Are mutants really natural? A study on how "naturalness" helps mutant selection

Matthieu Jimenez
SnT, University of Luxembourg
Luxembourg, Luxembourg
matthieu.jimenez@uni.lu

Thiery Titcheu Checkam
SnT, University of Luxembourg
Luxembourg, Luxembourg
thiery.titcheu@uni.lu

Maxime Cordy
University of Namur
Namur, Belgium
maxime.cordy@unamur.be

Mike Papadakis
SnT, University of Luxembourg
Luxembourg, Luxembourg
michail.papadakis@uni.lu

Marinos Kintis & Yves Le Traon
SnT, University of Luxembourg
Luxembourg, Luxembourg
{marinos.kintis,yves.letraon}@uni.lu

Mark Harman
University College London and Facebook
London, UK
mark.harman@ucl.ac.uk

ABSTRACT

Background: Code is repetitive and predictable in a way that is similar to the natural language. This means that code is "natural" and this "naturalness" can be captured by natural language modelling techniques. Such models promise to capture the program semantics and identify source code parts that 'smell', i.e., they are strange, badly written and are generally error-prone (likely to be defective). **Aims:** We investigate the use of natural language modelling techniques in mutation testing (a testing technique that uses artificial faults). We thus, seek to identify how well artificial faults simulate real ones and ultimately understand how natural the artificial faults can be. Our intuition is that natural mutants, i.e., mutants that are predictable (follow the implicit coding norms of developers), are semantically useful and generally valuable (to testers). We also expect that mutants located on unnatural code locations (which are generally linked with error-proneness) to be of higher value than those located on natural code locations. **Method:** Based on this idea, we propose mutant selection strategies that rank mutants according to a) their naturalness (naturalness of the mutated code), b) the naturalness of their locations (naturalness of the original program statements) and c) their impact on the naturalness of the code that they apply to (naturalness differences between original and mutated statements). We empirically evaluate these issues on a benchmark set of 5 open-source projects, involving more than 100k mutants and 230 real faults. Based on the fault set we estimate the utility (i.e. capability to reveal faults) of mutants selected on the basis of their naturalness, and compare it against the utility of randomly selected mutants. **Results:** Our analysis shows that there is no link between naturalness and the fault revelation utility of mutants. We also demonstrate that the naturalness-based mutant selection performs similar (slightly worse) to the random mutant selection. **Conclusions:** Our findings are negative but we consider them interesting as they confute a strong intuition, i.e., fault revelation is independent of the mutants' naturalness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEM '18, October 11–12, 2018, Oulu, Finland
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3240500

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Mutation testing, Fault Revelation, Language Models

ACM Reference Format:
Matthieu Jimenez, Thiery Titcheu Checkam, Maxime Cordy, Mike Papadakis, Marinos Kintis & Yves Le Traon, and Mark Harman. 2018. Are mutants really natural? A study on how "naturalness" helps mutant selection. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239235.3240500

1 INTRODUCTION

Empirical and experimental evaluations of software testing are typically performed by using artificial faults. These faults are seeded in selected programs and are used as the objectives for comparing techniques. Thus, the techniques and test cases are assessed by measuring their ability to detect these types of faults.

This type of assessment is known as fault seeding or mutation testing. Fault seeding is performed by altering the syntax of the programs. Thus, researchers transform (mutate) the syntax of the programs with the aim of generating program versions (mutants) that are semantically different. By demonstrating (revealing) the semantic differences between the mutants and the original program, one can effectively measure test effectiveness [3, 7, 40].

Evidently, as the mutant faults are generated by altering the programs' syntax, they alter the program semantics. However, in practice, most of the mutants tend to have a major effect on the program semantics, which makes them non-useful to testers (since they are trivial and can be revealed by many tests). On the contrary, testers need mutants with a small effect on the program semantics as these are hard to reveal and result in strong test cases [33, 37]. Nonetheless, the key question is how well mutants (which are in a sense artificial faults) mimic real code and real faults?

To this end, recent research has indeed shown that some (very few) mutants are realistic [37, 40]. However, since the number of realistic mutants is very small, compared to the total number of mutants [38, 40], these have almost no practical effect [7, 37]. In other words, mutation introduces a very large number of non-interesting (bad) mutants and very few interesting (good) ones. This raises the question of how to select mutants that are semantically useful and natural, e.g., simulate well real code and faults.

To identify semantically usefull mutants, we need a model capable of capturing the goodness of mutants. Previous research has focused on identifying the types of mutants that are the most important ones [25, 39]. However, these techniques have little or no success as they fail to outperform the random mutant selection [6, 25]. One potential explanation could be that it is the location of the mutants that makes them good and not their type. Another potential explanation could be that good mutants are the result of the combination of the location with the mutant type.

Nevertheless, we need a model capable of identifying the interesting program locations, interesting mutant types and interesting pairs of location and types. To this end, we investigate the use of Language Models (LMs), such as N-Grams, as an approximation mechanism for capturing the program semantics and select mutants. We believe that language models can exploit the implicit rules, coding conventions and generally repetitiveness of source code and categorise mutants (and their locations) as "natural" i.e., mutated code that is likely to appear in a codebase (follows the implicit coding norms of developers), and "unnatural", i.e., mutated code that is unlikely to appear in codebase.

Naturally, the LMs and the notion of "naturalness" raises the question of how natural or unnatural mutants are, since they are simulated faults. This intriguing question motivated our study and desire to understand the properties and connections between program syntax and semantics from a testing (fault revelation) perspective.

Interestingly, previous research has shown that the notion of "naturalness" is powerful [15] and capable of capturing code semantics. Naturalness has been useful in suggesting code [45], checking compliance with standards [1], and identifying error-prone code parts [43]. Therefore, our intuition is that natural mutants (considered as probable by such models) are more valuable than the unnatural ones because they follow the implicit norms and the way programmers code. We expect that mutants located on unnatural code locations (which previous research linked with error-proneness [18, 43]) to be of higher value than those located on natural code locations.

