# A new word embedding approach to evaluate potential fixes for automated program repair

Leonardo Afonso Amorim, Mateus F. Freitas, Altino Dantas, Eduardo F. de Souza,
Celso G. Camilo-Junior and Wellington S. Martins
Instituto de Informática (INF), Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Câmpus Samambaia, Goiânia, Goiás, Brazil
Emails: leonardoafonso@gmail.com, mateusfreitas@inf.ufg.br, altinodantas@hotmail.com, eduardosouza@inf.ufg.br
celso@inf.ufg.br and wellington@inf.ufg.br

***Abstract*—Debugging is frequently a manual and costly task. Some work recently presented automated program repair methods aiming to reduce debugging time. Despite different approaches, the process of evaluating source code patches (potential fixes) is crucial for most of them, e.g., generate-and-validate systems. The evaluation is a complex task given that patches with different syntaxes might share the same semantics, behaving equally for typically limited specifications. Hence, many approaches fail to better explore the search space of patches, leading them to not reach the patch that fixes the bug. Some research points that a buggy code is more entropic, i.e., less natural than its fixed version. So, this work proposes applying Word2Vec, a word embedding model, to improve the repair evaluation process based on the naturalness obtained from a corpus of known fixes. Word2Vec captures co-occurrence relationships between words in a given context and then predicts the contextual words of a given word. This technique has been applied to deal with richer semantic relationships in a text. Word2Vec evaluates patches according to distances of document vectors and a softmax output layer. We analyze the performance of our proposal with mutated patches created from correct source codes and we simulate potential fixes generated by automated program repair approaches. Thus, the main contribution of this paper is a new method to evaluate patches used in automated program repair methods. The results show that Word2vec-based metrics are capable of analyzing source code naturalness and be used to evaluate source code patches.**

***Index Terms*—Automated Program Repair, Naturalness of software, Fitness Evaluation, Word embedding, Word2vec.**

## I. Introduction

In the last decades, software has gained increasingly more importance in several activities of our daily lives. Common tasks such as taking a taxi, as well as more complex ones, such as air traffic control, are widely supported by software, be it simple or sophisticated. The maintenance of these software becomes indispensable to keep our society going on, while avoiding financial losses or fatal accidents, for instance. Meanwhile, according to [1], software maintenance is typically a costly activity, in which fixing buggy programs accounts for approximately 21% of all resources in the maintenance phase. Besides, [2] argues that repairing a single failure may cost up to 200 working days and they amounted 1.7T USD in losses from software failures in 2017 [3].

In this chaotic scenario, various approaches have been developed aiming at automatizing the software repair process and, consequently, reducing manual efforts of software maintenance. These works are leveraging an area called *Automated Program Repair* (APR) by applying several computational techniques with promising results. To name a few examples of APR techniques: [4], [5] applied Genetic Programming concepts; [6] and [7] used both symbolic execution and constraint solvers to produce software corrections; finally, [8] and [9] proposed different learning methods to fix incorrect fragments of code. In summary, they all exploit the space of solutions by automatically generating variants from an original buggy program and then evaluating them with some quality measure.

Despite recent findings, real-world APR applicability still continues to be a complex task, mainly because some of its essential steps are not trivial. For instance, evaluating variants (i.e. patches) generated by APR methods is a challenging task since different source codes (syntax) may share the same semantics. There is also the complexity of comparing partial results. The most common way to evaluate a patch is to rely on test cases or formal specifications. Typically, given an automatically generated program variant and a test suite, a possible evaluation is a weighted sum of how many test cases this variant passes [5]. Unfortunately, this kind of evaluation leads to plateaus, where patches with distinct source code have the same evaluation score as they pass in the same number of test cases, regardless of whether they are different test cases or not. Therefore, test case-driven methods are typically insufficient to establish a distinction over variants that pass the same number of test cases.

