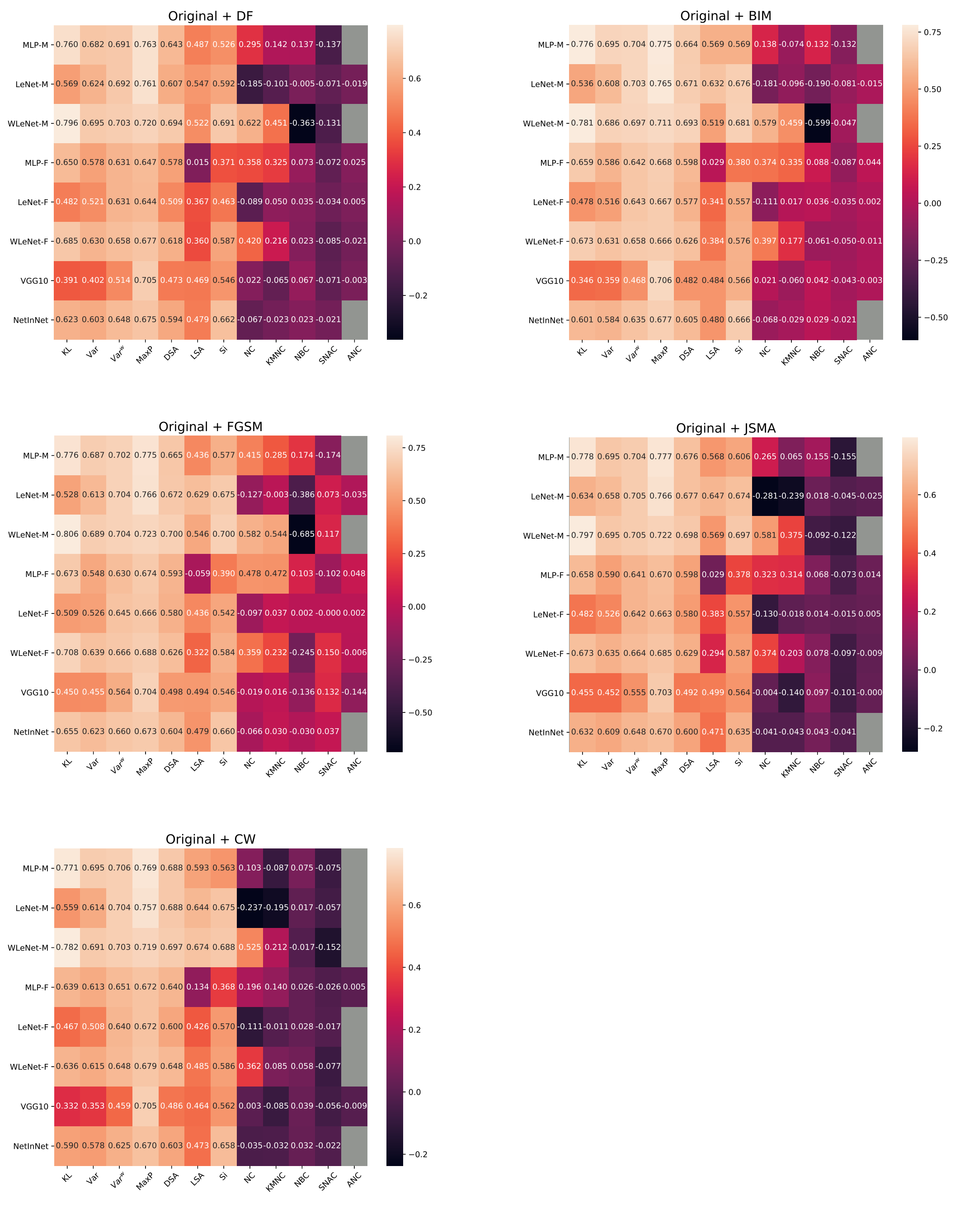
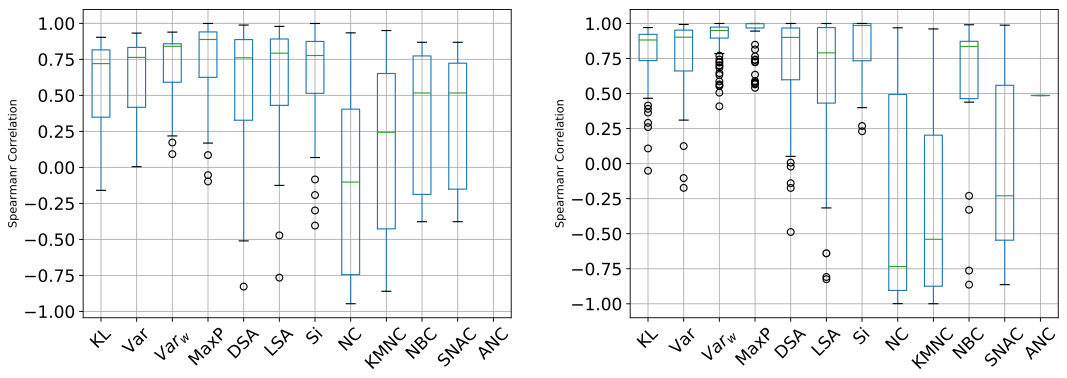


Figure 6.2: Heatmap showing the Kendall correlation between misclassification and the metrics on a *mix of real and adversarial (misclassified) data*, obtained using five different algorithms. The lighter the color the better. Grey parts in ANC correspond to no increase in neuron coverage. Overall, MaxP achieves the strongest correlations.