In essence, the question regards the likelihood for developers to do things wrong. Natural code fragments are easier for developers to compose and more probable of being semantically right (since they are highly repetitive) than unnatural code fragments. Thus, we expect that mutants making a code fragment more natural, while at the same time being semantically different from the original version, to have more utility than mutants making a code fragment less natural. This is because such mutants are likely to introduce expected semantic deviations, which have small effect on the program semantics. Furthermore, such mutants are worth investigating since they form likely alternatives to the original code.

To investigate our hypothesis, we consider a set of real bugs from 5 Java open source projects. We measure naturalness at both the file level granularity (used to compute the naturalness of mutated files, i.e., Java classes) and at the statement level (used to compute the naturalness of the original and mutated code statements).

We use the naturalness measurements to rank the mutants according to: a) the naturalness of mutated code files, b) the naturalness of the original code statements and c) the impact on the naturalness of the mutated statement(s) (difference on the naturalness of the original and mutated code). We evaluate these ranks w.r.t. their probability to be killed by test cases that reveal real faults. Thus, we assess whether mutants ranked higher are indeed preferable than those ranked lower (i.e. their killing implies the revelation of real faults).

Our results are negative. We deemed them as interesting since they confute intuition and increase our understanding of the interconnections of program syntax, program semantics and software faults. We show that the fault revealing utility of mutants is independent of their naturalness, which in a sense suggests that naturalness is not a discriminative factor for mutant selection. Interestingly, we find that fault revealing mutants are spread across both natural and unnatural code fragments in such a way that naturalness-based mutant selection is equivalent to the random one.

The rest of the paper is organized as follows: Section 2 discusses about mutation testing and naturalness of software. Sections 3 and 4 present the objectives of our study and the experimental design. Our findings are reported on Section 5, while Sections 6 and 7 discuss the threats to validity and related studies. Finally, Section 8 concludes the paper.

2 BACKGROUND

2.1 Mutation Testing

Mutation is a well-studied technique with increasing popularity among researchers and practitioners alike, as it is evident from the most recent survey in the area [39]. Mutation works by inserting artificial faults into the program under test, termed the *original program*; thus, creating many different versions of it, each one containing a single syntactic change. These versions are called *mutants*. Mutants are used to evaluate test cases based on their ability to distinguish the mutants' behavior from that of the original program. If such a test case exists (or can be created) for a particular mutant, then we term the mutant *killed* (or *killable*). We term a mutant "fault-revealing" with respect to a particular fault if the test cases that kill it are a subset of the test cases that can also reveal that fault, i.e. lead the program under test to an observable failure.

Not all mutants can be killed by test cases. In such a case, we say that the mutants remain *live* and we need to investigate why this happened. A mutant can remain live after its execution with test cases for two reasons: first, the test cases are not "strong" enough to exhibit the behavioral differences between the mutant and the original program, thus, indicating a weakness of our test suite; or the mutant is an *equivalent* one. Equivalent mutants are syntactically different versions of the original program but semantically equivalent, meaning that their behavior is the same to the original program for all the possible inputs [22, 30].

Mutation systematically introduces syntactic changes to the original program. These changes are based on specific, predefined rules called *mutations* or *mutant operators*. Such operators can replace relational operators with each other, replacing > with <, for example, or increase the values of variables by inserting appropriate arithmetic operators to variable usages. Research has shown that the choice of mutation operators and their implementation affects the effectiveness of mutation and its tools [3, 26], thus, it is important to carefully select the mutants and the tools that one uses when applying mutation.

In mutation testing the identification of "valuable" mutants is a known open issue [37, 39]. Previous research has shown that the majority of the mutants is redundant and this can induce severe problems in the mutation test assessment process [25, 38]. This means that not all the mutants are of equal value. Indeed, some few mutants are useful, while the rest (majority) are easy-to-kill, are duplicates of other mutants [22], or are redundant wrt. to the useful ones [37]. This begs the question: *How can we distinguish the valuable mutants before analysing them?*, or equally, *Do valuable mutants have specific properties that can distinguish them from the less-valuable ones?* Motivated by these questions we investigate whether the notion of software naturalness can formulate mutant selection strategies.

2.2 Naturalness of Software

Code is a form of human communication and as such it tends to follow patterns and norms that are similar to those found in natural languages. As such, code repetitiveness has been shown to be of interest for software engineering [14].

Indeed, experienced developers prefer to write code that is easily maintainable, i.e., well structured, readable and concise, which, as a side effect, induces repetitiveness. Hindle et al. [15] showed that this could be used to train probabilistic models like LMs. Code LMs have been shown to be of interest for a large variety of applications, including auto-completion of code [45] and defect prediction [18, 43]. They are able to compute a probability score of a given code fragment, such that a high score means that the code fragment is very natural.

The naturalness of code is a research area with a growing interest in recent years, as witnessed by the recent survey of the area [2]. This field is based on the same premise as naturalness, i.e., code is similar to natural languages, and aims at applying decades of works in Natural Language Processing (NLP) to improve Software Engineering. The idea to consider code not only as an instruction of a developer to a computer but also as a form of communication between humans is not recent, and can in fact be traced back to the work of Knuth [24] on literate programming. However, works on this area have really started 30 years later with the aforementioned work of Hindle et al. [15]. Overall, according to Allamanis et al. [2] the naturalness hypothesis has been defined as:

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

This paper can be classified according to Allamanis et al. [2] 's taxonomy as code-generating probabilistic models relying on LMs.

2.3 Code Language Models

LMs assign a non-zero probability to every possible slice of code without requiring any prior knowledge. These models can be evaluated using three different metrics: cross-entropy, perplexity and word error rate. Cross entropy originates from the information theory area and measures the degree of surprise of the trained model M given a slice of code s composed of n tokens, i.e., the average number of bits per token required to encode s given M using a perfect code. A lower cross entropy value is better, as it means that M is less surprised by s. More formally, cross-entropy is given by:

$$H_M(s) = -\frac{1}{n} log_2(p_M(a_1 ... a_n))$$

where p_M is the probability that M observes the sequence of token $a_1 ... a_n$. The perplexity metric is the reciprocal of the geometric average probability assigned by model M to each token of s [8]. It is given by $PP_M(s) = 2^{H_M(s)}$.