Assuming source codes are regular and predictable [10], we speculate that software naturalness can help the evaluation process of potential fixes for buggy programs. Some researchers have captured this naturalness of software through statistical models [11] and then used them for named entity recognition [12], coding standards checkers and suggesting accurate method and class names [13]. Due to good results of previous works using embedding words and sequence-to-sequence methods, we believe that these approaches, including Word2vec, can also be applied to modeling fix naturalness and then improve the evaluation process of the variants.

To the best of our knowledge, this is the first work which applies word embeddings in software source code to generate

978-1-5090-6014-6/18/$31.00 ©2018 IEEE

new metrics for evaluating potential fixes. To investigate this approach, we consider a corpus of fixes from six different programs (checksum, digits, grade, median, smallest, and syllables) from the IntroClass Benchmark [14]. Overall, our results show that Word2vec is a promising method to recognize the naturalness pattern of source code and, therefore, the derived metrics (Softmax output layer and distance between document vectors) can help the evaluation process of variants. Our experiments are done with C projects, but they can be made for any programming language. For this, we can train our system with a corpus formed by fixed codes (found in repositories of open source projects).

Our main contributions are:

- A method to statically evaluate variants generated by automated program repair methods;
- A methodology based on embedded words and sequence-to-sequence models to evaluate the naturalness of a potential fix;
- An experimental study using standard program repair benchmarks.

This paper is organized as follows. Section II provides background to basic concepts. Section III presents our proposed methodology based on Word2Vec. Section IV presents the experiment design. Section V presents the results. Section VI describes related works. Section VII concludes the paper and presents future works.

## II. Background

Word embedding is the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP), where words or phrases from the vocabulary are mapped to vectors of real numbers in a low-dimensional space relative to the vocabulary size ("continuous space"). Methods to generate this mapping include neural networks [15], dimensionality reduction on the word co-occurrence matrix, probabilistic models, and explicit representation regarding the context in which words appear [16]. The present work focuses on Word2Vec [17], a predictive method based on a two-layer neural net that can process large textual datasets and produces corresponding word embeddings.

### A. Word2Vec

Word2Vec learns continuous word embeddings from plain text in an entirely unsupervised way. The model assumes the Distributional Hypothesis [18], which states that words that appear in the same contexts tend to share semantic meaning. This allows us to estimate the probability of two words occurring close to each other.

With Word2Vec, a neural network is trained with streams of $n$-grams of words so as to predict the $n$-th word, given words $[1, \ldots, n-1]$ or the other way around. The output is a matrix of word vectors or context vectors, respectively. Since the neural network used has a simple linear hidden layer, the intensity of correlation between words is directly measured by the inner product between word embeddings.

The two neural network models used by Word2Vec are known as continuous bag-of-words (CBOW) and Skip-gram. Rather than predicting a word conditioned on its predecessor, as in a traditional bi-gram language model, the CBOW model [17] predicts the current word based on the context, while the Skip-gram model [17] predicts the neighborhood words given the current word.

### B. Word2Vec Softmax output layer

In mathematics, the softmax function (normalized exponential function) is a generalization of the logistic function. The softmax function is used to obtain a well-defined probabilistic (multinomial) distribution among words. This function limits the range of the normalized data to values between 0 and 1. These values can be considered as probabilities given an input in the artificial neural network. The softmax function is often used in the final layer of artificial neural networks, which are applied to classification problems.

The weights between the input layer and the output layer can be represented by a $V \times N$ matrix $\mathbf{W}$. Where V is vocabulary size and N is the hidden layer size. Each row of $\mathbf{W}$ is the $N$-dimensional vector representation $\mathbf{v}_{stm}$ of the associated token (word)[1] of the input layer. Given a context (token), assuming $x_b = 1$ and $x_{b'} = 0$ for $b' \neq b$, where the $b$ value represents a bit position in a word embedding vector $\mathbf{x}$[2], then:

$$\mathbf{h} = \mathbf{x}^T \mathbf{W} := \mathbf{v}_{stm_i} \tag{1}$$

In other words, this operation is essentially copying the $b$-th row of $\mathbf{W}$ to $\mathbf{h}$. $\mathbf{v}_{stmi}$ is the vector representation of the input token $stm_i$. From the hidden layer to the output layer, there is a different weight matrix $\mathbf{W'} = \text{stm'}_{ij}$, which is a $N \times V$ matrix. Using these weights, we can compute a score $u_j$ for each token in the vocabulary:

$$u_j = \mathbf{v'}_{stm_j} . \mathbf{h} \tag{2}$$

where $\mathbf{v'}_{stm_j}$ is the $j$-th column of the matrix $\mathbf{W'}$. Then we can use softmax to obtain the posterior distribution of tokens, which is a multinomial distribution [19].

