Technical Report UM-CS-2017-004, April 2017

Customized Program Mutation: Inferring Mutant Utility from Program Context

René Just
University of Massachusetts
Amherst, MA, USA
rjust@cs.umass.edu

Bob Kurtz
George Mason University
Fairfax, VA, USA
rkurtz2@gmu.edu

Paul Ammann
George Mason University
Fairfax, VA, USA
pammann@gmu.edu

ABSTRACT

The purpose of selective mutation strategies is to reduce the inherent redundancy of full mutation analysis and hence obtain most of its benefit for a fraction of the cost. Unfortunately, recent research has shown that there is no fixed selective mutation strategy that is effective across a broad range of programs. In other words, for any given mutation operator, the utility (i.e., usefulness) of a mutant produced by that operator varies greatly across programs. Hence, selective mutation, as currently defined, is a dead end despite the fact that existing mutation systems are known to produce highly redundant mutants.

This paper explores a novel path out of this conundrum. Specifically, it hypothesizes that mutant utility, in terms of equivalency, triviality, and dominance, can be predicted by incorporating context information from the program in which the mutant is embedded. This paper first explains the intuition behind this hypothesis with a motivational example and then tests the hypothesis by evaluating the predictive power of a series of program-context models and machine learning classifiers.

The results show that it is important to consider mutants at the mutation operator level, and not just at the mutation operator group level. Further, the results show that context extracted from a program's abstract syntax tree can indeed strongly predict mutant utility. The cross-validation results additionally show that mutant utility can be predicted within and across programs, with an area under the ROC curve above 0.8.

KEYWORDS

Mutation analysis, program mutation, program context, mutant utility, machine learning, equivalent mutants, trivial mutants

1 INTRODUCTION

Consider an engineer attempting to evaluate her test suite with mutation analysis. A full-blown mutation analysis system would produce far more mutants than necessary, or, indeed useful. Until recently, the advice to such the engineer would have been to choose a "do fewer" approach in which only a carefully chosen subset of the mutants were generated. Unfortunately, recent research has identified surprising fundamental weaknesses in the existing selective mutation approaches: all of them are very likely to miss "important" mutants on some programs [42], and none of them is likely to greatly outperform random selection [23], a strategy that simply chooses a fraction of the mutants generated. Intriguingly, this same research has also shown that redundancy in mutants can be precisely characterized with "dominator sets" and that the cardinality of dominator sets with respect to the output of a typical mutation engine is very small. In a related study, Ammann et al. [9] showed that the dominator sets, termed minimal mutant sets in that work, averaged just 1.2% of the non-equivalent mutants. That is, nearly 99% of the non-equivalent mutants in that study were redundant. In other words, there really is a small set of dominator mutants that captures the power of the large number of mutants generated by a classic mutation analysis. The good news is that the goal of selective approaches to mutation analysis, namely finding a small set of mutants that retains the utility of the original set, is sound. The bad news is that we don't know how to choose a useful proxy for a dominator set.[1] This paper is a first step in addressing exactly this problem.

Rationale of Our Approach

Why do existing selective mutation approaches fail? In this paper we conjecture that a root problem, and perhaps *the* root problem, is that existing approaches to selective mutation take no account of program context. For example, existing mutation approaches treat mutating a relational operator in a `for` loop test the same as mutating a relational operator in an `if` statement. But many mutations of relational operators in `for` loop tests are killed by every test case, and hence useless, and others are equivalent, and hence worse than useless. As another example, existing mutation systems treat mutating an arithmetic operator in a context of array indexes exactly the same as mutating an arithmetic operator in other contexts. But we know from experience with mutation tools that the former is less likely to result in a dominator mutant than the latter. As a third example, we know that the mutation operator that appends "++" to a variable is more likely to generate an equivalent mutant if it appears late in a computation, for the simple reason that there is less likely to be a data flow from the variable to the output.

The mutation literature has been addressing the "do fewer" approach to mutation analysis for decades, and yet the problem has proved surprising stubborn: recent research has shown that existing approaches don't significantly outperform random selection. Three recent advances suggest that the time is finally ripe to make significant progress.

First, Ammann et al. [9] and Kurtz et al. [42] developed the notion of dominator mutants. While dominator mutants are impractical for the practicing *engineer* to calculate, they provide the *researcher* with a precise, sound definition of mutation adequacy that is free of the distortions imposed by redundant mutants. Research using dominator mutation has shown that for an individual program, there exists a good set of mutation operators, but, unfortunately,

[1] Finding dominator sets directly is undecidable, and approximating dominator sets well with testing requires a very large number of tests. Hence, in a production development environment, it only make sense to look for a proxy.

that there is no set of mutation operators that works well across many programs. In other words, to be effective, the set of mutation operators must be *customized* to the program under test.

Second, empirical work by Just et al. [33] has shown that a strong coupling exists between mutants and, through the test cases that detect them, “real” faults. More recently, Allamanis et al.. [5] showed that additional mutation operators, which are tailored to a program under test, generate mutants that can further increase this coupling. The increased real-fault coupling, however, comes at a cost of significantly more mutants, most of which are redundant. Hence, if the goal is to generate a small set of mutants that are highly coupled to likely real faults in a program under test, then customized program mutation is the way to go.

Third, machine learning approaches have matured into robust and scalable tools, enabling large-scale studies of complex relationships (e.g., [60]). If we can effectively characterize the features of program context that are relevant to the utility of a specific mutant—and the anecdotal examples mentioned above suggest that we can—then off-the-shelf machine learning techniques can be used to train effective classifiers.

Contributions

The underlying hypothesis of this paper is that the selection of a set of effective mutants must be customized to the program context in which the mutants are generated. We expect this relationship between mutant utility and program context to be quite complex, in general, and hence we do not propose to derive the relationships theoretically. Rather, the goal of this paper is to show that context information is indeed useful for predicting mutant utility and that it is possible to employ machine learning to train an effective classifier.

The specific contributions of this paper are:

- A motivational example that provides insight into why program context is critical for assessing mutant utility.
- A model of mutant utility in terms of equivalency, triviality, and dominance.
- A model of program context based on neighboring nodes in the abstract syntax tree.
- An empirical study that shows that program context is a strong predictor for mutant utility.
- An empirical study that shows the prediction performance of different features of program context and different machine learning classifiers.

2 BACKGROUND ON MUTATION ANALYSIS

Mutation testing [18] is a test criterion that generates a set of program variants, called *mutants*, and then challenges the tester to design tests that detect these mutants. A test that can distinguish a mutant from the original program is said to detect, or *kill*, that mutant. In *strong* mutation testing, killing a mutant means that the mutant and the original program generate different outputs. In *weak* mutation testing, killing a mutant means that the internal program state of the mutant differs from the internal program state of the original program at some point during execution.

A mutant is generated by a *mutation operator*, which is a program transformation rule that generates a program variant of a given program based on the occurrence of a particular syntactic element. One example of a mutation operator is the replacement of an instance of the arithmetic operator “`+`” with “`-`”. Specifically, if a program contains an expression “`a + b`”, for arbitrary expressions “`a`” and “`b`”, this mutation operator creates a mutant where “`a - b`” replaces this expression. The *mutation* is the syntactic change that a mutation operator introduces. This paper considers *first-order mutants*, which means that each mutant contains exactly one mutation; higher-order mutation, in which a mutant is a product of multiple applications of mutation operators, is not the focus of this paper but briefly discussed in the related work.

A mutation operator is applied everywhere it is possible to do so. In the example above, if the arithmetic operator “`+`” occurs multiple times in a program, the mutation operator will create a separate mutant for each occurrence. A *mutation operator group* is a group of related mutation operators. For example, the AOR mutation operator group, which includes the mutation operator above, is the group of all mutation operators that replace an arithmetic operator.

2.1 Equivalent mutants

A mutant may behave exactly as the original program on all inputs. Such a mutant is called an *equivalent mutant* and cannot be killed. As an example of an equivalent mutant, consider the following comparison of two integers, which returns the smaller value of the two integers: `return (a < b) ? a : b`. Replacing the relational operator `<` with `<=` results in an equivalent mutant (`return (a <= b) ? a : b`)—if `a` and `b` are equal, returning either value is correct, and hence both implementations are equivalent.

Given a set of mutants, M, a test set T is *mutation-adequate* with respect to M iff for every non-equivalent mutant m in M, there is some test t in T such that t kills m.

2.2 Trivial mutants

Mutants vary widely in how difficult it is to find a test case that kills the mutant. A *trivial mutant* is one that is killed due to an exception by every test case that covers and executes the mutated code location. As an example, consider a `for` loop with a boundary check for an array index (`for int i=0; i<numbers.length; ++i`). If the index variable `i` is used to access the array `numbers` then a mutation `i<=numbers.length` alway results in an exception, as the last value for `i` is guaranteed to index `numbers` out of bounds. Hence, this mutant is trivial, as any test that reaches the loop will terminate with an exception.

2.3 Dominator Mutants

Mutation operators generate far more mutants than are necessary. This redundancy was formally captured in the notion of *minimal mutation* [9]. Give any set of mutants, M, a *dominator set* of mutants, D, is a minimal subset of M such that any test set that is mutation-adequate for D is also mutation-adequate for M.

Computing a dominator set is an undecidable problem, but it is possible to approximate it with respect to a test set [41]—the more comprehensive the test set, the better the approximation. This approximation isn’t useful for the practicing engineer, who needs to know the set of dominator mutants a-priori to develop a test set. However, from a research and evaluation perspective, a dominator set provides a precise way for identifying redundancy in a set of

Table 1: Example kill matrix. A check mark indicates that a test t_i kills a mutant m_j.

Mutant	Test			
	t_1	t_2	t_3	t_4
m_1: ROR_FOR_F		✓		
m_2: ROR_FOR_T	✓	✓	✓	✓
m_3: ROR_FOR_EQ		✓		✓
m_4: ROR_FOR_NE				
m_5: ROR_FOR_GT		✓		
m_6: ROR_FOR_LE	✓	✓	✓	✓
m_7: ROR_FOR_GE		✓		✓
m_8: ROR_IF_F		✓		
m_9: ROR_IF_T	✓			
m_{10}: ROR_IF_EQ		✓		
m_{11}: ROR_IF_NE	✓			
m_{12}: ROR_IF_GT	✓	✓		
m_{13}: ROR_IF_LE				
m_{14}: ROR_IF_GE	✓	✓		

mutants, and hence the dynamic approximation approach is an important research tool for analyzing mutation testing techniques.

Given a finite set of mutants M and a finite set of tests T, mutant m_i is said to *dynamically subsume* mutant m_j if some test in T kills m_i and every test in T that kills m_i also kills m_j. If two mutants m_i and m_j in M are killed by exactly the same tests in T, we say that m_i and m_j are *indistinguished.*

We capture the subsumption relationship among mutants with the *Dynamic Mutant Subsumption Graph* or DMSG [41]. Each node in a DMSG represents a maximal set of indistinguished mutants and each edge represents the dynamic subsumption relationship between two sets of mutants. More specifically, if m_i dynamically subsumes m_j, then there is an edge from the node containing m_i to the node containing m_j. Further, if m_i dynamically subsumes m_j but the converse is not true, we say that the subsumption is *strict.* If a test kills any arbitrary mutant in the DMSG, it is guaranteed to kill all the subsumed mutants [9], i.e., all connected mutants below it in the graph.