The most commonly used LMs for naturalness are the N-Gram Models [15, 45]. These rely on the Markov's property, i.e., the occurrence of a token is influenced only by a limited number (n) of previous tokens (gram). Hence, the probability of s can be approximated by the probability of all token sequences of size n it contains.

The probability of an n-gram N can be computed using a standard maximum likelihood estimate, where the number of occurrences of N in a training corpus is divided by the number of occurrences of the m first token of N. However due to data sparsity, it is likely that some n-gram that never appeared in a training corpus will appear in s. This n-gram would then get a probability of 0 leading to an infinite cross entropy, which is not possible for a LM. To circumvent this problem, the NLP community came up with a family of techniques called smoothing [8, 29]. Smoothing techniques take a part of the probability of existing N-grams and attribute it to non-existing ones by extrapolating on the information given by m-grams, where $m < n$.

In this paper, we aim at investigating whether mutation testing can benefit from naturalness analysis. Following the lines of previous work [15, 18, 45], we base our investigations on N-Gram models because these are simple and fast to compute.

3 RESEARCH QUESTIONS

We start our investigation by checking whether mutants alter the naturalness of a project and to which direction (make the code more natural or unnatural). This poses our first research question:

RQ1: What is the impact of mutants on the naturalness of code?

We answer this question by checking the differences in the naturalness of the original and mutant program files. We also check the number of mutants having the same naturalness values. The answer to this question ensures that we can leverage natural language models in mutation testing. Given that we found evidence that mutants have different naturalness values, we turn to design naturalness-based mutants selection strategies. We thus, investigate the fault revelation ability of the mutants that can be categorised as natural and unnatural. Hence:

Table 1: Java Subjects' Details

Test Subject	Description	LoC	#Faults Used
JFreeChart	A chart library	79,949	19
Closure	Closure compiler	91,168	92
Commons Lang	Java utilities library	45,639	30
Commons Math	Mathematics library	22,746	74
Joda-Time	A date and time library	79,227	15
Total	-	**318,729**	**114**

RQ2: Is "natural" mutant selection stronger than the "unnatural" mutant selection?

To answer RQ2, we need to know the the probability of revealing a fault when killing a mutant, for every mutant in our set. We therefore repeatedly applied mutation testing on our benchmark sets and compute the fault revelation of both natural and unnatural mutant sets (of different sizes).

We report on the differences in fault revelation of three strategies based on: (1) the naturalness of mutated code fragment; (2) the naturalness of the original code fragments; and (3) the impact of mutants on the naturalness of the code (entropy difference between the original and mutated code fragments). This information is useful for designing effective naturalness-based mutant selection strategies.

After experimenting with the different naturalness-based mutation testing strategies, we evaluate them with respect to other baseline methods. We select the random mutant selection as baseline since previous research showed that it is indeed the most effective mutant selection strategy [6, 25]. Thus, our next RQ is:

RQ3: How does naturalness-based mutant selection compares with random selection?

To demonstrate whether there are benefits related to the naturalness-based mutation testing, we repeatedly compute the fault revealing potential of the studied approaches and compared them on the basis of fault revelation.

4 METHODOLOGY

This section presents details related to experimental settings, i.e., test subjects, test suites, real-faults and tools, and the evaluation procedure that we use in the experiments.

4.1 Test Subjects: Real Faults, Mutants and Test Suites

To answer our RQs, we use 5 real-world projects and 230 real-world bugs from the Defects4J database [20]. Defects4J includes a reproducible set of real faults mined from source code repositories, along with scripts that facilitate the conduction of experiments on these faults. In total, we considered 357 real-world faults accompanying our test subjects. For each fault, the database provides the faulty version of the project, the fixed one and at least one test case that triggers the faulty behaviour, i.e. fails when executed against the faulty version of the project.

Table 1 present details about the test subjects. The first four columns of the table record the subject names, their description, their source code lines[1] and the number of faults that they include.

To compose test pools with large number of tests, we used the data from the study of Papadakis et al. [40], which involved two state-of-the-art test generation tools (EvoSuite [13] and Randoop [36]). The test pools are composed of the available developer test suites and 20 test suites, 15 from EvoSuite and 5 from Randoop. Randoop has 5 test suites since it generates a vast number of tests, which impose a big overhead on the experiment. In total the test pools are composed of 1,375,341 automatically generated tests and 58,131 tests from the project developers.

For the creation of mutants, Major [21] was used. Major implements the main mutation operators [34], i.e., the Arithmetic (AOR), Logical Connector Replacement (LCR), Relational (ROR), Bitwise (BTW), Shift (SFT), Unary Operator Insertion (UOI) and Statement Deletion (SDL).

Major is robust, easy to use and has been used in many empirical studies [39, 40]. It also operates at the source code level, which is mandatory for calculating the naturalness of code. In our experiments we applied the tools on the fixed program versions of the datasets using all the supported operators.

4.2 N-Gram Model Building

We built N-Gram LMs using all the source files of the selected projects. We then evaluate the naturalness of mutants using their own file and code fragments that they apply.

To measure naturalness we need a tokenizer and a tool to build LMs. The first one should be able to transform source code into a sequence of tokens, while the second one builds the models from a training (tokenized) corpus and returns the cross entropy of the targeted data (i.e. the testing corpus).

We tokenize the source code files according to the grammar of their language [18] using the the Java Parser [41].

We study two tokenization schemes; one at the file level of granularity and one at the statement level. The *file-level tokenized content* is the result of tokenizing the code files (Java classes) as as a whole. The *line-level tokenized content* is the result of separating the tokens according to the statements they belong in the code files as suggested by Jimenez et al. [18]. It is noted that the comments were discarded.