$$p(stm_j \mid stm_i) = y_j = \frac{exp(u_j)}{\sum_{j'=1}^{N} exp(u_{j'})} \tag{3}$$

where $y_j$ is the output of the $j$-th node in the output layer. Substituting (1) and (2) into (3), we obtain

$$p(stm_j \mid stm_i) = \frac{exp({v'_{stm_o}}^T v_{stm_i})}{\sum_{j'=1}^{N} exp({v'_{stm'_j}}^T v_{stm_i})} \tag{4}$$

Note that $\mathbf{v}_{stm}$ and $\mathbf{v'}_{stm}$ are two representations of the token $stm$. $\mathbf{v}_{stm}$ comes from rows of $\mathbf{W}$, which is the input $\rightarrow$ hidden weight matrix, and $\mathbf{v'}_{stm}$ comes from columns of $\mathbf{W'}$, which is the hidden $\rightarrow$ output matrix. In subsequent analysis, we call

[1] In this work, a word/token is considered as a line of code.

[2] In other words, only one element of $\{x_1, \ldots, x_V\}$ is 1, and all other elements are 0.

$\mathbf{v}_{stm}$ as the "input vector", and $\mathbf{v'}_{stm}$ as the "output vector" of the token *stm*.

The next step is to derive the weight update equation for this model. The training objective is to maximize (4), the conditional probability of observing the actual output token *stm* (denote its index in the output layer as $j^*$) given the input context token $stm_i$ with regard to the weights.

$$maxp(stm_j \mid stm_i) = u_j^* - log \sum_{j'=1}^{N} exp(u_{j'}) := -E \quad (5)$$

where $E = log\ p(stm_j \mid stm_i)$ is our loss function (we want to minimize $E$), and $j^*$ is the index of the actual output token in the output layer. The next step is to derive the update equation of the weights between hidden and output layers. Take the derivative of $E$ with regard to $j$-th node's net input $u_j$, we obtain

$$\frac{\varphi E}{\varphi u_j} = y_j - t_j := e_j \quad (6)$$

where $t_j$ = 1 (j = $j^*$), i.e., $t_j$ will only be 1 when the $j$-th node is the actual output token, otherwise $t_j$ = 0. Note that this derivation is the prediction error $e_j$ of the output layer. Next we take the derivative on $stm_{ij}$ to obtain the gradient on the hidden → output weights.

$$\frac{\varphi E}{\varphi stm'_{ij}} = \frac{\varphi E}{\varphi u_j} . \frac{u_j}{\varphi stm'_{ij}} = e_j . h_i \quad (7)$$

Therefore, using stochastic gradient descent, we obtain the weight updating equation for hidden → output weights:

$$\mathbf{v'}_{stm_j}{}^{(new)} = \mathbf{v'}_{stm_j}{}^{(old)} - \eta \,.\, e_j \,.\, \mathbf{h} \mid \ j = 1, 2, ..., N. \quad (8)$$

where $\eta > 0$ is the learning rate, $e_j$ = $y_i$ - $t_j$, and $h_i$ is the $i$-th node in the hidden layer; **stm'**$_j$ is the output vector of $stm_j$.