Table 1 shows an example kill matrix that indicates which test kills which mutants. In this example, the set M consists of 14 mutants and the set T consists of 4 tests. Every test that kills the last mutant in the table, ROR_IF_GE, also kills ROR_FOR_T, ROR_FOR_LE, and ROR_IF_GT, hence ROR_IF_GE dynamically subsumes these mutants. In the case of the first two mutants, the dynamic subsumption is strict. However, ROR_IF_GE and ROR_IF_GT are killed by exactly the same tests, so the subsumption is not strict; these mutants are indistinguished.

We use the subsumption relationships to construct the DMSG shown in Figure 1. Mutants ROR_FOR_NE and ROR_IF_LE are not killed by any of the tests in T, so it is shown in its own unconnected node with a dashed border. These mutants are equivalent with respect to T but they may be killable by a test that is not an element of T. The DMSG is based on a finite test set, so it can only make claims about test equivalence.

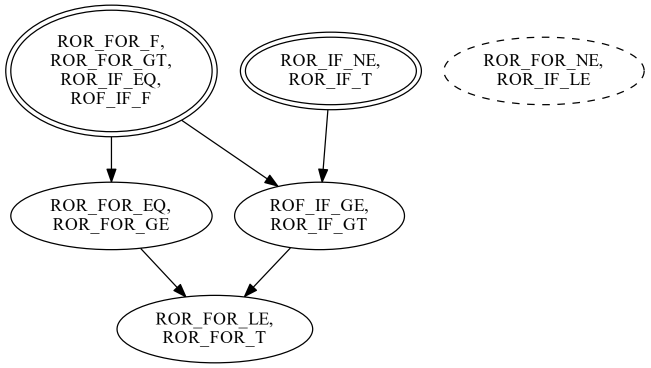


Figure 1: Dynamic mutant subsumption graph (DMSG) for the kill matrix shown in Table 1.

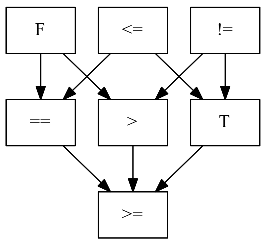


Figure 2: Static subsumption relationship for the relational operator < [37].

Dominator mutants are those not strictly subsumed by any other mutant and are shown in the graph in dominator nodes with double-borders. Figure 1 has two dominator nodes and any combination of one mutant from each dominator node forms a *dominator mutant set.* Hence, Figure 1 has $4 * 2 = 8$ distinct dominator mutant sets; {ROR_FOR_F, ROR_IF_NE} is an example. Because each dominator set contains one mutant from each dominator node, all dominator sets are equally useful and a dominator set can be selected arbitrarily from all possible sets. Consequently, only two of the 14 mutants matter—if a test set kills the mutants in a dominator mutant set, it is guaranteed to kill all non-equivalent mutants, which are redundant.

For completeness, Figure 2 shows the static subsumption relation for the mutants of the relational operator < analyzed in isolation. The top row in Figure 2, namely the three mutations where < is replaced by `F`(or false), <=, and `!=`, shows the dominator mutants. It is important to note that this dominance relation only holds for weak mutation testing[43], and that it assumes that none of the dominators is equivalent. In section 3, we'll revisit the example in Figure 1, where we will note that one of the mutants that is a dominator in isolation, namely `ROR_FOR_NE`, is indeed equivalent when considered in the context of an example program.

3 MUTANT UTILITY

Informally, equivalent and trivial mutants have low utility, and mutants high in the mutant subsumption graph have high utility. This section makes the notion of mutant utility precise along each of three dimensions of equivalency, triviality, and dominance. We defer an exploration of how these three dimensions might be combined into an overall notion of mutant utility to section 6.

Table 2: For each of the seven mutation operators, applied to each of the highlighted program locations in Listing 1, is the generated mutant a *dominator*, *subsumed*, *trivial*, or *equivalent* mutant?

Program location	Op_1	Op_2	Op_3	Op_4	Op_5	Op_6	Op_7
Line 7	**equivalent**	subsumed	trivial	**dominator**	subsumed	**trivial**	**dominator**
Line 8	**dominator**	subsumed	equivalent	**subsumed**	subsumed	**dominator**	**subsumed**

3.1 Equivalent mutants

Ideally, a mutation system would not generate any equivalent mutants, but since mutant equivalence (or program equivalence) is an undecidable problem, this goal can't be achieved in general. Instead, our approach is to rank mutants with respect to an estimate of how likely they are to be be equivalent. Specifically, mutant utility with respect to equivalence is an estimate of the likelihood that the mutant can be killed. Utility ranges from zero (certain to be equivalent) to one (certain to be killable). Program context is useful if it enables us to refine our estimate of equivalence for a given mutant, and the farther the refined estimate is from the base (in either direction), the more useful the context.

Formally, equivalence utility for a mutant m with respect to a mutant operator group G is the likelihood that an arbitrary mutant produced by some Op in G is not equivalent. Similarly, equivalence utility for a mutant m with respect to a mutant operator Op is the likelihood that an arbitrary mutant produced by Op is not equivalent. Finally, equivalence utility for a mutant m with respect to a mutant operator Op and context c is the likelihood that an arbitrary mutant produced by Op in context c is not equivalent.

3.2 Trivial mutants

Since mutation analysis is significantly more effort on the part of the practicing engineer than standard coverage criteria such as branch coverage, mutants that are always killed by branch-adequate test suites aren't of practical value. One of our goals is to identify non-trivial mutants. Again, program context is useful if it enables us to refine our estimate of triviality for a given mutant.

The utility definitions for triviality are an exact parallel of the utility definitions for equivalence.

3.3 Dominator strength

Mutants that are not dominators, but are "close" to being dominators are nonetheless valuable. For the purposes of this paper, we need a metric that captures this observation. To this end, we propose the *dominator strength* metric that satisfies three important properties: it is monotonically increasing along any path in the DMSG, it is fairly evenly distributed between 0 and 1, and it is insensitive to redundant mutants.

We define the dominator strength $s_D(M)$ for any mutant M as the number of nodes in the graph that are subsumed by M, divided by the number of nodes in the graph subsumed by M plus the number of nodes in the graph that subsume M:

$$s_D(M) = \frac{\#\mathit{nodes}\ M\ \mathit{subsumes}}{\#\mathit{nodes}\ M\ \mathit{subsumes} + \#\mathit{nodes\ that\ subsume}\ M}$$

$s_D = 1$ identifies a dominator mutant, and $s_D = 0$ identifies a mutant that does not strictly subsume any other mutants. As an example, in Figure 1, $s_D(\mathit{ROR\_IF\_GE}) = 1/(1+2) = 0.33$.

The utility definitions for dominance are an exact parallel of the utility definitions for equivalence.

4 PROGRAM CONTEXT

This section first informally describes the notion of program context using a motivational example (Section 4.1) and then details our proposed approach to modeling program context (Section 4.2).

4.1 Motivational example

Consider the following program listing, in particular the highlighted expressions that involve a relational operator in lines 7 and 8:

```
 1 /*
 2  * Compute the minimum value of a non-null,
 3  * non-empty array of integers.
 4  */
 5 public int getMin(int[] numbers) {
 6   int min = numbers[0];
 7   for (int i=1; i < numbers.length; ++i) {
 8     if (numbers[i] < min) {
 9       min = numbers[i];
10     }
11   }
12   return min;
13 }
```

Listing 1: A relational operator in two different program contexts.

The following seven mutation operators are applicable to each of the highlighted program locations:

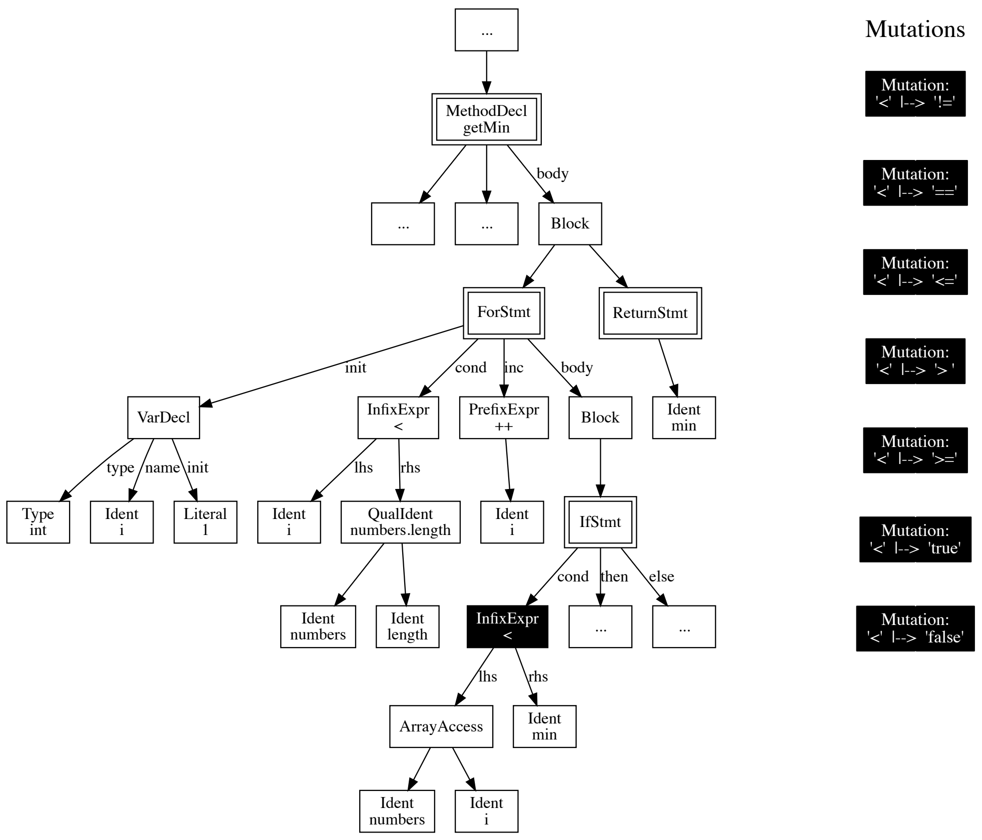
Op_1: `lhs < rhs` $\longmapsto$ `lhs != rhs`
Op_2: `lhs < rhs` $\longmapsto$ `lhs == rhs`
Op_3: `lhs < rhs` $\longmapsto$ `lhs <= rhs`
Op_4: `lhs < rhs` $\longmapsto$ `lhs > rhs`
Op_5: `lhs < rhs` $\longmapsto$ `lhs >= rhs`
Op_6: `lhs < rhs` $\longmapsto$ `true`
Op_7: `lhs < rhs` $\longmapsto$ `false`

Note that `lhs` and `rhs` are placeholders for any expression that appears as the left hand side or right hand side of a relational operator. Note also that the $\longmapsto$ symbol denotes the program transformation that a mutation operator (Op_i) applies and that `true` and `false` indicate that the mutated expression always evaluates to true or false, respectively.