We built the LMs using the Kylm toolkit [32] and the TUNA infrastructure [19]. Kylm is a reference LM toolkit that is written in Java and offers all the required functionalities for our experiments. To build N-Gram models, three main parameters are needed: (1) the maximum value of n, (2) the smoothing technique and (3) the unknown cut off. The first two parameters have been presented in Section 2.3. The unknown cut off represents a threshold on the number of times a token should appear before being considered by the model. If a token (after completing the training) fails to reach this threshold, it will be stored in a specific group of tokens that will be considered as 'unknown' by the model. This allows the generalization of the model on (being able to handle) 'unseen' tokens (cases of tokens that have not been appeared before, such as variable names) when evaluating the testing corpus.

[1] Reported by the `cloc` tool (http://cloc.sourceforge.net/).

In our experiments, we followed the best pracices suggested by Jimenez et al. [18] and pick the following parameters: n equals to 4 with *Modified Kneser Ney* as smoothing technique. We also set the unknown cut off to 1, the default value of Kylm.

Overall, to compute the naturalness of all mutants, we proceed as follows (for every mutated file):

(1) Collect and tokenize all source code files of the projects containing the mutated file under evaluation
(2) Exclude the file that has been mutated
(3) Use the resulting set of source files as the training corpus to build *two N-Gram models*, one at the file-level and another one at the line-level
(4) For the mutant files (test corpus):
    - Tokenize the mutant.
    - Compute its cross entropy as well as the original one using the *file level model*.
    - Do a `diff` between the line level tokenized versions of the original and the mutant.
    - Measure the cross entropy of the deleted lines and added lines using the line level model and attribute it to the original and mutant, respectively.

4.3 Evaluation Process

To answer the stated RQs, we applied mutation testing on the project files where the selected faults appeared. We then measured the cross entropy of all the original and mutant files using the process described above. Since the models are measuring the naturalness of code fragments based on the training corpus, we need to separate the training from the evaluation corpus in order to avoid biasing the ability of the model (judging code as natural because it is part of the training corpus).

Thus, for each fault, we train our models on all the project files excluding the faulty ones. This establishes a clear separation of training and evaluation targets as it ensures that the same files do not belong to both training and evaluation.

To answer RQ1 and show that the syntactic differences of mutants can be scored by language models (ranked according to their naturalness), we collect all mutants and categorize them according to the entropy of the original files (we record the entropy differences of the original and mutant files). We thus seek to identify trends regarding the syntactic transformations introduced by the mutants, e.g., whether mutants make natural files more or less natural.

To answer RQ2 and demonstrate the ability of the LMs to assist mutation testing, we rank the mutants according to their naturalness. We investigate three scenarios: the naturalness of the a) mutant location, b) mutated file and c) absolute difference differences of the original and mutant files. To check whether natural or unnatural cases are interesting, we rank the mutants in an increasing and decreasing order (of entropy) and contrast their fault revelation abilities. To determine fault revelation we repeatedly apply mutation testing by selecting the x% of the top rank mutants (we consider sets of 0, 5%, 10%, 15% to 100%) and compute the fault revelation probabilities of these sets. To account for coincidental and other random factors, we applied our process 1,000 times for every considered set of mutants.

Table 2: Distribution of entropy values. Number of mutants with equal entropy values and their frequency. For instance 83,707 mutants have unique entropy values, while 10,347 mutants have entropy value equal to another mutant.

No of mutants	1	2	3	4	5	6	7	8	>=9
Frequency	83,707	10,347	2,506	1,203	550	361	253	176	511

The fault revelation probabilities of the selected mutant sets were computed by measuring the ratio of the times that the faults were revealed by the test suites that kill all the considered mutants. The test suites that kill the candidate mutant sets were selected based on the following procedure: We start from empty test sets and stop when we kill all the mutants. At each step we select the next mutant in the list and randomly pick a test that kills it (selected from the pool of the available test suites). To avoid composing test suites with large redundancies we remove all the mutants that are killed by every test we select. In case no test kills a targeted mutant, we discard the mutant. This process mimics what a tester does when she uses mutation testing [3] and ensures that the selected tests are relevant to the mutants we study. Overall, this is a typical evaluation process that has been followed by many studies [3, 26, 38].

To answer RQ3 and compare with the random mutant selection we repeat the process followed in RQ2 for random mutant orderings. To cater for the stochastic nature of the random orderings we repeat this process 100 times and compare our results with the naturalness-based orderings.

5 RESULTS

5.1 RQ1: Impact of Mutants on Naturalness

Our first research question checks whether mutants change the cross entropy of the code under analysis. Thus, we check the ability of the LMs at identifying the syntactic changes introduced by mutants. To do so, we compare the cross entropy of the mutated (and original) code files.

Table 2 records the distribution of entropy values of the mutants we study. From these data we can see that entropy can distinguish the great majority of the mutants (very small sets of mutants are of identical entropy values). Fig. 1 presents the entropy differences between the mutant and the original files. The boxplots present the values resulting by the subtraction of the entropy of the mutated and the original files. Thus, positive values indicate that mutants are less natural than the original files, while negative values indicate the opposite.

Perhaps not surprising, we observe that models can indeed capture the syntactic differences between the original and the mutant files. Interestingly, we observe that mutants sometimes make the code more natural and sometimes less natural. The tendency is to make the code less natural as the majority of the mutant files have higher entropy, than the original files. In our results, only 30% of the mutant files are more natural than the original files.

Overall, our results demonstrate that mutants change the cross entropy of the code under analysis by making it sometimes more and sometimes less natural. This leads us to the question of whether there is a link between the value of mutants and their naturalness.