Having obtained the update equations for **W'**, we can now move on to **W**. We take the derivative of $E$ on the output of the hidden layer, obtaining

$$\frac{\varphi E}{\varphi h_i} = \sum_{j=1}^{V} \frac{\varphi E}{\varphi u_j} \cdot \frac{u_j}{\varphi h_{ij}} = \sum_{j=1}^{V} e_j \,.\, stm'_{ij} := EH_i \quad (9)$$

where $h_i$ is the output of the $i$-th node of the hidden layer; $u_j$ is defined in (2), the net input of the $j$-th node in the output layer; and $e_j$ = $y_j$ - $t_j$ is the prediction error of the $j$-th token in the output layer. EH, a N-dim vector, is the sum of the output vectors of all tokens in the vocabulary, weighted by their prediction error.

Next we should take derivative of $E$ on **W**. The hidden layer performs a linear computation on the values from the input layer. Expanding the vector notation in (1) we get

$$h_i = \sum_{k=1}^{V} x_k \,.\, stm_{ki} \quad (10)$$

Now we can take the derivative of $E$ with regard to **W**, obtaining

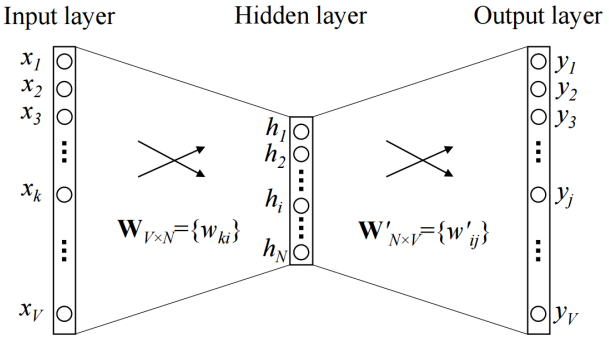


Fig. 1. A CBOW model adopted in this work [19]

$$\frac{\varphi E}{\varphi stm_{ij}} = \frac{\varphi E}{\varphi h_i} . \frac{h_i}{\varphi stm_{ki}} = EH_i . x_k \quad (11)$$

from which we obtain $V \times N$ matrix. Since only one component of **x** is non-zero, only row of $\frac{\varphi E}{\varphi \mathbf{W}}$ is non-zero, and the value of that row is EH, a $N$-dim vector. We obtain the update equation of **W** as

$$\mathbf{v}_{stm_j}{}^{(new)} = \mathbf{v}_{stm_j}{}^{(old)} - \eta \,.\, EH \quad (12)$$

where $\mathbf{v}_{stmi}$ is a row of **W**, the "input vector" of the only context token, and is the only row of **W** whose derivative is non-zero.

After the Word2Vec training, the softmax layer stores the final probability values. These values can be accessed by passing the input and output tokens. The artificial neural network training process shown in this section considers all vocabulary words (tokens) in the calculation performed between the hidden and output layers.

## III. Approach

This section presents the proposed approach to evaluate variants generated by APR methods.

Our approach has three phases (Figure 2). The first phase, called Corpus Selection, consists in selecting the fix corpus, which is a reference for determining the semantic programming patterns of a developer community. The quality of the selected fixes has a critical impact on the performance of the proposal since the learning method uses this data to recognize the pattern.

The second phase, called Tokenization, defines how the source codes from the corpus are split and are used to create the vocabulary. The input to this phase is a corpus of fixes, syntax formatted and standardized. The output is a vocabulary with all recognizable tokens (classes).

The third phase, called Learning, builds a model able to predict the next token (target) given its context (set of tokens). For each pair (context token and target token) in a training fix code, Word2vec examines lots of pairs to effectively learn that "token X and Y often appear together" and "token X and W do not". The algorithm output is a matrix with word vectors (a trained model).