The seven mutation operators are universally applicable to any relational operator in the program. However, the usefulness of the mutants they generate depends on the program location, as Table 2 shows.

Table 2 corroborates that any approach that universally selects and applies a subset of mutation operators—even within a single

Figure 3: Partial abstract syntax tree for the original version of the `getMin` method (Listing 1) and seven possible mutations for the highlighted node.



program—is doomed to failure. For example, the mutation operator Op_1 generates an equivalent mutant in line 7 but a dominator mutant in line 8. This means that the inclusion of Op_1 is crucial but at the same time that this operator should never be applied in a context similar to the one in line 7. This example motivates our work of capturing the notion of program context more precisely with the ultimate goal of learning what mutation operator is most likely to generate an equivalent mutant or a dominator mutant in what program context. Figure 1 (dynamic subsumption) and Figure 2 (static subsumption) illustrate exactly this same point: the subsumption relations change when the mutations are considered in context.

An immediate follow-up question to this motivational example is how frequently does it occur in practice. To this end, we conducted an exploratory study, using the Lang-1 subject from the Defects4J corpus. The test suite of Lang-1 achieves 98.2% statement coverage and kills 216 out of 508 ROR mutants that are generated by applying Op_1. Given the strength of the test suite, a large fraction of the remaining 292 mutants are very likely to be equivalent. Hence, absent context information, the estimate that an Op_1 mutant is equivalent is 292/508 or 57%.

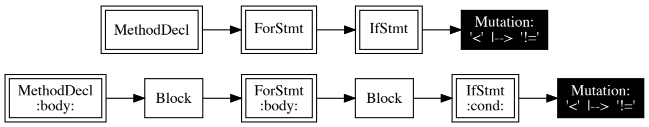
Incorporating context information and considering only mutations generated by Op_1 in the condition of a `FOR` loop, reduces the number of mutants from 508 to 165. Of these, 11 mutants are killed and 154 mutants are equivalent. Therefore, considering just the enclosing statement changes the estimate that an Op_1 mutation in a `FOR` loop context is equivalent from 57% to 93%. This enclosing statement context, all by itself, already provides a very strong signal for the equivalence of mutants generated by Op_1. Subsequent manual analysis revealed that for each of the 11 killed Op_1 mutants, there is additional context information that could be exploited to predict whether or not that mutant is equivalent. For example, some of these `FOR` loops start at a variable index instead of 0 or 1.

4.2 Modeling program context

Rather than predefining a very small set of exclusion patterns that provably generate equivalent or trivial mutants, we generalize this notion of patterns to program context. We model program context using the program's abstract syntax tree (AST). In contrast to the purely syntactic level, the AST provides a higher level of abstraction and semantic information. Figure 3 shows the partial AST for the `getMin` method from Listing 1. The highlighted node indicates the node targeted by the ROR mutation operators.

Our ultimate goal is to apply machine learning on labeled ASTs to train a classifier that can predict mutant utility, given a mutation operator and the program's AST. As a first step in this paper, we

Figure 4: Two AST node sequences for the mutation operator Op_1 applied to the highlighted node in Figure 3. The two sequences represent the same path in the AST at different levels of abstraction: the first sequence includes only nodes that represent a top-level statement, whereas the second sequence includes all nodes plus edge labels.



validate our assumption that program context indeed affects mutant utility, and we explore different dimension of program context to guide future research on developing more complex models.

Specifically, our implementation traverses the AST and computes the sequence of AST nodes, from the target node to the root node. Note that such a traversal can generate AST node sequences at different levels of abstraction. Figure 4 gives an example for the highest level of abstraction, which only considers top-level statements rather than individual expressions.

Operating on the abstract syntax tree not only allows us to abstract over potentially irrelevant details but also provides semantic information such as scope and data types of expressions. This information can enable a classifier to learn, among other things, whether a mutation operator should discriminate between 1) integer and floating-point expressions, 2) expressions that involve local vs. global variables, and 3) different identifier names in identical program contexts. In summary, modeling program context using AST nodes provides the following advantages:

- Different levels of abstraction (top-level statements vs. expressions).
- Possible abstraction over identifier names.
- Availability of type information.
- Availability of information about visibility and scope.

In addition to the structural program context, modeled as AST node sequences, the data type context of the target AST node can affect mutant utility. For example, consider again a mutation of the expression `lhs < rhs`. Knowing whether the comparison is floating point vs. integer leads to different expectations about mutant behavior: mutating a floating point comparison is highly unlikely to lead to an `ArrayIndexOutOfBoundsException`—neither `lhs` nor `rhs` is likely to be used as an array index. On the other hand, the same mutation on integers is likely to lead to this exception if `lhs` or `rhs` is used as an array index. Moreover, differentiating < from <= is straight-forward for integers but quite complex for floating point numbers. Hence, such a mutation is likely to be equivalent for floating point numbers but not for integers.

Similarly, the scope and visibility of a variable matters with respect to mutant equivalency. Consider mutating the value of a global variable vs. mutating the value of a local variable. The persistence of global variables means that mutations to them are far less likely to be equivalent compared to mutations to local variables.

5 EVALUATION

The purpose of this empirical evaluation is threefold. First, it aims to study whether program context affects mutant utility. Second, it aims to study whether the expected mutant utility differs between all mutants of a particular mutation operator groups (e.g., ROR) and all mutants of a particular mutation operator (e.g., Op_1 in the motivational example). Third, it aims to study what dimension of program context is a strong predictor for expected mutant utility.

5.1 Subjects

We used the Defects4J benchmark [32] (v1.0.1) for the empirical evaluation, which provides a set of 357 subjects, each accompanied by a thorough, developer-written test suite. For each subject, Defects4J provides a buggy and a fixed program version—the difference between the two versions is a set of classes that a developer fixed. We refer to each class in that set as a *modified class*.

We selected 163 subjects from the Defects4J corpus where the test suite achieved at least 95% statement coverage on all modified classes. Given that the number of trivial mutants and the dominator strength are approximations that rely on an exhaustive test suite, we selected these subjects because of their thorough test suites.

We employed the Major mutation framework [29] to generate mutants for the modified classes for these subjects, perform a mutation analysis using the subjects' test suites, and compute the kill matrix (i.e., execute all tests against all mutants) for each subject. Major could not generate the kill matrix for 56 out of 163 subjects due to a computational timeout of 48 CPU hours.[2] An automated step then filtered the remaining 107 subjects to remove subjects containing duplicate classes (and thus removed the possibility of duplicate mutants) using the following procedure:

(1) When two or more subjects contained the same classes, we retained only the subject with the highest statement coverage.
(2) When two or more subjects contained the same classes and had the same statement coverage, we retained only the subject with the larger number of test cases.
(3) When two or more subjects contained the same classes, had the same statement coverage, and had the same number of test cases, we retained only the subject with the newest version of the subject source code.

This filtering process resulted in 80[3] subjects for which Major generated 79,002 mutants. Of these, 878 mutants were not covered by any test and we discarded them from the evaluation. The remaining 78,124 mutants were covered by an average of 63.7 tests per mutant.

Table 3 provides details about the generated mutants across these 80 subjects. In contrast to the other mutation operator groups, LOR,

[2]The expected runtime to compute the kill matrix for some of the Defects4J subjects is beyond 100 CPU days[59].
[3]Chart-24, Closure-5, Closure-8, Closure-9, Closure-12, Closure-13, Closure-14, Closure-15, Closure-28, Closure-36, Closure-45, Closure-46, Closure-49, Closure-55, Closure-58, Closure-67, Closure-72, Closure-88, Closure-89, Closure-91, Closure-92, Closure-98, Closure-102, Closure-108, Closure-111, Closure-121, Closure-124, Closure-130, Closure-132, Lang-1, Lang-2, Lang-4, Lang-11, Lang-21, Lang-22, Lang-25, Lang-28, Lang-31, Lang-33, Lang-37, Lang-39, Lang-40, Lang-44, Lang-45, Lang-49, Lang-51, Lang-53, Lang-54, Lang-55, Lang-58, Lang-59, Lang-62, Math-4, Math-10, Math-15, Math-22, Math-25, Math-26, Math-28, Math-35, Math-39, Math-40, Math-52, Math-64, Math-69, Math-79, Math-84, Math-86, Math-89, Math-95, Math-96, Math-100, Math-105, Time-3, Time-4, Time-5, Time-7, Time-10, Time-21, Time-22

Table 3: Summary of mutation operator groups and mutants. Dominator strength gives the average dominator strength for all killed mutants, and covering/killing tests give the average number of tests that cover/kill each mutant. The highlighted rows indicate discarded mutation operator groups whose number of mutants is too small for this study.

Group	Total mutants	Killed mutants		Dominator mutants		Trivial mutants		Dominator stength	Covering tests	Killing tests
AOR	13456	12217	(90.8%)	4016	(29.8%)	1671	(12.4%)	0.679	43.0	19.9
COR	12161	9960	(81.9%)	4284	(35.2%)	1444	(11.9%)	0.704	71.5	24.3
EVR	3893	3731	(95.8%)	1209	(31.1%)	1081	(27.8%)	0.588	70.2	40.4
LOR	344	294	(85.5%)	115	(33.4%)	6	(1.7%)	0.696	311.2	19.3
LVR	13448	10907	(81.1%)	5087	(37.8%)	2793	(20.8%)	0.707	105.1	15.1
ORU	537	486	(90.5%)	153	(28.5%)	10	(1.9%)	0.659	492.8	22.4
ROR	27510	22768	(82.8%)	10429	(37.9%)	3925	(14.3%)	0.683	36.0	15.0
SOR	186	167	(89.8%)	69	(37.1%)	1	(0.5%)	0.792	548.5	7.5
STD	6589	5652	(85.8%)	2597	(39.4%)	1084	(16.5%)	0.751	56.9	20.2
Overall	78124	66182	(84.7%)	27959	(35.8%)	12015	(15.4%)	0.690	63.7	22.5

ORU, and SOR yielded very few mutants. To avoid spurious results due to an insufficient sample size, we discarded the 1,067 mutants from these three mutation operator groups, leaving 77,057 remaining for analysis. In addition to the breakdown of killed mutants, dominator mutants, and trivial mutants, Table 3 gives, for each mutation operator group, the average dominator strength and the average number of tests that cover and kill a mutant of that group.

5.2 Program context

Given the preliminary evidence that mutant utility depends on program context, our goal is to identify what dimensions of program context are most likely to predict whether a mutant is trivial, a dominator, or equivalent. This study investigates the following three dimensions of program context:

(1) *Parent statement context (detailed)*—structural context: The type of the nearest ancestor AST node that corresponds to a top-level statement, annotated (where applicable) with the relationship between the mutated node and that parent statement node. Figure 4 gives an example for such an annotated relationship, where the mutated node is the child node of an if statement condition (`IfStmt:cond:`).