(a) using 18,418 intermediate images from CW attacking (b) using 3,603 intermediate images from FGSM attacking

Figure 6.3: Spearman rank-order correlation between the metrics and the iteration number of the images generated by the adversarial algorithms. The iteration number reflects the distance from a misclassification and, thus, high correlation suggests that the metrics reflect well the likelihood of misclassification.

Table 6.3: Kendall Correlation between misclassification and the metrics on adversarial data generated by CW (*mix of well-classified (intermediate) and misclassified (final) adversarial images*).

Model (Dataset)	KL	Var	Var_w	MaxP	DSA	LSA	Si	NC	ANC	KMNC	NBC	SNAC
MLP(MNIST)	0.5070	0.5282	0.6261	**0.6589**	0.5265	0.4419	0.2983	-0.0458	N/A	-0.1617	-0.0013	-0.0013
LeNet (MNIST)	0.1011	0.132	0.4474	0.5784	0.5917	0.5465	**0.5925**	-0.0958	N/A	-0.0631	-0.0559	-0.0284
WLeNet (MNIST)	0.4417	0.474	0.5255	**0.538**	0.4600	0.3041	0.3018	0.0568	N/A	-0.1545	-0.0837	N/A
MLP(Fashion)	0.2505	0.2428	0.3631	**0.4628**	0.3213	0.1394	0.1059	-0.0553	0.0103	-0.0336	-0.0547	-0.0715
LeNet (Fashion)	0.0994	0.1233	0.3126	0.4370	**0.4425**	0.2777	0.4381	0.0191	0.0201	0.0199	0.0545	0.0545
WLeNet (Fashion)	0.2818	0.2908	0.3855	**0.5244**	0.2618	0.2109	-0.0528	0.0597	N/A	-0.0324	-0.0059	-0.0145
VGG10 (CIFAR)	0.0606	0.0599	0.153	**0.5031**	0.2643	0.2106	0.2720	0.0434	0.0585	0.0640	-0.0093	0.0117
NetInNet (CIFAR)	0.1380	0.1094	0.2027	**0.4560**	0.1751	0.0453	0.2968	0.0055	N/A	-0.0150	0.0150	-0.001

6.5.2.2 Well- and Miss-Classified Adversarial Data

Figure 6.3 shows, for each metric, boxplots representing the statistical distribution (over all models and images) of the Spearman correlation between the number of the iteration at which the image was produced and the metric value for this image. Metrics based on uncertainty and surprise achieve strong correlations (Var$_w$ being the best in this regard), meaning that they are close to being monotonous over the iterations and thus capture well the adversarial generation process. On the contrary, the metrics based on neuron coverage reach very weak or irregular correlations.

Table 6.3 and Table 6.4 demonstrates the Kendall correlations between misclassification and the metrics computed on the intermediate (mostly well-classified) and final (misclassified) images generated by CW and FGSM, respectively. When considering FGSM, the correlations are similar to what they were when mixing real data with adversarial (misclassified) data. In the case of CW, however, they get weaker, although they remain medium to strong for some metrics (Var$_w$, MaxP, LSA and DSA). In particular, the correlations of KL are disappointing although this metric performed well in the previous experiments. These regressions can be explained by the fact that CW is known to generate smaller perturbations than the other adversarial algorithms. Thus, the difference between the intermediate images and the final images are smaller than what they are in the FGSM case.

When confronted with adversarial inputs only, the test selection metrics lose part of their capability. This is due to the inherent noise introduced by the adversarial generation algorithms. MaxP still achieves the strongest correlations overall, outperforming the other metrics in 13/16 cases.

Table 6.4: Kendall Correlation between misclassification and the metrics on adversarial data generated by FGSM (*mix of well-classified (intermediate) and misclassified (final) adversarial images*).

Model (Dataset)	KL	Var	Var_w	MaxP	DSA	LSA	Si	NC	ANC	KMNC	NBC	SNAC
MLP(MNIST)	0.6984	0.6459	0.6989	**0.7125**	0.5902	0.3736	0.3242	0.1828	N/A	0.1301	0.0237	0.0237
LeNet (MNIST)	0.3009	0.4176	0.7078	**0.7081**	0.6327	0.5779	0.6326	-0.1206	N/A	-0.0361	-0.2202	-0.1007
WLeNet (MNIST)	**0.6951**	0.5840	0.6927	0.6905	0.5802	0.3364	0.5418	0.2901	N/A	0.3282	-0.3831	-0.1956
MLP(Fashion)	0.576	0.2429	0.5541	**0.6645**	0.4527	0.0595	0.1444	0.1051	N/A	0.2732	N/A	N/A
LeNet (Fashion)	0.2848	0.3315	0.6410	**0.6848**	0.5292	0.3563	0.4802	-0.0458	-0.0002	0.0112	-0.0514	-0.0514
WLeNet (Fashion)	0.6178	0.4869	0.6064	**0.6817**	0.4469	0.2845	0.0651	0.1059	N/A	0.0568	-0.1779	-0.1703
VGG10 (CIFAR)	0.2325	0.2480	0.4526	**0.7014**	0.4346	0.3610	0.3969	-0.0346	-0.0273	0.0364	-0.1121	-0.1138
NetInNet (CIFAR)	0.538	0.5332	0.6236	**0.6732**	0.4989	0.3411	0.6047	-0.0168	N/A	0.0108	-0.0108	0.0069

6.5.3 Data Selection for Retraining

Figure 6.4 shows the best accuracy achieved of each retraining round by augmenting, iteratively, the training data with 5,000 data selected according to the different metrics. Here it must be noted that, while the raw accuracy values may seem to have small differences, they are due to the high initial accuracy of the model. Improving beyond this level is challenging.