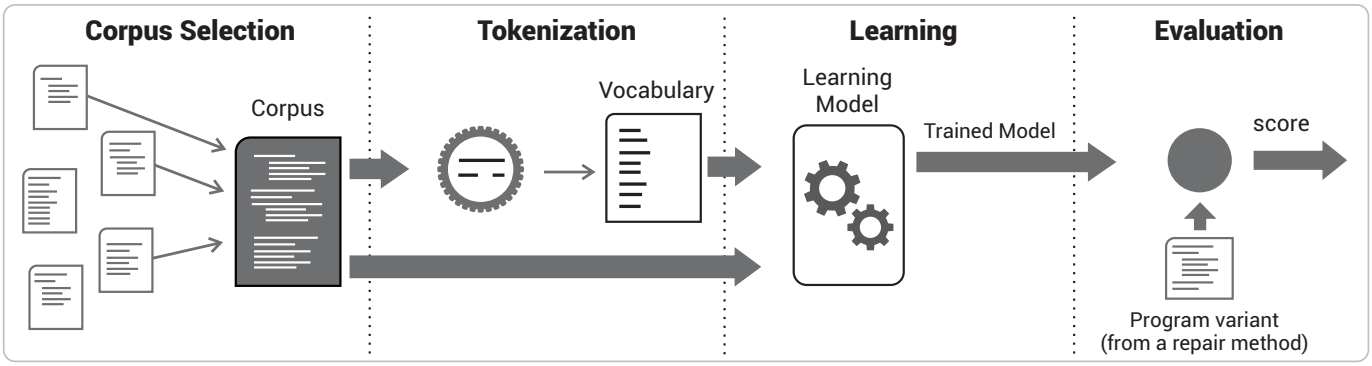


Fig. 2. Proposal's flowchart

The fourth phase, called Evaluation, uses the embedded words or softmax layer output to calculate the score for each variant. After that, the highest score variant is chosen as the RefCode (Reference Code).

To perform the last phase, we adapted Word2Vec algorithm to obtain the probabilities in the output softmax layer after the artificial neural network training finish. Figure 1 shows the network with one-word or one-token context, the architecture used in this work. In our setting, the vocabulary size is $V$, and the hidden layer size is $N$. The nodes on adjacent layers are fully connected. The input vector is the one-hot encoded vector, which means that for a given output context word, only one node of $\{x_1,\ldots,x_V\}$ is 1, and all other nodes are 0 [19].

## IV. Experiments

### A. Setup

The experiments were conducted on a machine running CentOS 7.2 64-bits, with an Intel® Xeon®E5-2620 2GHz and 16GB RAM. Our proposal is as follows:

- **Phase 1 (Corpus Selection)**: we use the programs checksum, digits, grade, median, smallest and syllables from IntroClass[3] to validate our proposal. We create a corpus composed of all IntroClass benchmark codes (fixes made by humans). The idea is to capture the naturalness of code from the programming community in general.
- **Phase 2 (Tokenization):** we split each fixed code in a corpus line-by-line. Thus, a token is an entire line because we speculate that the line has enough semantic.
- **Phase 3 (Learning):** we use a token as context and the next one as a target. Then, each pair (context, target_token) in a training fixed code is a sample to train the model. We use the Word2vec toolkit to fit the model's parameters to recognize the sequence. Each token has a representative vector. The vector size produced in most WordVec applications is between 100 and 500 units (neurons). These applications usually have vocabularies with hundreds of thousands of words. Since the vocabulary used in programming is usually small, we have chosen to reduce the word vector size to 50. The chosen Word2Vec architecture was CBOW because this model predicts which token is more likely to happen given another context token. Also, CBOW is faster to train and it has better accuracy for more frequent words than Skip-gram architecture. In addition, Skip-gram model works best for small databases and when we need to represent rare words or phrases[4]. The size of the context window used in model training was one (1). This size captures pieces of codes that have a more recurring neighborhood. Therefore, we can score codes that have more naturalness. This phenomenon was observed empirically in several tests performed with many windows.
- **Phase 4 (Evaluation)** we use two metrics based on Word2Vec softmax output and embedded words distances. These are explained in section IV-B.