(2) *Child node(s) context*—structural context:
- *Has literal*: indicates whether any immediate child node of the mutated AST node is a literal.
- *Has variable*: indicates whether any immediate child node of the mutated AST node is a variable.
- *Has operator*: indicates whether any immediate child node of the mutated AST node is an operator.

(3) *Node data type (detailed)*—data type context: The detailed data type of the mutated AST node. This gives, for example, for a relational operator the types of the operands and the return type `(int,int)boolean`.

5.3 Mutation operator groups

This study considers the following mutation operator groups:

- AOR: Arithmetic operator replacement
- COR: Conditional operator replacement
- EVR: Expression value replacement
- LVR: Literal value replacement
- ROR: Relational operator replacement
- STD: Statement deletion

In addition to the mutation operator groups, this study considers the individual mutation operators (e.g., Op_1–Op_7 in the motivational example) because it's very likely that an interaction effect between different dimensions of program context and the individual mutation operators exists.

5.4 Mutant utility

This study explores the following three dimensions of mutant utility (recall Section 3):

(1) *Equivalency*: Measured as the ratio of non-killed mutants.
(2) *Triviality*: Measured as the ratio of trivial mutants.
(3) *Dominance*: Measured as the average dominator strength.

5.5 Results

Each of figures 5, 6, and 7 displays the expected mutant utility at three levels of granularity. The first level is the mutation operator group level. For example, in each figure, the first row in the first column shows the expected mutant utility when considering all AOR mutants. The second level is the mutation operator level. For example, in each figure, the second row in the first column shows the expected mutant utility when considering all mutants for each of the AOR mutation operators. The third level puts each mutation operator into context—that is, it associates each mutation operator with parent statement context. For example, in each figure, the third row in the first column shows the expected mutant utility when considering all mutants for each of the possible mutation operator/parent statement context combinations.

Since extremely low and extremely high values are of the most interest, the x-axis for the graph in each column shows data sorted in non-descending order. Informally, "flat" graphs are "bad", in that they don't convey much predictive power as to mutant utility, but graphs with "low" and/or "high" values are "good", in that these regions identify mutants that are either highly desirable or should be avoided.

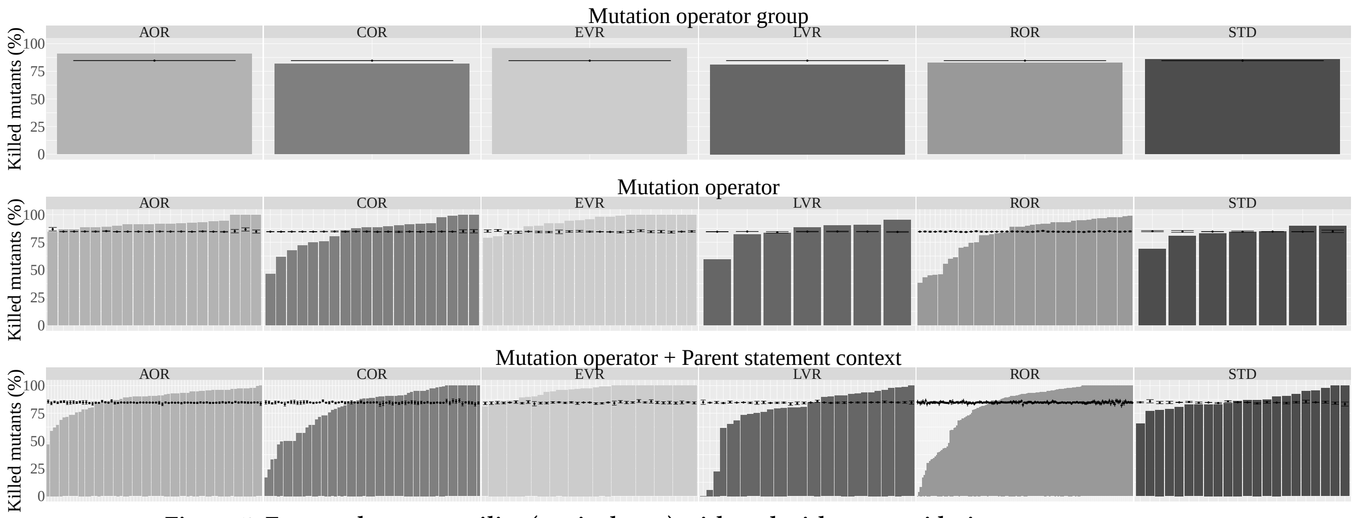


Figure 5: Expected mutant utility (equivalency) with and without considering program context.

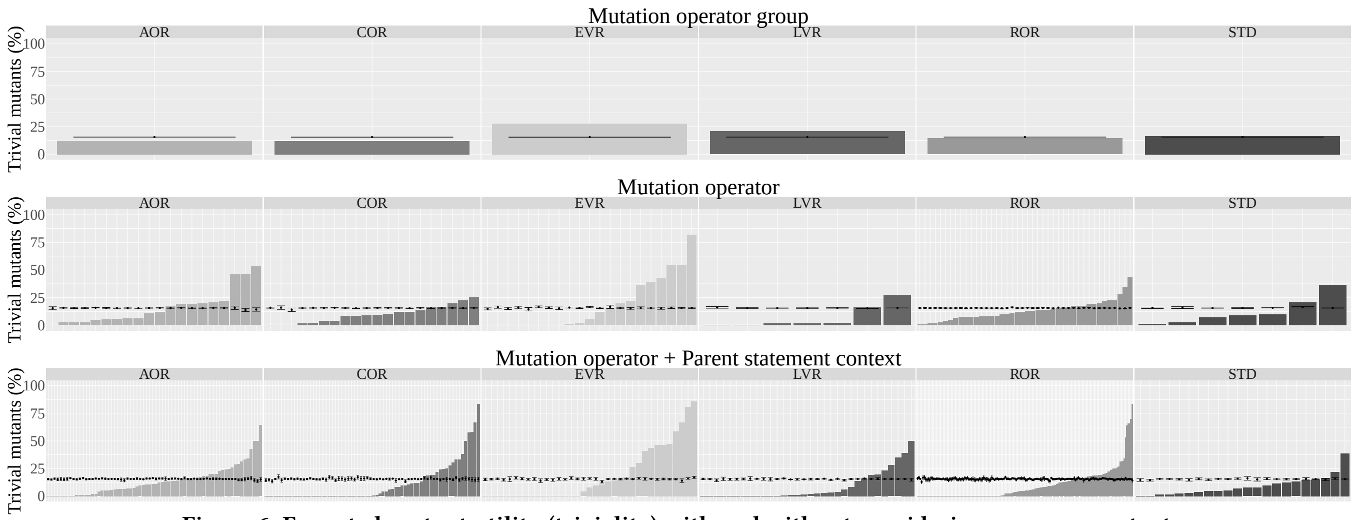


Figure 6: Expected mutant utility (triviality) with and without considering program context.

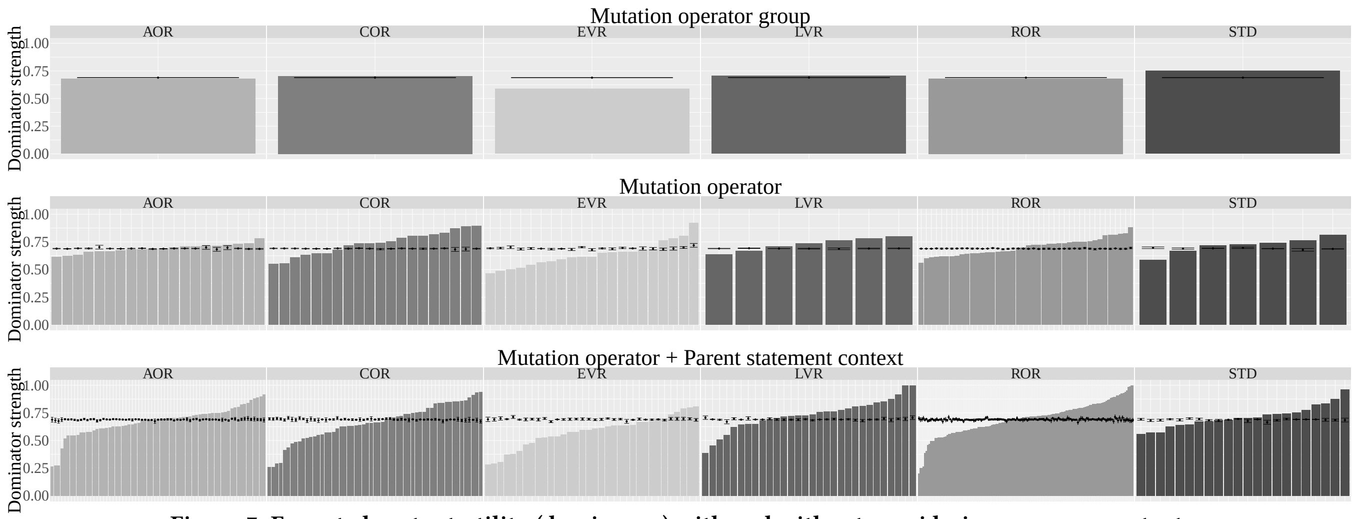


Figure 7: Expected mutant utility (dominance) with and without considering program context.

5.5.1 Equivalency. Figure 5 shows the impact of the program context on the expected mutant utility considering only the equivalency dimension. Recall that, due to the thoroughness of the employed test suites, we are using killed mutants as a proxy for non-equivalent mutants. The top row of the figure shows that there isn't much difference between the six mutation operator groups that we studied. In particular, trying to predict likely equivalent mutants from mutation operator groups is hopeless. The second row in the figure shows that there is some variance between mutation operators. For example, while the expected mutant utility shows almost no variation for AOR and STD mutation operators, some COR and ROR mutation operators are much more likely to generate equivalent mutants than others. Adding parent context information, as is shown in the last row of the figure dramatically increases this variance. In particular, parent context information allows one to identify some COR, EVR and ROR mutants, namely the ones at the left hand edge of the graph, that are highly likely to be equivalent. Conversely, the right edge of each graph shows mutants in contexts where they are much less likely than other mutants in their mutation operator group to be equivalent.

Figure 5 shows a strong signal that certain mutation operators are very likely to be equivalent in certain contexts, and also that others are very unlikely to be equivalent in other contexts. In terms of our motivational example, the parent context of a FOR statement combined with the Op_1 ROR mutation operator, which replaces "less than" with "not equals", appears near the left hand side of the third row, fifth column graph, precisely because this combination is highly likely to generate an equivalent mutant. Hence, a context-aware mutation system should not generate these mutants, as most of them are worse than useless.