Overall, we see that uncertainty and surprise metrics outperform those based on neuron coverage, which are comparable to random selection. For example, at the 3rd training augmentation round and for model WLeNet applied on Fashion-MNIST, Var achieves a gain in accuracy (compared to the accuracy of the initial training set) more than 45% higher than the best coverage-based metric – ANC – (+2.2% vs +1.5%) on WLeNet (Fashion), while the accuracy increases by +3% from the initial training set to the final (whole) training set.

On NetInNet applied to CIFAR-10 and at the 3rd round, Var achieves a gain in accuracy more than 20% than ANC does (+8.4% vs +6.9%), while the accuracy increases by +10.4% from the initial training set to the final training set. We observe similar conclusions when validation loss is considered. Indeed, metrics based on neuron coverage lead to slower decreases (similar to random selection), which reveals the inappropriateness of these metrics to select data for retraining.

In addition to the metrics considered so far, we augment Var and KL with a tie-breaking method: when two inputs have the same Var or KL scores, we select the input that has the lowest $MaxP$ score. Interestingly, those two new metrics (denoted by Var_p and KL_p) further improve the increase in accuracy of five models out of eight, and the decrease in validation loss in four models. Overall, those new metrics increase the accuracy up to two times faster than coverage metrics and random selection. Compared with the other uncertainty metrics, the additional gain is not significant, though it keeps the merit to exist. Thus, should one require the use of a single metric, KL_p and Var_p would appear as effective choices.

For each model and metric, we computed the evolution of the empirical robustness over the retraining rounds. Overall, we observed that the robustness score barely varies over the rounds, regardless of the considered model and metric. Indeed, the largest gap across all models and metrics is 0.022, which is insignificant. Moreover, we cannot infer that any of the metric is comparatively best or worst than the others in this regard, as the variations are not monotonous. Thus, uncertainty and surprise-based metrics can increase accuracy faster than coverage-based metrics without compromising robustness.

Uncertainty-based and surprise-based metrics, in particular the tie-breaking metrics KL_p and Var_p, are the best at selecting retraining inputs and lead to improvements up to 2 times faster than random selection. They achieve this without significant variations of the robustness ($<$ 0.022).