### B. Metrics

We use two metrics *Prob* and *Dist* based on the proposal's Phase 4 outputs. The metric *Prob* uses the pairwise probability obtained from the last layer of Word2vec. Thus, for each variant evaluated, the metric *Prob* is the average of the pairwise probabilities (context, target_token) obtained from it. Let $M$ denote the number of tokens in a fix, the metric *Prob* is calculated by (13).

$$Prob(variant) = \frac{\sum_{i=1}^{M-1} probability(token_i, token_{i+1})}{M} \quad (13)$$

The metric *Dist* uses the vectors, semantically meaningful representations, obtained from the embedded word layer of Word2vec. The metric *Dist* in this work is the Word Mover's Distance (WMD). The WMD distance measures the dissimilarity between two documents as the minimum amount of distance that the embedded words of one document need to "travel" to reach the embedded words of another document [20]. Since this metric evaluates the dissimilarity, we choose the fixed code in the corpus with the highest *Prob* as the reference for comparison with the variant under evaluation.

[3]https://github.com/ProgramRepair/IntroClass

[4]https://code.google.com/archive/p/word2vec

### C. Scenarios

We evaluated the proposal on three scenarios. For each scenario, we simulate variants applying different mutation operators. We applied the operator one to ten times in a reference code (highest *Prob* on corpus) to analyze the impact of naturalness reduction on evaluation. Thus, we analyzed whether the metrics are sensitive to changes.

The first scenario applies the delete operator to generate variants; the second one applies the insert operator; and the third applies the swap operator.

All operators were applied on different levels, and the result for each level is the average of ten independent applications of the operator. Thus, the level one is the single random application of an operator, the second level is the random application of an operator two times and so on. Consequently, each level represents a reduction level of naturalness, since more perturbation and entropy in the code causes less naturalness.

Therefore, we aim to answer the following research questions with the scenarios aforementioned:

- **RQ1)** Is the metric *Prob* able to capture the loss of naturalness of variants?
- **RQ2)** Is the metric *Dist* able to capture the loss of naturalness of variants?

## V. Results

We present in this section the results of each program regarding metrics *Prob* and *Dist*.

Overall, as the perturbation level increases, we observe an increase in the distance and a decrease in the probability. This behavior can be more easily seen in Figures 3a, 4a and 5 for the metric *Prob*. For the metric *Dist* on Figures 3b and 4b. The metric *Dist* for the Swap operator is always 0 since the order of the tokens (line of code) does not affect it. Thus the variant is equal to the reference fix for this metric.

The results show that the variants' *Prob* decreased when lines of RefCode were deleted. In some cases, the probability score can increase when the level goes up. For example, the metric *Prob* increased at level 5 for checksum and digits, and at levels 3 and 4 for grade. This happens when a line of code that occurs less frequently in the corpus is removed. Thus, we note that it is possible for an incomplete variant to have a higher score than a fix, although this occurs rarely.

However, the metric *Dist* does not present exceptions. As the perturbation level increases, the metric *Dist* also increases or at least maintains equal to the previous level.

The insert operator causes the variant score to decrease as the level increases since equal tokens out of sequence are not expected. As for the delete operator, tokens from the training corpus of the model that have low frequency can be removed. This can cause the score to diminish with less intensity or even rise. Therefore, the insert operation had more linear behavior than the delete operation.

When the metric *Prob* results in a high score for a variant, we can use the metric *Dist* to see how far the variants are from the RefCode, as shown in the charts 3b and 4b. The increase