5.5.2 Triviality. Figure 6 shows the impact of the program context on the expected mutant utility considering only the triviality dimension. The top row of figure 6 shows that for all mutation operator groups, roughly 20% of the generated mutants are trivial. The top row, however, does not give any guidance as to which of these mutation operator groups are more likely to generate trivial mutants than others. Breaking the analysis down by mutation operator significantly improves matters. For example, the third graph in the second row shows that some EVR operators are highly unlikely to generate a trivial mutant, while other EVR operators are highly likely to do so. Adding parent statement context information, as shown in the third row improves the predictive power even more.

5.5.3 Dominance. Figure 7 shows the impact of the program context on the expected mutant utility considering only the dominance dimension. Again, knowing just the mutation operator group is not enough to make a meaningful prediction of dominator strength. Adding in the specific mutation operator adds some predictive power, but not nearly as much as also including the parent statement context, as shown in the last row.

5.5.4 Random selection. To ensure that our results were not simply an artifact of grouping smaller numbers of mutants together, we performed a control experiment using random mutant selection. Our approach was to duplicate our new process as closely as was practical. For each data point, which represents the expected mutant utility for a set of grouped mutants, we substituted an identical number of mutants randomly chosen from the entire set of mutants. We repeated this randomized process 100 times. The error bars in Figures 5–7 show the results in terms of mean expected mutant utility and 90% confidence interval. The random selection results show very little differentiation in score compared to our new process. We conclude that the differentiation of results is in fact due to our selection process and not a sampling artifact.

5.5.5 Node data type and child node(s) context. We performed the same analyses described in the previous subsections, considering node data type context and child node(s) context. More concretely, we studied each dimension of context in isolation and in combination with the other two. The results showed that the node data type and child node(s) context in isolation only marginally improve over considering individual mutation operators. Furthermore, adding both, node data type and child node(s) context, to the parent statement context improves over parent statement context alone, but parent statement context provides the strongest signal.

6 CONTEXT-BASED MUTANT SELECTION

Section 5.5 shows our results for the three dimensions of mutant utility (equivalency, triviality, dominance) in isolation. These results suggest that program context is a very strong predictor of mutant utility, but individually they do not provide enough information to select mutants that will achieve high dominator scores with few equivalent and trivial mutants. To do that effectively, we must consider the groups of mutants in multiple dimensions.

The left side of Figure 8 shows groups of mutants based on unique combinations of mutation operator and parent statement context. The x-axis indicates dominator strength, while the y-axis shows the percentage of mutants killed. The size of the data points shows the relative number of mutants within each group, with a minimum of 10 mutants.

Groups of mutants with kill ratios below some threshold[4] have too many equivalent mutants, and thus cause too much work for the tester[42]. At the same time, mutant groups with lower mean dominator strength tend to contain redundant mutants, which add no value. Our goal is to select those combinations of mutation operators and program context that generate mutants in the upper-right corner of the graph, with few equivalent mutants and high dominator strength. We conjecture that these mutants will elicit test sets that achieve high dominator scores while minimizing the work required from the tester.

The right side of Figure 8 shows groups of mutants selected at random. These groups contain the same number of mutants as a corresponding group in left graph, but selected randomly[5] rather than by mutation operator and context. This graph shows considerably less differentiation in either dominator strength or kill ratio.

Figure 9 shows the distribution of the expected mutant utility for all unique mutation operator and program context combinations.

[4]While a threshold of 75% is shown here, it is intended to be notional and subject to further research.
[5]For clarity, this graph shows a single randomized example rather than the mean of multiple runs. While the results of individual randomized runs obviously differ, this graph is a representative result.

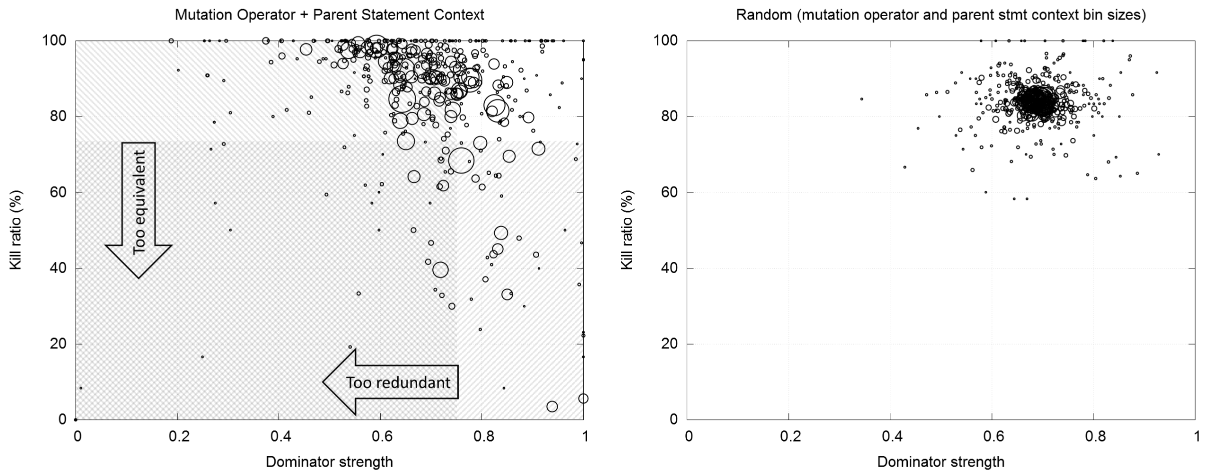


Figure 8: Distribution of expected mutant utility for context-based selection and random selection. Each circle represents a group of mutants and the size of each circle indicates the number of mutants in that group.

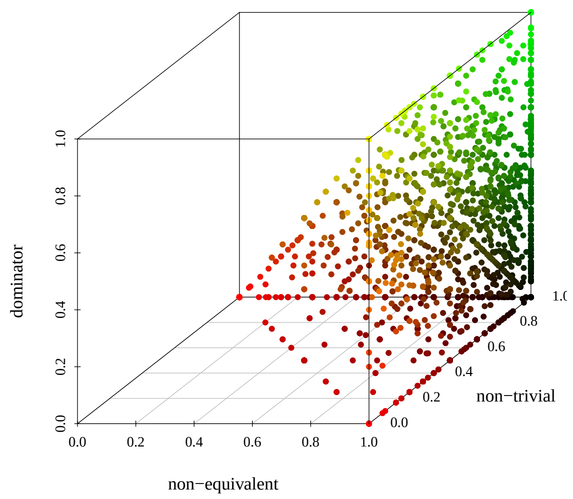


Figure 9: Distribution of the expected mutant utility. Each data point represents the expected utility of a mutant generated by a particular mutation operator in a particular program context.

Each data point represents the expected utility of a mutant generated by a particular mutation operator in a particular program context. The mutation operator/context combinations close to $(1, 1, 1)$ generate highly desirable mutants—very high expected dominance with very low expected equivalency and triviality. In contrast, operator/context combinations close to $(0, 1, 0)$ and $(1, 0, 0)$ generate highly undesirable mutants—very low expected dominance with very high equivalency or triviality.

7 PREDICTING MUTANT UTILITY USING MACHINE LEARNING CLASSIFIERS

To further investigate the usefulness of program context to predict mutant utility, we evaluated the effectiveness of different context information and machine learning algorithms, using the Scikit-Learn[60] package.

Machine learning allows a computer program to refine its decision-making abilities based on experience [73]. In *supervised learning*, training data or previous experience is used to guide the analysis of new data [63]. This experience is provided in the form of a *training set*, a collection of inputs and their associated output(s). The machine learning algorithm uses the training set to improve its ability to predict the appropriate output for any new set of inputs.

Supervised learning algorithms require each element of the training set to include two components:

(1) A *feature vector*, an n-dimensional numeric description of the input features of the training element.
(2) A *label*, the known output or result associated with the input vector.

Once trained, the machine learning algorithm can accept new, previously unseen feature vectors and determine the most likely label associated with these features.

7.1 Program context features

In order to train a machine learning algorithm on a set of mutants, we must have some understanding of the features that describe the program context surrounding the corresponding mutations. Further, we must be able to encode that information into a feature vector for input into the machine learning algorithm. Section 4.2 gives an overview of our approach to modeling program context using information from a program's abstract syntax tree. For our machine learning experiments, we explored the following program context features (other useful features may be identified as research progresses):

- *Mutation operator group*: one of AOR, COR, EVR, LOR, LVR, ORU, ROR, SOR, or STD[6].
- *Mutation operator*: the specific program transformation that generates a mutant (e.g., `lhs < rhs` $\longmapsto$ `lhs != rhs`).
- *Node data type (basic)*: the summary data type of the mutated node in the AST (e.g., `int`, `boolean`, `class`).
- *Node data type (detailed)*: the detailed data type of the mutated node in the AST—the data types of the operands and result (e.g., `(int,int)int`) for mutated operators or methods; identical to *node data type (basic)* for other mutations.
- *Node type (basic)*: the type of the mutated AST node (e.g., 'binary operator', 'int literal', 'method call').
- *AST context (basic)*: the sequence of AST node types from the mutated node (inclusive) to the root node of the AST. Figures 3 and 4 give an example.
- *AST context (detailed)*: similar to *AST context (basic)*, but nodes with multiple children are annotated with the edge label that indicates the relationship (e.g., 'for' becomes 'for:init', 'for:cond', 'for:inc', or 'for:body'). Figures 3 and 4 give an example.
- *AST statement context (basic)*: the sequence of AST node types from the mutated node (inclusive) to the root node of the AST, abstracted to include only those nodes that represent a top-level statement.
- *AST statement context (detailed)*: similar to *AST statement context (basic)*, but nodes with multiple children are annotated with the edge label that indicates the relationship.
- *Parent context (basic)*: the AST node type of the immediate parent node of the mutated AST node.
- *Parent context (detailed)*: the AST node type of the immediate parent node of the mutated AST node plus the edge label that describes the relationship between the two nodes (see *AST context detailed*).
- *Parent statement context (basic)*: similar to *Parent context (basic)*, but abstracted to include only those nodes that represent a top-level statement.
- *Parent statement context (detailed)*: similar to *Parent context (detailed)*, but abstracted to include only those nodes that represent a top-level statement.
- *Child node type*:
  - *Has literal*: indicates whether the mutated AST node has an immediate child node that is a literal.
  - *Has variable*: indicates whether the mutated AST node has an immediate child node that is a variable.
  - *Has operator*: indicates whether the mutated AST node has an immediate child node that is an operator.

While some context features like 'has literal' are simple boolean values, others such as 'node data type' are made up of a set of distinct strings. We used the Scikit-learn *CountVectorizer* feature extraction tool to build a vocabulary of all the strings, then binarized the vocabulary into a sequence of binary values that represent each entry in the vocabulary. For the 'AST context' and 'AST statement context' components, we also generated sets of n-grams to represent subsets of the AST node sequences. After experimentation, we decided on n-grams of lengths one to four.[7] In other words, *CountVectorizer* built a vocabulary consisting of all individual nodes in the AST context, plus all sequences of two consecutive AST nodes, plus all sequences of three consecutive nodes, plus all sequences of four consecutive nodes.