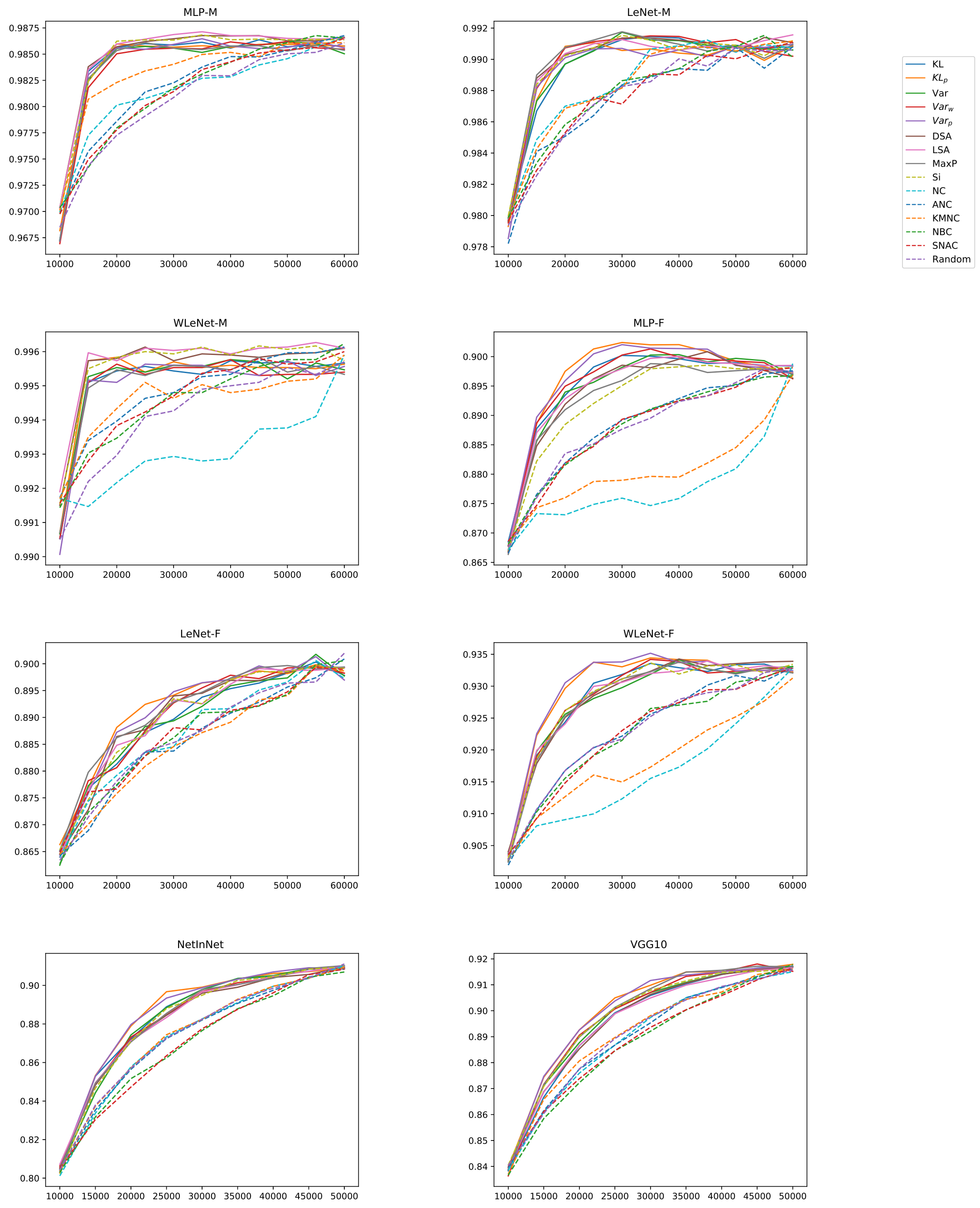


Figure 6.4: Validation accuracy over a fixed set of 10,000 original test data and achieved by successively augmenting the training data with 5,000 data (at each retraining round) selected by the different metrics. X-axis denotes the size of the training set at each round, while Y-axis shows the average accuracy over three repetitions (variance is less than 10^{-5}).

6.5.4 Threats to Validity

Threats to internal validity concern the implementation of the software constituents of our study. Some are addressed by the fact that we reuse existing model architectures with typical parameterizations. The resulting models obtain a high accuracy on state-of-the-art datasets used as is (including their splitting into training and test sets), which indicates that our setup was appropriate.

We implemented dropout “from scratch” (i.e. as Lambda layers) in one case and, in the other cases, we reused the implementation natively embedded in the training process. The use of these two alternatives increases our confidence in the validity of our results. Finally, the implementation of the different metrics was tested manually and through various experiments. Moreover, we reused available implementations of the surprise-based and coverage-based metrics. Regarding LSA, it has been shown that the choice of the layers has an impact on the adequacy of the metric [98]. However, Kim et al. could find no correlation with the depth of the layer. As such, we make the same choice as Kim et al. and compute LSA on the deepest hidden layer.

The threats to external validity originate from the number of datasets, models and adversarial generation algorithms we considered. The settings we used are established in the scientific literature and allow the comparison of our approach with the related work. Performing well on such established and generic datasets is a prerequisite for real-world applications, which generally exhibit biases inherent to their application domain. The replication and the complementation of our study are further facilitated by the black-box nature of uncertainty metrics: all of them necessitate only the prediction probabilities to be computed.

Construct validity threats originate from the measurements we consider. We consider the correlation between the studied metrics and misclassification, which is a natural metric to use (and is in some sense equivalent to the fault detection and test criteria relations studied by software engineering literature [8, 69]).We also compare with surprise adequacy [98] and coverage metrics [131, 169], which are the current state-of-the-art methods.

6.5.5 Discussion and Lessons Learned

Our experimental results shed some light on the ability of existing metrics (coverage- and uncertainty-based) to drive the selection of test inputs.

Starting with DeepXplore [169], previous research advocates that increasing neuron coverage is a good way to perform “better” testing and has been using this criterion for test input generation. Additionally, in traditional (code-based) software engineering, coverage metrics (like statement and branch coverage) are commonly used to guide test generation/selection. It is therefore natural for software engineers to consider neuron coverage for test selection in DL systems as well. Another advantage of neuron coverage is that it provides a natural end-point when testing DL systems, viz. reaching 100% coverage. Yet, as in traditional (code-based) software, finding an adequate stopping criterion for testing DL systems remains an open problem. Our results confirm that achieving 100% of neuron coverage does not guarantee the absence of bugs, just like achieving 100% of statement coverage in traditional software does not. Even worse, coverage-based metrics exhibit weak correlations to misclassification, sometimes weaker than random selection. This brings an important message to the community: the misclassified inputs are not necessarily those that cover new neurons. Overall, while coverage-based metrics are convenient driving criteria for test input generation, different metrics should be used for test selection.

Regarding the remaining metrics (i.e. those based on uncertainty and surprise adequacy), our results provide new significant findings. When selecting test inputs to trigger misclassifications, the highest class probability – a simple metric often overlooked in the literature – performs the best regardless of the nature of the inputs (real or adversarial). Thus, we show that this simple metric forms a strong baseline for future research and that developers can rely on it as all-rounder test selection metrics. Another lesson is that dropout variance, the state-of-the-art metric to estimate model uncertainty, can be improved by normalizing the variance score with the highest class probability (yielding the weighted variance score). This is revealed by the fact that weighted variance has a stronger correlation than dropout variance in all our experiments.

When selecting inputs to retrain the model, we observe that combining KL divergence or weighted variance with the highest class probability yields consistently better results than the other metrics, although by a small margin. Thus, the difference between the uncertainty (and surprise adequacy) metrics is rather observed when selecting inputs for testing.

Another important finding is that some metrics (like KL divergence) are particularly sensitive to the noise introduced by adversarial data and significantly lose their capability (to distinguish well-classified and misclassified data) when confronted to adversarial data only. Indeed, the results of Section 6.5.2 indicate that introducing misclassified adversarial data into the test set yields a stronger correlation between the uncertainty metrics and misclassification. This means that the adversarial examples engender more uncertainty than real ones. This is because most adversarial algorithms aim at achieving misclassification while minimizing input perturbation. Making the model misclassify those examples with high confidence (low uncertainty) model is not part of the objective function of those algorithms, although some studies (e.g. [71]) have shown that this may happen incidentally. Our results in Section 6.5.2.1 confirm that the uncertainty of the model increases over the iteration of the adversarial algorithm, due to the increasing noise it introduces over the iterations.

6.6 Conclusion

We considered test selection metrics for deep learning systems based on the concept of model uncertainty. We experimented with these metrics and compared them with surprise adequacy and coverage related metrics wrt to their fault revealing ability, i.e., ability to trigger missclassifications.