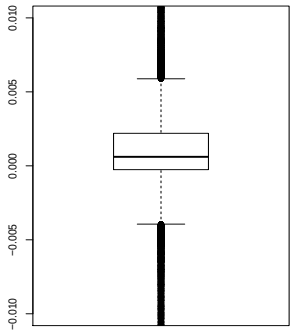


Figure 1: Cross Entropy difference between mutant and original files. We observe that mutants alter the cross entropy of the code with the majority of them making the code less natural.

5.2 RQ2: "Natural" Vs "Unnatural" Mutant Selection

This RQ regards the design of naturalness-based mutant selection strategies. To this end, we need to evaluate whether natural or unnatural mutants are preferable. We thus, collect all the mutants from our subjects and measure their entropy, using both the file-level and statement-level tokenizers, see section 4.2 for details. We then collect a) the entropy of the code fragments where mutants are applied (using, the statement-level tokenizer), b) the entropy of the mutant files (using, the file-level tokenizer), and c) the absolute difference in entropy between the original and mutated files (using, the file-level tokenizer), and form naturalness-based mutation testing strategies (by ranking mutants in an increasing and decreasing order).

These three entropy measurements help us investigate which ones of the strategies we can compose leads to interesting mutant sets. We investigate these particular cases because they involve likely (intuitively) interesting properties: the case a) regards the locations that should be mutated, the case b) regards the natural/unnatural order of mutants, and the case c) regards the 'extreme' mutants, i.e., choosing mutants that impact the entropy measure too much, either by making the code much more natural or much more unnatural.

We apply naturalness-based mutation testing by ranking mutants in an increasing and decreasing entropy order, i.e., we follow the procedure explained in Section 4.3, and obtain the fault revelation ability of our mutant sets. Here we compare the increasing and decreasing entropy orders in order to identify the strategy that leads to the most promising results. We use the same number of mutants in every comparison to establish a fair comparison.

In the following subsections we discuss the results related to the cases a), b) and c). All our results are presented by computing the differences in the fault revelation values of the increasing and decreasing order strategies. Thus, by observing positive values we can conclude that increasing strategies are preferable over the decreasing ones.

5.2.1 Mutant Locations. Figure 2(a) depicts the results related to the fault revelation ability of the mutants located on natural and unnatural code locations for several ratios of selected mutants. Higher values indicate that natural locations are preferable. We observe a small difference in favour of the natural locations over the unnatural ones. To validate this, we performed a Wilcoxon signed-rank test and found no statistical differences (at the $a < 0.05$ significance level). These results suggests that the naturalness of the code locations is not discriminative of the fault revelation ability of the mutants. In other words ranking mutants according to the naturalness of their location is not really helpful (is not a good feature of the semantic usefulness of the mutants).

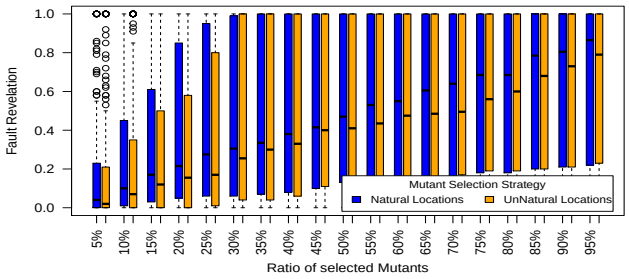
5.2.2 Mutant Files. Figure 2(b) records the results regarding the fault revelation ability of the natural and unnatural mutants. Natural mutants are those having mutant files (whole files) with low entropy. In this case the results show a tendency towards the natural mutants but the difference is small. By performing a Wilcoxon signed-rank test we find statistical differences (at the $a < 0.05$ significance level) when selecting in the range 5% to 25%. When selecting more than 25% of the mutants the differences are not significant. When measuring the effect size of the differences, using the Vargha and Delaney $\hat{A}_{12}$ [46], we get values of approximately 0.57 to 0.58 (meaning that natural mutants are preferable in 57-58% of the cases). Interestingly the fault detections of both natural and unnatural mutants are much higher than those of the natural or unnatural locations indicating a potential of such strategies.

5.2.3 Mutants Impact. Figure 2(c) records the results regarding the strategies with extreme impact, i.e., abs(original entropy - mutant entropy) (impact on the entropy of the file). The underlying idea is that the 'extreme' mutants (mutants with largest impact) are of higher value than the non-extreme ones. Unfortunately, the results indicate that this choice does not make any big difference (since almost all such values are close to each other). The differences are not statistically significant indicating that the impact on the naturalness is not a discriminative factor that we could use.

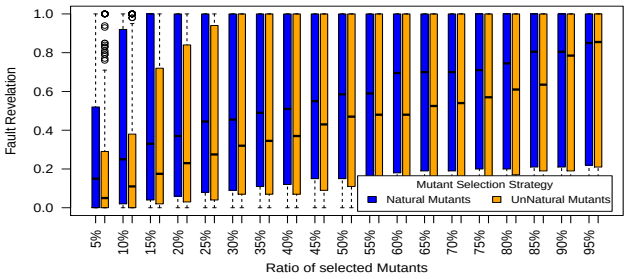
5.3 RQ3: Naturalness-based Mutant Selection VS Random

This RQ regards the comparison of naturalness-based mutation testing with a baseline in order to see if it is of any practical value. To investigate this, we select the two best performing strategy (i.e., selecting the most natural mutants) and compare them with the random selection. As we discussed earlier random mutant selection forms a tough baseline and thus, by demonstrating that the LMs outperforms the random selection, we effectively establish an approach capable of discriminating between good and bad mutants.

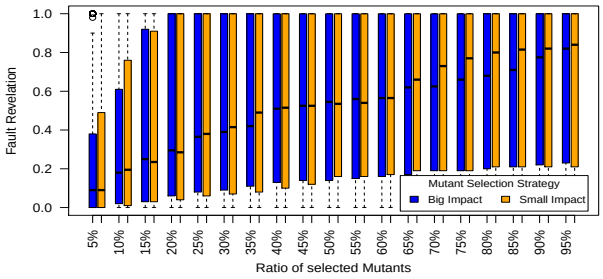
Figure 3 summarizes the results of the comparison. The boxes record the fault revelation ability of the mutant sets that are composed of (0-100%, in steps of 5%) of the considered mutants. As can be seen the naturalness-based strategy performs similarly to the random mutant selection. By performing a Wilcoxon test, we find that the results are not statistically significant. This means that the differences are marginal and we cannot expect any important benefit.