7.2 Evaluating program contexts

In our first machine learning experiment, we selected the decision tree (DT) machine learning classifier for its simplicity. To evaluate the effectiveness of different sets of program context, we trained the decision tree on all mutants in our dataset that were covered by at least one test, then tested all mutants against the trained classifier. The goal of this experiment is not to evaluate the performance of the classifier in a realistic setting but rather to determine the relative utility of different features of program context.

Consistent with our previous experiment, we examined the classifiers and program context features using three dimensions of mutant utility (equivalency, triviality, and dominance). For the dominance dimension, our experiments use the notion of strong mutants (dominator strength $s_D >= 0.95$). .

We present the results in form of receiver operating characteristic (ROC) curves, which show the number of true positive results and false positive results across a range of cut-off points. Recall that a classifier assigns a probability to a feature vector that indicates how likely this feature vector belongs to a particular class. In case of binary classification, the cut-off point is the threshold value used to assign a binary label to a feature vector (e.g., true if the probability is above the cut-off point, false otherwise).

Consider the dashed gray "Random" line in Figure 10. This graph shows the idealized result of randomly selecting killable mutants. As mutants are randomly selected, we tend on average to sample equal proportions of both killed and unkilled mutants, so the curve increases linearly. The line terminates at (1,1), where all killed and unkilled mutants have been selected.

One metric for the effectiveness of a binary classifier is the area under the ROC curve (AUC)[13]. For random selection, $AUC = 0.5$. Classifiers that are more effective than random selection will have curves offset to the upper-left above the random curve, where true positive results are found preferentially compared to false positive results. Thus, the AUC will be higher for more effective classifiers. The ideal curve would begin at (0,0) and rise vertically to (0,1), where all true positive results and no false positive results have been identified, resulting in $AUC = 1.0$.

Figrue 10 shows the results for the decision tree classifier predicting mutant equivalence when trained on different program context features[8]. Using only the mutation operator group (MutOp Group), the classifier performance was only slightly better than random. Using the specific mutation operators (MutOp) significantly improved the ability to predict equivalent mutants. Adding the parent statement context (MutOp+Parent) and adding the data type of the

[6]Our machine learning experiments included all mutation operator groups.

[7]Longer n-gram sequences substantially increased processing time with no appreciable increase in predictive power.

[8]While we tested many program context features, we show only a representative set.

Table 4: Area under the ROC curve for various program context features, considering the decision tree classifier and three dimensions of mutant utility.

Program context	Area under the ROC curve		
	Equivalency	Triviality	Dominance
Mutation operator & node data type & child node types & AST context (n-grams)	0.911	0.922	0.835
Mutation operator & node data type & child node types & AST context (complete)	0.907	0.919	0.828
Mutation operator & parent statement & node data type & child node types	0.833	0.833	0.683
Mutation operator & parent statement	0.792	0.774	0.633
Mutation operator	0.747	0.730	0.592
Mutation operator group	0.581	0.583	0.524
Random	0.500	0.500	0.500

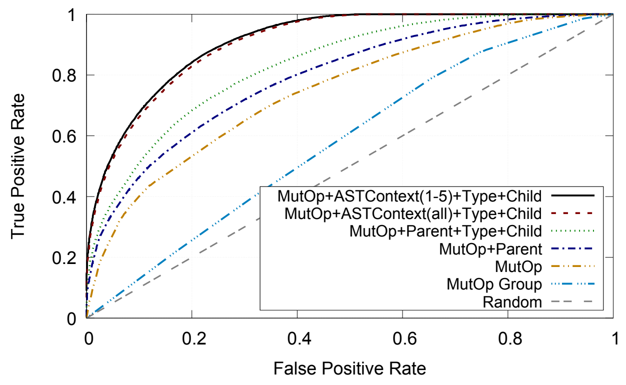


Figure 10: ROC curves for various program context features using the decision tree classifier to predict equivalent mutants.

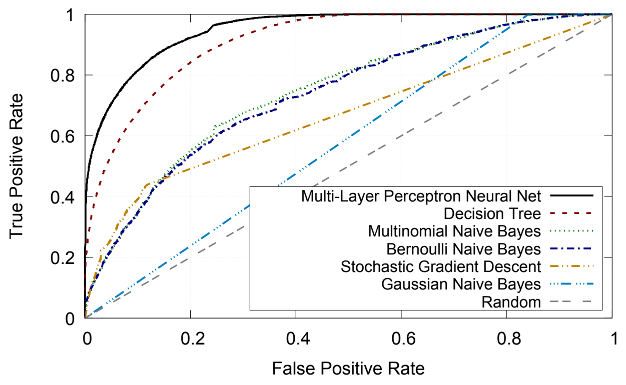


Figure 11: ROC curves for various classifiers predicting equivalent mutants.

mutated node and its child node types (MutOp+Parent+Type+Child) further improved equivalent mutant prediction. The decision tree classifier achieves the best prediction results when trained on the specific mutation operator, node data type, child node types, and AST context n-grams (MutOp+ASTContext(1-5)+Type+Child). In contrast to using n-grams of the AST context, using the entire AST context (MutOp+ASTContext(all)+Type+Child) resulted in slightly lower predictive power.

Table 4 shows the area under the ROC curve (AUC) measures, In general, prediction of equivalent and trivial mutants was approximately equal, while the prediction of strong mutants was somewhat less effective. In all cases, using the program context features mutation operator, AST context n-grams, node data type, and child node types (MutOp+ASTContext(1-5)+Type+Child) showed the best prediction results for the desired mutant properties.

7.3 Evaluating machine learning classifiers

In our second machine learning experiment, we used several different machine learning classifiers from Scikit-Learn to determine the most effective classifier for our mutant dataset. In all cases we continued to use all mutants that were covered by at least one test. All classifiers used the best program context features determined in the previous section (MutOp+ASTContext(1-5)+Type+Child) and were trained and tested against the entire dataset.

Figure 11 shows the ROC curves for six different classifiers, predicting equivalent mutants, and Table 5 shows the AUC measures for all three dimensions of mutant utility. The Gaussian naive Bayes (GNB) classifier performed only marginally better than random selection, while Bernoulli naive Bayes (BNB), multinomial naive Bayes (MNB), and stochastic gradient descent (SGD) showed better predictive results—although well below decision trees. The multi-layer perceptron neural net (MLP) classifier showed the greatest predictive ability. Again, the results are consistent for all three dimensions of mutation utility.

7.4 Cross validation

While training and testing on the same dataset may be useful for comparing program context features and machine learning classifiers, it is not a useful approach for assessing the generalizability and predictive power of a classifier on previously unseen data. Therefore, in our third machine learning experiment, we used several different cross-validation techniques to assess the practicality of using machine learning to predict mutant utility in a more realistic setting. All cross-validation experiments in this section use the most predictive program context features (MutOp+ASTContext(1-5)+Type+Child) and the best-performing classifier (MLP).

Table 5: Area under ROC curve for various classifiers, considering three dimensions of mutant utility.

Classifier	**Area under the ROC curve**		
	Equivalency	Triviality	Dominance
Multi-layer perceptron	0.950	0.945	0.898
Decision tree	0.911	0.922	0.835
Multinomial naive Bayes	0.745	0.812	0.704
Bernoulli naive Bayes	0.738	0.809	0.696
Stochastic gradient descent	0.667	0.659	0.681
Gaussian naive Bayes	0.579	0.621	0.571
Random	0.500	0.500	0.500

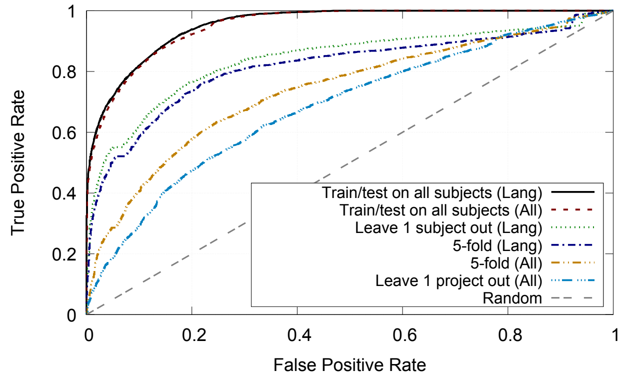


Figure 12: ROC curves for MLP classifier in various cross-validation settings, predicting equivalent mutants.

We used *k-fold* and *leave-one-out* cross validation—each within a single project (Lang) and between all projects (All). In particular, we used 5-fold cross validation, which randomly groups mutants into five folds of equal size. For each fold, we trained the MLP classifier on four folds and then tested it against the fifth. Furthermore, we employed a leave-one-project-out cross validation, considering all projects, and a leave-one-subject-out cross validation, considering only the project Lang, which contributes 45% of all mutants.

Figure 12 shows the ROC curves for the four different cross validations plus three baselines (training/testing on all data for all projects and Lang, and random selection). Additionally, Table 6 gives the AUC measures for all three dimensions of mutant utility. The 5-fold cross validation confirms that the MLP classsifier, using the chosen program context features, is a good predictor of equivalent mutants within a single project ($AUC = 0.815$) and between multiple projects ($AUC = 0.737$).

Consider an engineer who is starting to test a new program. In this scenario, the engineer would train a classifier on other programs and use that classifier to predict mutant utility for the new program under test. We simulated this scenario using a leave-one-project-out cross validation considering all project. Specifically, we trained the MLP classifier on all the mutants of four projects and then tested it against the fifth project, repeating to test each project. Figure 12 gives the results (Leave 1 project out (All)), showing

Table 6: Area under ROC curve in various cross-validation settings, considering the MLP classifier and three dimensions of mutant utility.

Cross validation	**Area under the ROC curve**		
	Equivalency	Triviality	Dominance
Train/test on all subjects (All)	0.950	0.945	0.898
5-Fold (All)	0.737	0.761	0.596
Leave 1 project out (All)	0.684	0.697	0.523
Train/test on all subjects (Lang)	0.953	0.927	0.882
5-Fold (Lang)	0.815	0.839	0.761
Leave 1 subject out (Lang)	0.839	0.818	0.738
Random	0.500	0.500	0.500

that the predictions generated from training on different programs have lower accuracy, compared to the 5-fold cross validation. We conjecture that some programming constructs represented in the AST context are not equally common across all projects. Further, the numbers of mutants generated for each project differ, which is likely to decrease the accuracy when holding out a project that contributes a large fraction of the generated mutants. Nevertheless, with $AUC = 0.684$, training on other programs does have some predictive value for equivalent mutants, and is a better way to start testing than random mutant selection. Interestingly, training on other programs has approximately equally good predictive power for identifying equivalent and trivial mutants, but almost no ability to predict strong mutants.