Overall, our findings can be summarised by the following points:

- When dealing with original data, uncertainty metrics (in particular, KL and MaxP) perform best, significantly better than previously proposed metrics (coverage based and surprise adequacy).
- When dealing with a mix of original and adversarial data, MaxP – a simple certainty metric often overlooked by the literature – is the most effective. Additionally, uncertainty-based metrics are also effective at test selection, independent from the training set (coverage based and surprise adequacy metrics fall behind).
- Our results also revealed that the use of adversarial data in testing-related experiments should be performed with caution. All the studied metrics experience significant performance differences when considering original, adversarial or a mix of them.
- We also demonstrated that the metrics and particularly KL_p and Var_p, lead to major classification accuracy improvement (when selecting data for retraining), achieving a gain in accuracy of up to 80% higher than the previously proposed metrics and random selection.

Our work forms an essential step towards a long-term goal of equipping researchers and practitioners with test assessment metrics for DL systems. These automatic data selection metrics pave the way for the systematic and objective selection of test data, which may lead to standardised ways of measuring test effectiveness.

7 Conclusions and Future Work

This chapter revisits the main contributions of this dissertation and points to future research work.

7.1 Conclusions

The dissertation demonstrated the usefulness of Mutation Testing in the era of computational Artificial Intelligence: the traditional software systems and the intelligence software systems. This dissertation comes up with a novel approach to code representation to support a wide range of Software Engineering tasks.

More precisely, we have the four(4) following contributions:

1) Adapting Mutation Testing for the evolving systems. Mutation Testing is quite expensive and is difficult for evolving systems in Continuous Integration. We adapt Mutation Testing and focus on the mutants (commit-aware/relevant mutants) related to the code change to test program behavior change and reduce the mutation testing cost. We evaluate the commit-aware mutants using C and Java programs. As a finding, most mutants are useless to test the code change. Commit-aware mutants can effectively test code change and reduce the computing resource cost.

2) A novel machine learning approach, *MuDelta*, to select the mutants related to code change for the evolving system. Since we find that relevant-change mutants are a small portion of the mutants and can speed up the mutation testing, we develop a machine learning approach to label the relevant mutants to test the program. Our experiments show our technique can improve Mutation Testing a lot.

3) A novel code embedding approach integrating code syntax and semantics, GraphCode2Vec, to support the SE downstream tasks. The researchers usually use domain knowledge to design the features for the code data, which is not general. The work proposes a code embedding approach via leveraging code syntax and semantic information. Our experiments show the advantages of GraphCode2Vec, achieving better or comparable results than the state-of-the-art methods.

4) An in-depth empirical study on test data selection for DL systems based on uncertainty-based/mutation metrics. Deep Learning (DL) systems play an essential role as one component of software systems in practice. The existed testing techniques are not supportive enough to test the program. The work studies how to select good test data or training data for DL systems. We evaluate the different testing uncertainty-based metrics and find that the uncertainty-based metrics can effectively choose the informative data for DL systems.

7.2 Future Work

We now summarize potential future directions that are in line with this dissertation.

1. **Mutation Testing Automation in Continuous Integration.** Mutation Testing automation is still challenging because the current tools are not designed well for the evolving systems, which still requires manual effort. We also have noticed that few Mutation Testing frameworks can fully support the state-of-the-art mutation testing techniques, e.g., automatically reporting valuable mutants. Another problem is that some data science programming language lacks Mutation Testing tool support automatically. For example, Python does not have a mature tool like Pitest for Java. Therefore, we need to integrate SOTA techniques to develop a multiple-language-support tool to automate mutation testing.
2. **Leveraging Mutation Testing to explore AI robustness.** Though Mutation Testing has contributed to deep learning testing, little attention focuses on the relationship of the robustness between the behaviours of the mutants and the original models. Current DL Mutation Testing mainly evaluates the model performance, e.g., accuracy. Mutation testing score for DL systems is also defined based on the performance metrics. Robustness is lacking, and it may be meaningful to explore such a direction.
3. **Encoding Dynamic Features of Program.** SOTA code representation approaches learn code embedding via big models and massive code static data. However, the dynamic features of the program are ignored by these approaches. It is questionable for these static code embedding models to contain how much dynamic information. Many software engineering tasks depend on the program dynamic behaviours, e.g., flaky tests and running-time crashes. Therefore, it should benefit these dynamic tasks if we can encode the program dynamic behaviours into the vectors.

Bibliography

[1] Alex Krizhevsky and Vinod Nair and Geoffrey Hinton. The CIFAR-10 dataset.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[6] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014.

[7] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[8] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 402–411, 2005.

[9] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[10] Taweesup Apiwattanapong, Raúl A. Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. MATRIX: maintenance-oriented testing requirements identifier and examiner. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), 29-31 August 2006, Windsor, United Kingdom*, pages 137–146, 2006.

[11] Andrea Arcuri and Lionel C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1–10. ACM, 2011.

[12] R. Barandelaa and E. Rangela. Strategies for learning in class imbalance problems. 2002.

[13] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[14] Richard E Bellman. *Adaptive control processes*. Princeton university press, 2015.

[15] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3589–3601, Red Hook, NY, USA, 2018. Curran Associates Inc.

[16] Yoshua Bengio and Jean-Sébastien Senecal. Quick training of probabilistic neural nets by importance sampling. In Christopher M. Bishop and Brendan J. Frey, editors, *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, volume R4 of *Proceedings of Machine Learning Research*, pages 17–24. PMLR, 03–06 Jan 2003. Reissued by PMLR on 01 April 2021.