(a) Natural VS Unnatural code locations



(b) Natural VS Unnatural mutants



(c) Big VS Small impact (entropy differences introduced by mutants)

Figure 2: Identifying fault revealing mutants. Natural VS Unnatural program locations, Natural VS Unnatural mutant files and Big VS Small mutants’ impact. The x-axis records ratios of the top ranked selected mutants, while the y-axis records the fault revelation ability of the two selected strategies, i.e., fault revelation of natural and unnatural mutants. Higher values indicate higher fault revelation.

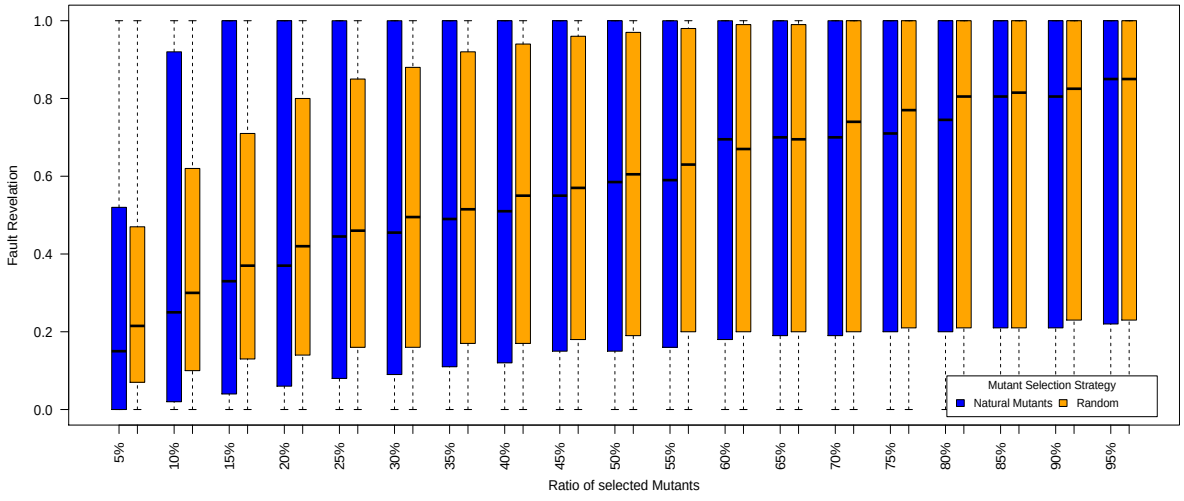


Figure 3: Fault Revelation of naturalness and random mutant selection. The x-axis records ratios of selected mutants and the y-axis records the fault revelation for every fault considered.

6 DISCUSSION

6.1 Visualizing Naturalness and Fault Revelation

To further investigate the relation between naturalness and fault revelation we visualize our data (with the hope to see some general trends that might not be captured by our analysis). Figure 4 plots the naturalness (naturalness of the original file minus the naturalness of the mutant file) and fault revelation probabilities for every mutant we consider. In the figure we observe that there is no pattern that we can exploit.

Mutants with high fault revelation (points on the x-axis with values above 0.5) are spread across all the spectrum of naturalness values. Mutants with low or no fault revelation (close to 0 value on the x-axis) have the most extreme negative values. Nevertheless, the visualization helps us demonstrate the absence of any relation between the examined variables.

6.2 Additional Attempts with Negative Results

Our results are in a sense negative (the expected benefit was not reached). However, this could be attributed to a number of parameters that were not considered. To account for some of them, we repeated our experiment (without any success) by using different parameters. Thus, we also used a different way to tokenize our programs using the program Abstract Syntax Tree (and compute naturalness), we composed models by considering n-values upto 10, we composed models by considering a much larger training corpus, i.e., we trained using 20 (related) Java programs (from Apache) and we measured the number of tests and equivalent mutants required (by the naturalness-based and random mutant selection) to reach the same level of fault revelation. All these attempts yielded quite similar results and overall we found no significant differences between naturalness-based and random mutant selection.

6.3 Threats to Validity

The generalisability of the results is a common threat to the external validity of every experimental study. To mitigate this threat, we used real-world projects with real faults.

A potential threat affecting the internal validity of our study stems from the sets of mutants and test suites that we used. We used state-of-the-art mutation testing tools [39] supporting all the mainstream mutation operators [23]. To compose the test pools, we used multiple test suites that were generated by state-of-the-art test generation tools, i.e., Randoop [36] and EvoSuite [13]. Although it is possible that different tests may lead to different results, the practice we employed reflects what current test case generation research has to offer in large-scale experiments.

Another threat originates from the computation of naturalness as we used three external tools (the two tokenizers and a language models toolkit). Thus potential bugs or errors in the use of those tools might impact the reported results. Regarding the tokenizers, we used well known and reliable ones.

We used Java Parser, which is used by more than 50 libraries and 100 projects on Github as well as some companies and is as well regularly updated. Java Parser is also well documented and provide handful examples preventing any misuse of the tools.

Regarding Kylm, the project is relatively old and not well documented but is still regularly used as comparison for new approaches like in the work of Pickhardt et al. [42], which shows that the results provided by the tool are still considered as relevant by the NLP community. To reduce this threat, we carefully analysed and tested the tool. Similarly, the use of 4-gram and of Modified Kneser Ney smoothing may have an impact on our results. We chose these options as suggested by our previous work [18].

Threats that affect the construct validity of our study concerns the metrics we used. To evaluate mutant ranking we used fault-revealing mutants and fault revelation probabilities approximated by multiple test executions. We deem this metric appropriate since fault revelation forms the purpose of testing.