As an engineer continues to incrementally develop software, the training material in terms of same-program mutants and associated tests grows. This means that at some point, the engineer could train a classifier on existing mutants from the same project and then use that classifier to predict mutant utility for a newly developed method, class, or component. We simulated this scenario using a leave-one-subject-out cross validation for the project Lang. Recall that a subject in this scenario is a unique class in the project Lang. We trained the MLP classifier on all the mutants of all but one subjects and tested it against the hold-out subject, repeating to test each subject. Again, Figure 12 and Table 6 show the results (Leave 1 subject out (Lang)), suggesting that training a classifier on different classes for presumably similar software yields good predictions. In particular, the predictive power for all three dimensions of mutant utility is considerably better (AUC between 0.74 and 0.84) when compared to the leave-one-project-out cross validation.

For comparison, Figure 12 also shows the ROC curves for the MLP classifier when trained and tested on all subjects for a single project (Train/test on all subjects (Lang)) and all subjects for all projects (Train/test on all subjects (All)). These best-case scenarios, which show higher predictive power than any of the simulated real-world scenarios, provide an upper bound and point of reference.

We conclude that program context features, in particular the mutation operator, AST context n-grams, node data type, and child node types are a useful feature set for machine learning algorithms to predict equivalent mutants, trivial mutants, and (within a given project) mutants with high dominator strength.

8 RELATED WORK

Redundant Mutants: The large number of mutants generated by mutation testing has long been a recognized problem. Mathur [46] determined that the complexity of mutation testing is $O(n^2)$, where n is the size of the program under test, and introduced the idea of *constrained mutation* to reduce that complexity to $O(n)$ by reducing the number of mutation operators to create fewer mutants. Offutt et al. [54, 57] took an empirical approach to defining an appropriate set of selective mutation operators, and proposed the *E-selective* set of five operators[9] based on achieving a mutation score of 0.99 or higher over ten small programs. Wong et al. [69, 70] evaluated combinations of mutation operators for efficiency and effectiveness.

Other researchers have examined whether selective mutation is more effective than random sampling of similar numbers of mutants. Acree [1] and Budd [14] separately concluded that executing tests that kill a randomly-selected 10% of mutants could provide results close to executing tests that kill the full set of mutants. Wong and Mathur [71] demonstrated similar results and found that adding randomly-selected mutants beyond 10% yielded comparatively small improvements. More recently, Zhang et al. [74] explored selective mutation and random selection using the Proteum mutation tool and the Siemens suite programs and also found no appreciable difference in performance between selective mutation and random selection.

Gopinath et al. [21] expanded this investigation using a much larger body of open-source code and compared several different mutation selection strategies with random selection, again finding that random selection performs as well as any other strategy. In a later paper, Gopinath et al. [22] took a different approach to dealing with the large number of mutants, and showed that determining the mutation score based on as few as 1,000 randomly-selected mutants provides an estimate of quality of a test suite in terms of mutation score.

It seems counterintuitive that targeted approaches to mutant selection should perform no better than a random approach, yet Kurtz et al. reassessed the performance of *E-Selective* mutation using dominator mutation score [42] and found it indistinguishable from both statement deletion and random mutant selection.

Barbosa et al. [11] applied a well defined set of guidelines to obtain a sufficient set of mutation operators that would substantially reduce the computational cost of mutation testing without losing effectiveness. They applied such guidelines in two experiments with two different sets of C programs. They obtained reduced sets of mutant operators that would produce effective test cases, but these sets of sufficient operators were substantially diverse between the experiments, showing that it was not possible to select a single set of operators that was optimal for both programs. Namin et al. [48–50] analyzed the Siemens suite programs using variable reduction techniques to identify three high-performing operator sets using between seven and 13 operators. Delamaro et al. [17] defined a growth model for mutation operator selection, adding operators using a greedy algorithm until a mutation score of 1.00 was achieved, and concluded that there is no single way to select the best set of operators for any particular program, a result consistent with Kurtz et al. [42].

Taking mutation operator reduction to an extreme, Untch [66] evaluated the performance of the statement deletion (SDL) operator on its own and found it to be competitive with the operator sets found by Namin. Deng et al. [20] applied the SDL operator to 40 classes written in Java using the muJava tool [44] and found that SDL achieved a mutation score close to that of Offutt's *E-selective* operators while generating approximately 80% fewer mutants. SDL also generates far fewer equivalent mutants than most mutation operators. Delamaro et al. [16] evaluated the SDL operator against programs written in C using Proteum and confirmed Deng's findings. Like *E-selective*, SDL sometimes results in very low dominator scores.

Several projects have addressed the notion that some mutants are more valuable than others, an idea Kurtz et al. made precise with subsumption graphs[41]. Yao et al. [72] suggest the notion of a *stubborn* mutant, defined as one that is not killed by a branch-adequate test set. While stubborn mutants are similar to dominator mutants (such as being difficult to kill), identifying both via dynamic testing have the same problem—they require an extensive test set and thus cannot be used to reduce the effort of creating a test set. Namin et al. [47] describe *MuRanker*, a tool to identify difficult-to-kill mutants based on the syntactic distance between the mutant and the original artifact. They postulate the existence of "super mutants" that are difficult to kill and for which a killing test may also kill a number of other mutants. This is closely related to what Kurtz et al. have formalized as *dominator mutants*[9, 41].

Kaminski et al. [36, 37] were the first to consider mutation operators at the next level of detail, recognizing that the relational operator replacement (ROR) mutation operator class (replacing '>' with '<', '!=', etc.). has many redundant sub-operators. They showed that, for any given relational operator, three mutants will always weakly subsume the other four mutants, making them redundant. Lindström and Márki [43] later showed that this subsumption does not always hold under strong mutation. Just et al. [34] performed a similar analysis for the conditional operator replacement operators, and Yao et al. [72] found similar results for the arithmetic operator replacement mutation operators. Just and Schweiggert [35] identified seven mutation operators from the COR, UOI, and ROR mutation operators classes showed that they form a sufficient set of non-redundant mutants, analyzed a set of real-world programs, and determined that redundant mutants cause an inflated mutation score that fails to accurately reflect the effectiveness of a test suite.

Existing mutation selection strategies are flawed, and, at the same time, more mutant operators are needed to generate mutants that couple to real faults. Hence, we need a viable selection strategy to cope with redundancy, and this paper provides evidence for the novel idea that context can help predict which mutants will have high dominator scores.

Equivalent Mutants: Baldwin and Sayward described how compiler optimization techniques could detect equivalent mutants [10]. Offutt and Craft investigated this approach in the context of Mothra [53], and reported an equivalent mutant detection rate of 15%. An extention of this approach to include infeasible constraint detection enabled Offutt and Pan to improve the detection rate to 45% [55, 56].

[9]The *E-selective* operators, derived from Mothra [19], are absolute value insertion (ABS), arithmetic operator replacement (AOR), logical connector replacement (LCR), relational operator replacement (ROR), and unary operator insertion (UOI).

Voas and McGraw suggest a program slicing approach to equivalent mutant detection [67], which Hierons et al. [26] formalized and Harman et. al extended with a notion of fine-grained dependence [24]. These works do not report empirical equivalent mutant detection rates.

Kintis and Malevris [40] used data flow analysis to detect equivalent mutants; in particular, mutants that change the value of a variable after its last use are always equivalent. Schuler and Zeller [62] attempted to identify equivalent mutants based on impact in terms of *coverage difference*, the difference in statement coverage between the original artifact and the mutant. Just et al. took this approach a step further and identify test-equivalent and equivalent mutants using state infection and local propagation conditions [30, 31]. While effective, these methods also requires a pre-existing test set.

Jia et al. surveyed mutation testing in general and provided a detailed review of mutation equivalence detection techniques [28]. Subsequent to this survey, Papadakis et al. proposed trival mutant detection [58], where program binaries are simply compared via diff. Surprisingly, this very simple approach yields a detection rate of 7% to 21%, with a reported potential improvement to 30%.

Harman et al. argued that a Higher-Order Mutant (HOM) approach might introduce fewer equivalent mutants than normal first-order approaches, and that a co-evolutionary approach to mutant generation should "almost guarantee that no equivalent mutants will be created" [2, 25].

Because equivalent mutants directly result in wasted work, limiting equivalent mutants to a very low number is one of the keys to making mutation analysis practical, and hence a key focus of this paper. Mutation analysis generates very large numbers of mutants, so if even a modest fraction of them are equivalent, the absolute number can be quite high, even after all known techniques for detecting equivalent mutants are applied. Hence we propose techniques to identify likely equivalent mutants based on program context.

Machine Learning for Program Analysis: Tripp et al. [64] applied machine learning techniques to the problems of false positives in static analysis. The training data was early feedback from developers in a given project; later false positives were identified by the classifier. The authors used machine learning implementations in a black-box manner by simply calling an appropriate API, and evaluated how well each machine learning technique performed. Our insight here is that treating equivalent mutants as false positives allows us to adopt this same approach in our efforts to identify equivalent mutants. In a closely related work, Hanam et al. [64] seek to identify for "actionable alerts" in static analysis. The goal of the machine learning approach here is to discover static-analysis alert patterns using code patterns built atop slices taken from a program's abstract syntax tree. Kanewala et al. characterized program features as node and path features of a program's control flow graph (CFG) [38].

Our approach also models program context using a program's abstract syntax tree, but we study a richer set of program context features (considering structural and data type context) and link them to equivalent, trivial, and dominator mutants, rather than actionable alerts.

Allamanis et al. piloted an investigation of *tailored mutants* [5] in an effort to increase the coupling of mutants to real faults in the Defects4J testbed. The additional mutation operators required to increase the coupling result in a large increase in total mutants. To combat this the paper uses a novel filtering approach to preferentially select mutants that are more likely to increase real-fault coupling. The filtering approach is based on "code naturalness", an idea originally developed by Hindle et al. [27], who used *n*-gram language models to show that source code is predictable and "natural". Hindle's seminal work attracted a number of refinements to the language models [6, 45, 52, 65]. Applications of these models include source code autocompletion, learning coding conventions and suggesting names [3, 4], predicting program properties [61], extracting code idioms [7], code migration [39, 51] and code search [8]. Campbell et al. [15] used code naturalness to provide more accurate syntax error messages from compilers. Bhatia and Singh [12] went further and used neural networks to provide automated correction of syntax errors. Wang et al. [68] extracted program features from abstract syntax trees and used deep belief networks to improve both within-project and cross-project defect prediction.

This large body of related work suggests that machine learning and natural language processing methods can be fruitfully applied to source code text for many purposes. This fact supports the central hypothesis of this paper, namely that machine learning of context information can be used to identify likely equivalent, trivial, and dominator mutants.

9 CONCLUSIONS

Mutation analysis is the "gold standard" for assessing the quality of test sets, and rightly so: mutation-adequate test sets are much more highly coupled to fault detection than test sets adequate for any other coverage criteria. Yet mutation testing is rarely used by practicing engineers. One of the reasons for this unfortunate state of affairs is that current mutation analysis techniques produce far more mutants than necessary and, even worse, many of these mutants are equivalent. The result is that engineers waste effort tracking down equivalent mutants and cannot easily assess how close they are to being "done".