[17] David W. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Software Eng.*, 23(8):498–516, 1997.

[18] David W. Binkley, Nicolas Gold, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. ORBS: language-independent program slicing. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 109–120. ACM, 2014.

[19] David W. Binkley, Nicolas E. Gold, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. ORBS and the limits of static slicing. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*, pages 1–10, 2015.

[20] David W. Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 177–186. IEEE Computer Society, 2005.

[21] David W. Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.

[22] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 334–344, 2013.

[23] Marcel Böhme and Abhik Roychoudhury. Corebench: studying complexity of regression errors. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 105–115, 2014.

[24] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., USA, 1990.

[25] Timothy Alan Budd and Dana Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, 1982.

[26] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1186–1197. IEEE, 2021.

[27] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*, 2020.

[28] Mark Anthony Cachia, Mark Micallef, and Christian Colombo. Towards incremental mutation testing. *Electr. Notes Theor. Comput. Sci.*, 294:2–11, 2013.

[29] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[30] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 964–974, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 3–14, New York, NY, USA, 2017. ACM.

[32] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[33] Marek Chalupa. Slicing of llvm bitcode. *Masaryk Univ*, 2016.

[34] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.

[35] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. Killing stubborn mutants with symbolic execution, 2020.

[36] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Mart: a mutant generation tool for LLVM. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 1080–1084, 2019.

[37] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Muteria: An extensible and flexible multi-criteria software testing framework. In *In AST '20: International Conference on Automation of Software Test (AST '20), October 7–8, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 4 pages*, 2020.

[38] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608, 2017.

[39] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Henry Coles. Pitest mutators. `http://pitest.org/quickstart/mutators/`. Online; accessed 25 May 2020.

[41] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 449–452, 2016.

[42] Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[43] Chris Cummins, Hugh Leather, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, and Michael O'Boyle. Deep data flow analysis. *arXiv preprint arXiv:2012.01470*, 2020.

[44] Fabiano Cutigi Ferrari, Alessandro Viola Pizzoleto, and Jeff Offutt. A systematic review of cost reduction techniques for mutation testing: Preliminary results. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, 2018.

[45] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 77–101, Berlin, Heidelberg, 1995. Springer-Verlag.

[46] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[48] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[49] I. M. M. Duncan and D. J. Robson. Ordered mutation testing. *SIGSOFT Softw. Eng. Notes*, 15(2):29–30, apr 1990.

[50] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010.

[51] Robert B. Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 549–552, 2007.

[52] Giorgio Fagiolo. Clustering in complex directed networks. *Phys. Rev. E*, 76:026107, Aug 2007.

[53] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[54] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. Flakify: A black-box, language model-based predictor for flaky tests. *arXiv preprint arXiv:2112.12331*, 2021.

[55] Reuben Feinman, Ryan R. Curtin, Saurabh Shintre, and Andrew Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[56] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. Deepgini: Prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 177–188, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[58] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[59] Martin Fowler. Continuous integration. `https://martinfowler.com/articles/continuousIntegration.html`. Online; accessed 10 February 2020.

[60] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012.

[61] Jerome H Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

[62] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pages 1050–1059, 2016.

[63] Abel Garcia and Cosimo Laneve. Jada–the java deadlock analyser. *Behavioural Types: from Theories to Tools*, pages 169–192, 2017.

[64] A. Garg, M. Ojdanic, R. Degiovanni, T. Chekam, M. Papadakis, and Y. Le Traon. Cerebro: Static subsuming mutant selection. *IEEE Transactions on Software Engineering*, (01):1–1, jan 5555.

[65] Salah Ghamizi, Maxime Cordy, Martin Gubri, Mike Papadakis, Andrey Boystov, Yves Le Traon, and Anne Goujon. Search-based adversarial testing and improvement of constrained credit scoring systems. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[66] Sahar Ghannay, Benoit Favre, Yannick Esteve, and Nathalie Camelin. Word embedding evaluation and combination. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 300–305, 2016.

[67] Shlok Gilda. Source code classification using neural networks. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6. IEEE, 2017.

[68] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[69] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 302–313, 2013.

[70] Yoav Goldberg. Assessing bert's syntactic abilities. *arXiv preprint arXiv:1901.05287*, 2019.

[71] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[72] Emily Goodwin, Koustuv Sinha, and Timothy J. O'Donnell. Probing linguistic systematicity. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1958–1969, Online, July 2020. Association for Computational Linguistics.

[73] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *IEEE Trans. Reliab.*, 66(3):854–874, 2017.

[74] Kathrin Grosse, David Pfaff, Michael T. Smith, and Michael Backes. The limitations of model uncertainty in adversarial settings. *CoRR*, abs/1812.02606, 2018.

[75] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1321–1330. JMLR. org, 2017.

[76] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[77] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[78] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. Comparing mutation testing at the levels of source code and compiler intermediate representation. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 114–124, 2019.

[79] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 212–222, New York, NY, USA, 2011. Association for Computing Machinery.

[80] Mark Harman and Peter W. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*, pages 1–23. IEEE Computer Society, 2018.

[81] Benjamin Heinzerling and Michael Strube. BPEmb: Tokenization-free pre-trained subword embeddings in 275 languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA).

[82] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 523–534, 2016.

[83] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 518–529, New York, NY, USA, 2020. Association for Computing Machinery.

[84] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[85] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[86] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161, 2019.

[87] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*, 2019.

[88] Chao Huang, Junbo Zhang, Yu Zheng, and Nitesh V. Chawla. Deepcrime: Attentive hierarchical recurrent networks for crime prediction. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, page 1423–1432, New York, NY, USA, 2018. Association for Computing Machinery.