7 RELATED WORK

Mutation testing is a well-studied, fault-seeding technique with a rich background both in theoretical and practical advances. A summary of these advances can be found in the recent survey of Papadakis et al. [39] which summarises the advances in the area between 2007–2016, complementing previous surveys [17, 35].

The quality of mutants and how to generate "good" mutants have concerned researchers for many years. This problem has many facets. First, the question "what changes should be applied to the program under test" can be posed. This is directly related to the mutation operators that one should use. Although mutation's research has expanded the available mutation operators to handle multiple and diverse artefacts, ranging from mobile applications [11, 27] to models [4, 12], and includes specialised sets of operators, e.g., energy- [16], security-[28] and memory-related [47] ones, it is not clear what constitutes a "good change" for mutant creation. Ultimately, a "good" mutant will be a fault-revealing one (for testing purposes), i.e., it will be killed by test cases that reveal underlying faults in the program under test [7].

Offutt et al. [33] introduced a theoretical model of the "size" of program faults which makes the separation between its syntactic and semantic characteristics. The syntactic size of a fault is related to the source code of the program under test and how it differs from its correct counterpart and the semantic size to the divergence between the program under test and its specification due to the presence of the fault. Thus, the authors suggest that mutants having a small semantic size are more valuable to testing.

Semantic mutation testing has been proposed as a way to generate mutants that affect the semantics of the language of the artefact under test rather than the syntax [9, 10]. Thus, the semantic mutants simulate a different category of faults than the traditional ones and are more useful in several scenarios.

Sridharan and Namin [44] attempt to rank mutants by focusing on mutation operators that are likely to generate mutants that will not be killed by a specific test suite. Their approach is based on a probabilistic, Bayesian model which analyses a small portion of the generated mutants and the available test suite to rank the whole set of mutants. In a similar vein, Namin et al. [31] introduced MuRanker, an approach that predicts the difficulty and complexity of mutants and ranks them accordingly. This approach is based on distance functions which take into account differences among the control flow graph, the Jimple representation and coverage data

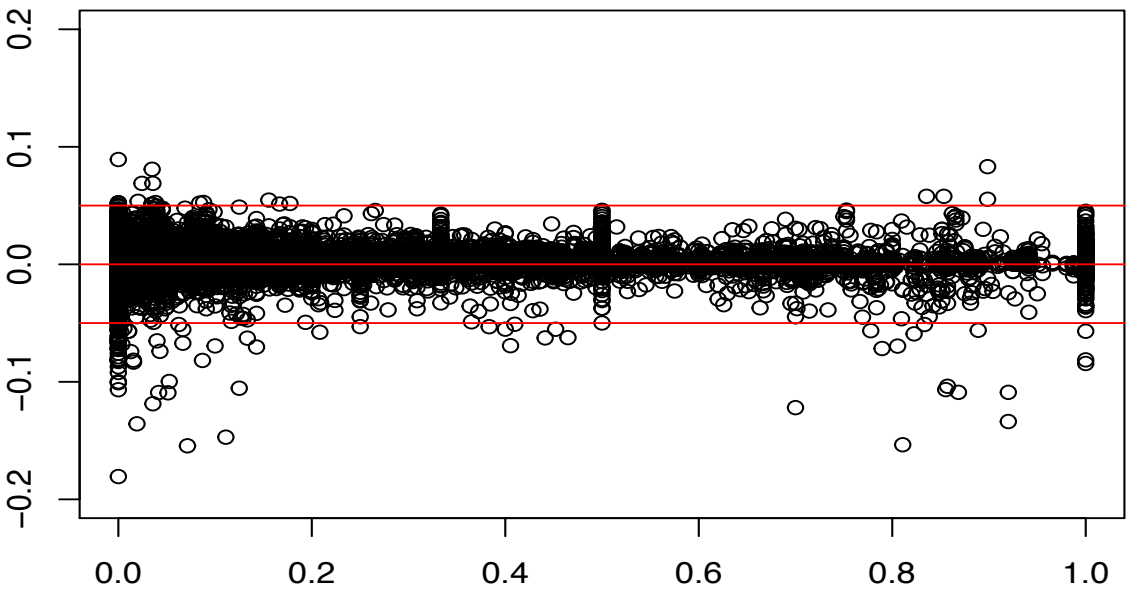


Figure 4: Fault Revelation probabilities and naturalness. The x-axis records fault revelation probability of each mutant (measured by counting the number of tests that kill the mutant and at the same time expose a real fault) and the y-axis records the naturalness value of every mutant.

between the mutants and the original program. The basic difference between our approach and the previous ones is that they depend on the available test suite whereas our approach leverage mutants' naturalness and is applied statically.

Other studies attempt to create mutation operators that resemble real faults by analysing previous faults that developers have made. Brown et al. [5] mine fault-fixing commits from the version control history of projects and extract fault-fixing patterns. Based on these patterns, they propose new mutation operators that reverse the patterns, thus, creating faulty program versions (mutants). Linares-Vásquez et al. [27] created a taxonomy of faults found in Android applications and propose a new set of Android mutation operators based on these patterns. However, the utility of these techniques have not yet been evaluated wrt to their ability to reveal faults.

8 CONCLUSION

Code forms a human artefact and as such it tends to follow patterns and norms that are repeatable and similar to those found in text and generally natural language. In the context of fault-based testing, what does it means for an artificial fault to follow the patterns and norms of the programmers? We investigated this question and found no link between the naturalness of mutants and the semantic alterations of real faults. We investigate this statement using statistical language models and provide evidence that there is no link between naturalness and the fault revelation utility of mutants. We also demonstrate that the naturalness-based mutant selection performs similar (slightly worse) than the random mutant selection.