To reduce both the total number of mutants and the number of equivalent mutants, the research community has developed various selective mutation approaches at the mutation operator group level. Recent research, however, has shown that the effectiveness of such techniques varies greatly across programs, with extremely poor effectiveness in some cases. The sobering conclusion is that selective mutation at the operator group level doesn't significantly outperform random mutant selection.

This paper promises to forge a new path out of the selective mutation dead end. The main result of this paper is that program context can predict mutant utility: some program contexts are more likely than others to yield mutants that are equivalent, or trivial, or have high dominator strength. The results of this paper are actionable in the following three ways.

First, the results strongly suggest that any useful selective mutation approach must be context-sensitive and must be applied at the mutation operator, rather than the mutation operator group, level.

Second, the results show program context dimensions and features within each dimension that can predict mutant utility. These findings can guide future work on developing more complex models for program context. While this paper focused on both structural and data type context, extracted from a program's abstract syntax tree, it is plausible that there is additional context information, not investigate in this study, that contributes a signal to mutant utility.

Third, the results suggest that machine learning classifiers, most notably decision trees and neural networks, are suitable for predicting mutant utility based on program context. The cross-validation results demonstrate high accuracy within and across projects. We conjecture that more complex models of and interactions between program context dimensions are likely to further improve the accuracy of these classifiers.

The broader vision of this paper is to customize program mutation to a target program, only generating mutants with high utility for that program. As a first step in realizing this vision, this paper has explored different dimensions of program context, which are predictive of mutant utility, and different machine learning techniques to train a classifier that enables such customization. We hope that the promising results will spawn research that will ultimately yield a turn-key, context-sensitive mutation system that engineers will widely adopt in practice.

ACKNOWLEDGEMENTS

We are pleased to acknowledge Huzefa Rangwala and Andrew McCallum for suggestions on applying machine learning techniques to predict mutant utility based on program context features. Bob Kurtz's research is supported by Raytheon.

REFERENCES

[1] Alan T. Acree. 1980. *On Mutation.* Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA.

[2] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. 2004. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Springer LNCS 3103, 1338–1349.

[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Symposium on the Foundations of Software Engineering (FSE).*

[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Symposium on the Foundations of Software Engineering (FSE).*

[5] Miltiadis Allamanis, Earl T. Barr, René Just, and Charles Sutton. 2016. Tailored mutants fit bugs better. *arXiv preprint arXiv:1611.02516* (2016).

[6] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the Tenth International Workshop on Mining Software Repositories.* IEEE Press, 207–216.

[7] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *Symposium on the Foundations of Software Engineering (FSE).*

[8] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of The 32nd International Conference on Machine Learning.* 2123–2132.

[9] Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. 2014. Establishing Theoretical Minimal Sets of Mutants. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014).* Cleveland, Ohio, USA, 21–31.

[10] Douglas Baldwin and Fred G. Sayward. 1979. *Heuristics for Determining Equivalence of Program Mutations.* Research Report 276. Department of Computer Science, Yale University.

[11] Ellen Francine Barbosa, José C. Maldonado, and Auri Marcelo Rizzo Vincenzi. 2001. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification, and Reliability, Wiley* 11, 2 (June 2001), 113–136.

[12] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *arXiv preprint arXiv:1603.06129* (2016).

[13] Andrew P. Bradley. 1997. The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms. *Pattern Recogn.* 30, 7 (July 1997), 1145–1159. DOI: http://dx.doi.org/10.1016/S0031-3203(96)00142-2

[14] Tim A. Budd. 1980. *Mutation Analysis of Program Test Data.* Ph.D. Dissertation. Yale University, New Haven, Connecticut, USA.

[15] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 252–261.

[16] Marcio E. Delamaro, Lin Deng, Serapilha Dureli, Nan Li, and Jeff Offutt. 2014. Experimental Evaluation of SDL and One-Op Mutation for C. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014).* Cleveland, Ohio.

[17] Márcio E. Delamaro, Lin Deng, Nan Li, and Vinicius H. S. Durelli. 2014. Growing a Reduced Set of Mutation Operators. In *Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES).* Maceió, Alagoas, Brazil, 81–90.

[18] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (April 1978), 34–41.

[19] Richard A. DeMillo and Jeff Offutt. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering* 17, 9 (September 1991), 900–910.

[20] Lin Deng, Jeff Offutt, and Nan Li. 2013. Empirical Evaluation of the Statement Deletion Mutation Operator. In *6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013).* Luxembourg, 80–93.

[21] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. *Do Mutation Reduction Strategies Matter?* Technical Report. School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, Oregon, USA.

[22] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be anyway?. In *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE).*

[23] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the Limits of Mutation Reduction Strategies. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* ACM, 511–522.

[24] Mark Harman, Rob Hierons, and Sebastian Danicic. 2001. Mutation Testing for the New Century. Kluwer Academic Publishers, Chapter The Relationship Between Program Dependence and Mutation Analysis, 5–13.

[25] Mark Harman, Yue Jia, and William Langdon. 2010. A Manifesto for Higher Order Mutation Testing. In *Sixth IEEE Workshop on Mutation Analysis (Mutation 2010).* Paris, France, 80–89.

[26] Rob Hierons, Mark Harman, and Sebastian Danicic. 1999. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability, Wiley* 9, 4 (December 1999), 233–262.

[27] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *International Conference on Software Engineering (ICSE).* IEEE, 837–847.

[28] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering* 37, 5 (September 2011), 649–678.

[29] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* 433–436.

[30] René Just, Michael D. Ernst, and Gordon Fraser. 2013. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. In *Proceedings of the Dagstuhl Seminar 13021: Symbolic Methods in Testing,* Vol. abs/1303.2784. arXiv:1303.2784, preprint.

[31] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* San Jose, CA, USA, 315–326.

[32] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* San Jose, CA, USA, 437–440.

[33] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering.* Hong Kong, 654–665.

[34] René Just, Gregory M Kapfhammer, and Franz Schweiggert. 2012. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE).* 11–20.

[35] René Just and Franz Schweiggert. 2015. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification, and Reliability, Wiley* 25, 5-7 (2015), 490–507.

[36] Garrett Kaminski, Paul Ammann, and Jeff Offutt. 2011. Better Predicate Testing. In *Sixth Workshop on Automation of Software Test (AST 2011).* Honolulu HI, 57–63.

[37] Garrett Kaminski, Paul Ammann, and Jeff Offutt. 2013. Improving Logic-Based Testing. *Journal of Systems and Software, Elsevier* 86 (August 2013), 2002–2012. Issue 8.

[38] Upulee Kanewala and James M Bieman. 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 1–10.

[39] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 173–184. DOI:http://dx.doi.org/10.1145/2661136.2661148

[40] Marinos Kintis and Nicos Malevris. 2014. Using Data Flow Patterns for Equivalent Mutant Detection. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*. Cleveland, Ohio.

[41] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutation Subsumption Graphs. In *Tenth IEEE Workshop on Mutation Analysis (Mutation 2014)*. Cleveland, Ohio, USA, 176–185.

[42] Robert Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the Validity of Selective Mutation with Dominator Mutants. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '16)*. ACM, Seattle, Washington, 571–582.

[43] Birgitta Lindström and András Márki. 2016. On Strong Mutation and Subsuming Mutants. In *Twelfth IEEE Workshop on Mutation Analysis (Mutation 2016)*. Chicago, Illinois, USA.

[44] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: A Mutation System for Java. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06), tool demo*. IEEE Computer Society Press, Shanghai, China, 827–830.

[45] Chris J Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *International Conference on Machine Learning (ICML)*.

[46] Aditya Mathur. 1991. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*. Tokyo, Japan, 604–605.

[47] Akbar Namin, Xiaozhen Xue, Omar Rosas, and Pankaj Sharma. to appear. MuRanker: A mutant ranking tool. *Software Testing, Verification, and Reliability* (to appear). Published online August 2014.

[48] Akbar Siami Namin and James H. Andrews. 2006. Finding Sufficient Mutation Operators via Variable Reduction. In *Second Workshop on Mutation Analysis (Mutation 2006)*. Raleigh, NC.

[49] Akbar Siami Namin and James H. Andrews. 2007. On Sufficiency of Mutants. In *Proceedings of the 29th International Conference on Software Engineering, Doctoral Symposium*. ACM, Minneapolis, MN, 73–74.

[50] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. 2008. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, Leipzig, Germany, 351–360. DOI:http://dx.doi.org/10.1145/1368088.1368136

[51] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 544–547.

[52] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 532–542.

[53] Jeff Offutt and William Michael Craft. 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification, and Reliability, Wiley* 4, 3 (September 1994), 131–154.

[54] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutation Operators. *ACM Transactions on Software Engineering Methodology* 5, 2 (April 1996), 99–118.

[55] Jeff Offutt and Jie Pan. 1996. Detecting Equivalent Mutants and the Feasible Path Problem. In *Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*. IEEE Computer Society Press, Gaithersburg MD, 224–236.

[56] Jeff Offutt and Jie Pan. 1997. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability* 7, 3 (September 1997), 165–192.

[57] Jeff Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*. IEEE Computer Society Press, Baltimore, MD, 100–107.

[58] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 936–946.

[59] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[60] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and others. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[61] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from âĂIJBig CodeâĂİ. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 111–124.

[62] David Schuler and Andreas Zeller. 2010. (Un-)Covering Equivalent Mutants. In *3rd IEEE International Conference on Software Testing, Verification and Validation (ICST 2010)*. Paris, France, 45–54.

[63] S. Theodoridis. 2015. *Machine Learning: A Bayesian and Optimization Perspective*. Elsevier. https://books.google.com/books?id=hxQRogEACAAJ

[64] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, 762–774.

[65] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 269–280.

[66] Roland Untch. 2009. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In *ACM Southeast Regional Conference*. Clemson, SC, 19–21.

[67] Jeffrey M. Voas and Gary McGraw. 1997. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, Inc., New York, NY, USA.

[68] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Austin, Texas, 297–308.

[69] W. Eric Wong, Márcio E. Delamaro, José C. Maldonado, and Aditya P. Mathur. 1994. Constrained Mutation in C Programs. In *Proceedings of the 8th Brazilian Symposium on Software Engineering*. Curitiba, Brazil, 439–452.

[70] W. Eric Wong and Aditya P. Mathur. 1995. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software, Elsevier* 31, 3 (December 1995), 185–196.

[71] W. Eric Wong and Aditya P. Mathur. 1995. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software* 31, 3 (December 1995), 185–196.

[72] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence. In *Proceedings of the 36th International Conference on Software Engineering, (ICSE 2005)*. IEEE Computer Society Press, Hyderabad, India.

[73] Du Zhang and Jeffrey J. P. Tsai. 2007. *Advances in Machine Learning Applications in Software Engineering*. IGI Global, Hershey, PA, USA.

[74] Lu Zhang, Shan-San Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *32nd ACM/IEEE International Conference on Software Engineering*. 435–444.