[89] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[90] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[91] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[92] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[93] Dan Jurafsky and James H. Martin. *Speech & language processing.* 2021.

[94] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[95] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

[96] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *NIPS*, 2017.

[97] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.

[98] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *ICSE '19 (to appear)*, 2019.

[99] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, 2014.

[100] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Trans. Software Eng.*, 44(4):308–333, 2018.

[101] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Employing second-order mutation for isolating first-order equivalent mutants. *Softw. Test., Verif. Reliab.*, 25(5-7):508–535, 2015.

[102] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[103] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[104] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999.

[105] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI '95 - Vol. 2*, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[106] Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, pages 179–186. Citeseer, 1997.

[107] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. Shadow symbolic execution for testing software patches. *ACM Trans. Softw. Eng. Methodol.*, 27(3):10:1–10:32, 2018.

[108] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[109] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[110] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *CoRR*, abs/1611.01236, 2016.

[111] Bob Kurtz, Paul Ammann, Marcio E Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014.

[112] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 571–582, 2016.

[113] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *arXiv preprint arXiv:1605.07648*, 2016.

[114] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435. IEEE, 2017.

[115] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[116] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[117] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. Assessing transition-based test selection algorithms at google. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 101–110, 2019.

[118] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Softw. Test. Verification Reliab.*, 23(8):613–646, 2013.

[119] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 220–229. IEEE, 2009.

[120] Shaohua Li, Jun Zhu, and Chunyan Miao. A generative word embedding model and its low rank positive semidefinite solution. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1599–1609, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[121] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 169–180. ACM, 2019.

[122] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics.

[123] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[124] Bill Yuchen Lin, Seyeon Lee, Rahul Khanna, and Xiang Ren. Birds have four legs?! NumerSense: Probing Numerical Commonsense Knowledge of Pre-Trained Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6862–6868, Online, November 2020. Association for Computational Linguistics.

[125] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[126] Pedro G. Lind, Marta C. González, and Hans J. Herrmann. Cycles and clustering in bipartite networks. *Phys. Rev. E*, 72:056127, Nov 2005.

[127] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[128] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[129] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4765–4774, 2017.

[130] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. Deepct: Tomographic combinatorial testing for deep learning systems. In *SANER '19 (to appear)*, 2019.

[131] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131. ACM, 2018.

[132] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2018.

[133] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 175–186, New York, NY, USA, 2018. Association for Computing Machinery.

[134] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. *MuDelta: Delta-Oriented Mutation Testing at Commit Time*, page 897–909. IEEE Press, 2021.

[135] Wei Ma, Thomas Laurent, Miloš Ojdanić, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. Commit-aware mutation testing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–405, 2020.

[136] Wei Ma, Thomas Laurent, Milos Ojdanic, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. Commit-aware mutation testing. In *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, ICSME*, 2020.

[137] Wei Ma, Mike Papadakis, Anestis Tsakmalis, Maxime Cordy, and Yves Le Traon. Test selection for deep learning systems. *ACM Trans. Softw. Eng. Methodol.*, 30(2), jan 2021.

[138] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. Graphcode2vec: Generic code embedding via lexical and program dependence analyses. *arXiv preprint arXiv:2112.01218*, 2021.

[139] Dongyu Mao, Lingchao Chen, and Lingming Zhang. An extensive study on cross-project predictive mutation testing. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 160–171, 2019.

[140] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. Time to clean your test objectives. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 456–467. ACM, 2018.

[141] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 235–245, 2013.

[142] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. Software engineering for ai-based systems: A survey. *arXiv preprint arXiv:2105.01984*, 2021.

[143] Julian J McAuley, Luciano da Fontoura Costa, and Tibério S Caetano. Rich-club phenomenon across complex network hierarchies. *Applied Physics Letters*, 91(8):084103, 2007.

[144] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[145] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[146] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[147] Ron Milo, Nadav Kashtan, Shalev Itzkovitz, Mark EJ Newman, and Uri Alon. On the uniform generation of random graphs with prescribed degree sequences. *arXiv preprint cond-mat/0312028*, 2003.

[148] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *CVPR*, pages 2574–2582. IEEE Computer Society, 2016.

[149] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[150] T Nathan Mundhenk, Daniel Ho, and Barry Y Chen. Improvements to context based self-supervised learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9339–9348, 2018.

[151] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 351–360, Leipzig, Germany, 10-18 May 2008.

[152] Curtis G. Northcutt, Anish Athalye, and Jonas Mueller. Pervasive label errors in test sets destabilize machine learning benchmarks. In *Proceedings of the 35th Conference on Neural Information Processing Systems Track on Datasets and Benchmarks*, December 2021.

[153] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, apr 1996.

[154] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*, pages 100–107, 1993.

[155] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century*, pages 34–44, 2001.

[156] Hiroki Ohashi and Yutaka Watanobe. Convolutional neural network for classification of source codes. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 194–200. IEEE, 2019.

[157] Jukka-Pekka Onnela, Jari Saramäki, János Kertész, and Kimmo Kaski. Intensity and coherence of motifs in weighted complex networks. *Physical Review E*, 71(6):065103, 2005.

[158] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *International Conference on Software Maintenance (ICSM 2001)*, pages 158–167, Los Alamitos, California, USA, November 2001.

[159] Oyebade K Oyedotun and Djamila Aouada. Why do deep neural networks with skip connections and concatenated hidden representations work? In *International Conference on Neural Information Processing*, pages 380–392. Springer, 2020.