We believe that our results are of interest to the software testing community since they confute a natural intuition. We show that the fault revealing utility of mutants is independent of their naturalness, which is in contrast to the findings (of previous research) related to the strong link between error proneness and naturalness. Our findings suggest that mutants (and their locations) coupled with faults are both natural and unnatural and that naturalness-based mutant selection strategies are not better than the random mutant selection.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 281–293. https://doi.org/10.1145/2635868.2635883

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *CoRR* abs/1709.06182 (2017). http://arxiv.org/abs/1709.06182

[3] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on* 32, 8 (2006), 608–624. https://doi.org/10.1109/TSE.2006.83

[4] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa. 2011. Mutation Testing of "Go-Back" Functions Based on Pushdown Automata. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 249–258. https://doi.org/10.1109/ICST.2011.30

[5] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-caught Mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/3106237.3106280

[6] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. 2018. Selecting Fault Revealing Mutants. *CoRR* abs/1803.07901 (2018). arXiv:1803.07901 http://arxiv.org/abs/1803.07901

[7] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault

Revelation That Avoids the Unreliable Clean Program Assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 597–608. https://doi.org/10.1109/ICSE.2017.61

[8] Stanley F. Chen and Joshua Goodman. 1999. An Empirical Study of Smoothing Techniques for Language Modeling. *Comput. Speech Lang.* 13, 4 (Oct. 1999), 359–394. https://doi.org/10.1006/csla.1999.0128

[9] J. A. Clark, H. Dan, and R. M. Hierons. 2010. Semantic Mutation Testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 100–109. https://doi.org/10.1109/ICSTW.2010.8

[10] H. Dan and R. M. Hierons. 2012. SMT-C: A Semantic Mutation Testing Tools for C. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 654–663. https://doi.org/10.1109/ICST.2012.155

[11] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. 2015. Towards mutation analysis of Android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–10. https://doi.org/10.1109/ICSTW.2015.7107450

[12] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 655–666. https://doi.org/10.1145/2884781.2884821

[13] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 416–419. https://doi.org/10.1145/2025113.2025179

[14] Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/1882291.1882315

[15] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[16] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: An Energy-aware Mutation Testing Framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 208–219. https://doi.org/10.1145/3106237.3106244

[17] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[18] Matthieu Jimenez, Maxime Cordy, Yves Le Traon, and Mike Papadakis. 2018. On the Impact of Tokenizer and Parameters on N-Gram Based Code Analysis. In *34th International Conference on Conference on Software Maintenance and Evolution, ICSME 2018, September 23 - 29, 2018, Madrid, Spain*.

[19] Matthieu Jimenez, Maxime Cordy, Yves Le Traon, and Mike Papadakis. 2018. TUNA: TUning Naturalness-based Analysis. In *34th International Conference on Conference on Software Maintenance and Evolution, ICSME 2018, September 23 - 29, 2018, Madrid, Spain*.

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 437–440. https://doi.org/10.1145/2610384.2628055

[21] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. 612–615. https://doi.org/10.1109/ASE.2011.6100138

[22] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Trans. Software Eng.* 44, 4 (2018), 308–333. https://doi.org/10.1109/TSE.2017.2684805

[23] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering* 23, 4 (2018), 2426–2463. https://doi.org/10.1007/s10664-017-9582-5

[24] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (May 1984), 97–111.

[25] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 571–582. https://doi.org/10.1145/2950290.2950322

[26] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. 2017. Assessing and Improving the Mutation Testing Practice of PIT. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 430–435. https://doi.org/10.1109/ICST.2017.47

[27] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/3106237.3106275

[28] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans. 2017. Towards Security-Aware Mutation Testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 97–102. https://doi.org/10.1109/ICSTW.2017.24

[29] Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

[30] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. 2018. Time to clean your test objectives. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 456–467. https://doi.org/10.1145/3180155.3180191

[31] Akbar Siami Namin, Xiaozhen Xue, Omar Rosas, and Pankaj Sharma. 2015. MuRanker: a mutant ranking tool. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 572–604. https://doi.org/10.1002/stvr.1542

[32] Graham Neubig. 2017. Kyoto Language Modeling Toolkit. (2017). https://github.com/neubig/kylm

[33] A. Jefferson Offutt and J. Huffman Hayes. 1996. A Semantic Model of Program Faults. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96)*. ACM, New York, NY, USA, 195–200. https://doi.org/10.1145/229000.226317

[34] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118. https://doi.org/10.1145/227607.227610

[35] A. J. Offutt and R. H. Untch. 2001. Mutation 2000: Uniting the Orthogonal. In *Mutation Testing for the New Century*, W.Eric Wong (Ed.). The Springer International Series on Advances in Database Systems, Vol. 24. Springer US, 34–44. https://doi.org/10.1007/978-1-4757-5939-6_7

[36] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 815–816. https://doi.org/10.1145/1297846.1297902

[37] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. 2018. Mutant Quality Indicators. In *13th International Workshop on Mutation Analysis (MUTATION'18)*.

[38] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the Validity of Mutation-based Test Assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 354–365. https://doi.org/10.1145/2931037.2931040

[39] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2018. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* (2018). https://doi.org/10.1016/bs.adcom.2018.03.015

[40] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 537–548. https://doi.org/10.1145/3180155.3180183

[41] Java Parser. 2017. Java Parser Github. (2017). https://github.com/javaparser/javaparser

[42] Rene Pickhardt, Thomas Gottron, Steffen Staab, Paul Georg Wagner, Till Speicher, and Typology Gbr. 2014. A generalized language model as the combination of skipped n-grams and modified Kneser-Ney smoothing. In *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.

[43] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[44] Mohan Sridharan and Akbar Siami Namin. 2010. Prioritizing Mutation Operators Based on Importance Sampling. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. 378–387. https://doi.org/10.1109/ISSRE.2010.16

[45] 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 (FSE 2014)*. ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/2635868.2635875

[46] A. Vargha and H. D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Jrnl. Educ. Behav. Stat.* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101

[47] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. 2017. Memory mutation testing. *Information and Software Technology* 81, Supplement C (2017), 97 – 111. https://doi.org/10.1016/j.infsof.2016.03.002