[160] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[161] Kai Pan, Xintao Wu, and Tao Xie. Automatic test generation for mutation testing on database applications. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 111–117, 2013.

[162] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 32–39. IEEE Computer Society, 2018.

[163] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 354–365, 2016.

[164] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 936–946. IEEE Computer Society, 2015.

[165] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[166] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019.

[167] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 537–548, 2018.

[168] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 372–387, 2016.

[169] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[170] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? *arXiv preprint arXiv:2105.04297*, 2021.

[171] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[172] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 226–237, 2008.

[173] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[174] Goran Petrovic and Marko Ivankovic. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 163–171, 2018.

[175] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale. *arXiv preprint arXiv:2102.11378*, 2021.

[176] Thomas Pinder. Adversarial detection through bayesian approximations in deep learning, 2018.

[177] Ádám Pintér and Sándor Szénási. Classification of source code solutions based on the solved programming tasks. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000277–000282. IEEE, 2018.

[178] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[179] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[180] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test generation to expose changes in evolving programs. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 397–406, 2010.

[181] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[182] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox v0.8.0: A python toolbox to benchmark the robustness of machine learning models. *CoRR*, abs/1707.04131, 2017.

[183] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.

[184] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA 1994, Seattle, WA, USA, August 17-19, 1994*, pages 169–184, 1994.

[185] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[186] Yunus Saatchi and Andrew Gordon Wilson. Bayesian gan. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3622–3631. Curran Associates, Inc., 2017.

[187] Raúl Santelices and Mary Jean Harrold. Applying aggressive propagation-based strategies for testing changes. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 11–20, 2011.

[188] Raúl A. Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 218–227, 2008.

[189] Raúl A. Santelices and Mary Jean Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 195–206, 2010.

[190] Jari Saramäki, Mikko Kivelä, Jukka-Pekka Onnela, Kimmo Kaski, and Janos Kertesz. Generalizations of the clustering coefficient to weighted complex networks. *Physical Review E*, 75(2):027105, 2007.

[191] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

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

[193] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.

[194] Fatemeh Sheikholeslami, Swayambhoo Jain, and Georgios B. Giannakis. Minimum uncertainty based detection of adversaries in deep neural networks. *CoRR*, abs/1904.02841, 2019.

[195] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[196] Ben H. Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009.

[197] Ben H. Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, 2009.

[198] Lewis Smith and Yarin Gal. Understanding measures of uncertainty for adversarial example detection, 2018.

[199] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[200] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[201] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[202] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[203] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 303–314, 2018.

[204] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *ICSE '18*, pages 303–314, New York, NY, USA, 2018. ACM.

[205] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.

[206] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[207] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312. IEEE, 2019.

[208] Siddhaling Urolagin, KV Prema, and NV Subba Reddy. Generalization capability of artificial neural network incorporated with pruning method. In *International Conference on Advanced Computing, Networking and Security*, pages 171–178. Springer, 2011.

[209] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[210] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Jrnl. Educ. Behav. Stat.*, 25(2):101–132, 2000.

[211] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[212] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[213] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–27, 2020.

[214] Julian von der Mosel, Alexander Trautsch, and Steffen Herbold. On the validity of pre-trained transformers for natural language processing in the software engineering domain. *arXiv preprint arXiv:2109.04738*, 2021.

[215] Ivan Vulić, Edoardo Maria Ponti, Robert Litschko, Goran Glavaš, and Anna Korhonen. Probing pretrained language models for lexical semantics. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7222–7240, Online, November 2020. Association for Computational Linguistics.

[216] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *ICSE '19 (to appear)*, 2019.

[217] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 968–980, 2020.

[218] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[219] Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. Learning to represent programs with heterogeneous graphs. *arXiv preprint arXiv:2012.04188*, 2020.

[220] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[221] Michael Weiss and Paolo Tonella. Uncertainty-wizard: Fast and user-friendly neural network uncertainty quantification. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 436–441. IEEE, 2021.

[222] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.

[223] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[224] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[225] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[226] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[227] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Deephunter: Hunting deep neural network defects via coverage-guided fuzzing. *arXiv preprint arXiv:1809.01266v3*, 11 2018.

[228] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*, pages 789–799, 2018.

[229] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[230] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 257–266, 2010.

[231] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[232] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 2021.

[233] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab.*, 22(2):67–120, 2012.

[234] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[235] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 45(9):898–918, 2019.

[236] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Trans. Software Eng.*, 45(9):898–918, 2019.

[237] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An empirical study on the scalability of selective mutation testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 277–287, 2014.

[238] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

[239] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 235–245, 2013.

[240] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 331–341, 2012.

[241] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[242] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 141–151, New York, NY, USA, 2018. Association for Computing Machinery.

[243] Mengjie Zhao, Philipp Dufter, Yadollah Yaghoobzadeh, and Hinrich Schütze. Quantifying the contextualization of word representations with semantic class probing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1219–1234, Online, November 2020. Association for Computational Linguistics.

[244] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. Masking as an efficient alternative to finetuning for pretrained language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2226–2241, Online, November 2020. Association for Computational Linguistics.

[245] Mengjie Zhao and Hinrich Schütze. A multilingual BPE embedding space for universal sentiment lexicon induction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3506–3517, Florence, Italy, July 2019. Association for Computational Linguistics.

[246] Alice Zheng. *Evaluating Machine Learning Models A Beginner's Guide to Key Concepts and Pitfalls*. O'Reilly Media, Inc, 2015.

[247] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[248] Andrew Zisserman. Self-supervised learning. `https://project.inria.fr/paiss/files/2018/07/zisserman-self-supervised.pdf`, 2018.