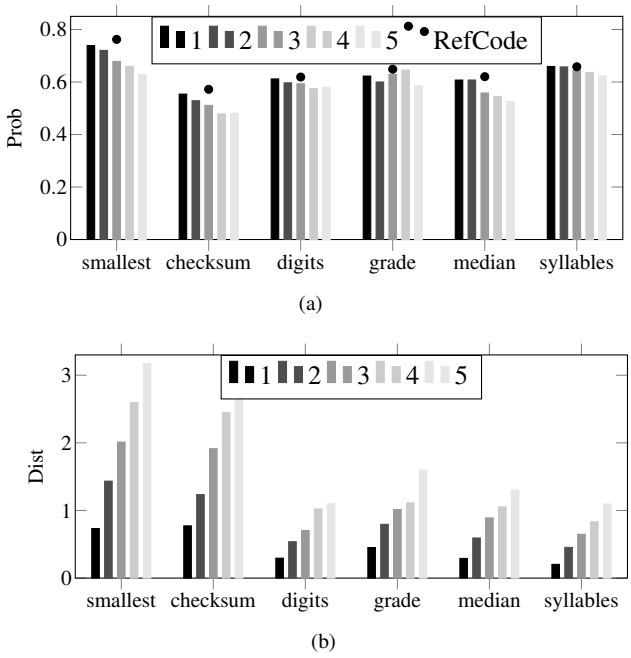


Fig. 3. Results after Delete operator: (a) Prob (b) Dist

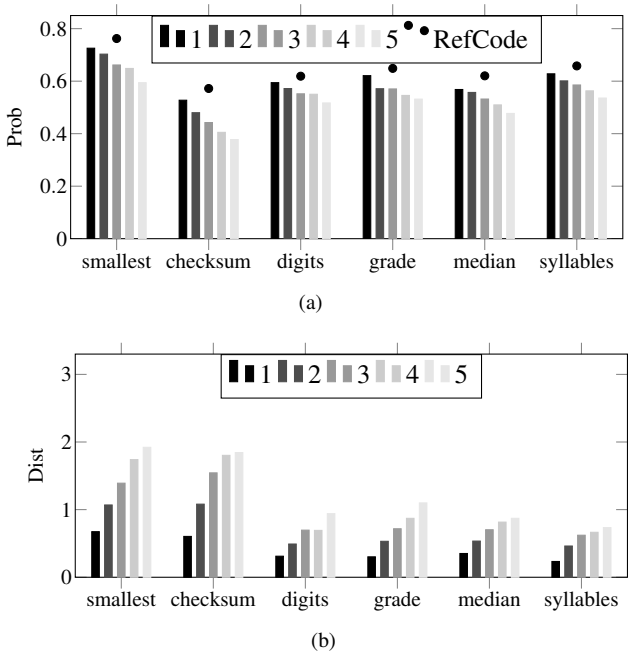


Fig. 4. Results after Insert operator: (a) Prob (b) Dist

in mutation level results in a higher distance from it in relation to the RefCode.

There were benchmark programs that did not behave as expected in terms of decreasing the metric *Prob* as the level of mutation increases. This is because there are an increased variety of fixes to the same problem (different developers). For instance, the smallest program has the fixes with a strong similar structure whereas the grade program is not.

Metrics based on test cases (baseline) have as a disability the generation of a lot of plateaus. Our metrics have a low

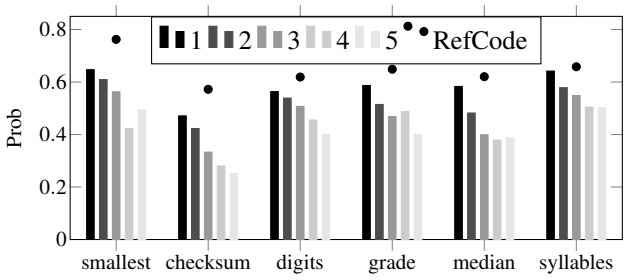


Fig. 5. Prob results after Swap operator

probability of generating plateaus. Exceptions occur rarely in the delete operation when the same line of the variant is removed in level 1 as can be seen in Table I for variants 9 and 10 in the smallest problem and for variants 8 and 9 in the checksum problem. Another exception is the swap operation for the metric *Dist*. Most problems in delete and insertion operations do not occur on a plateau. An example can be seen in the level 1 of the delete operator for the grade problem.

TABLE I
SCORES OF DIFFERENT VARIANTS FOR EACH PROGRAM GENERATED BY DELETE OPERATOR LEVEL 1

| | **smallest** | | **checksum** | | **grade** | |
|---|---|---|---|---|---|---|
| **V. ID** | **Prob** | **Dist** | **Prob** | **Dist** | **Prob** | **Dist** |
| 1 | 0.7162 | 0.7891 | 0.5169 | 0.7271 | 0.6065 | 0.443 |
| 2 | 0.7229 | 0.7013 | 0.5784 | 0.6353 | 0.5847 | 0.3387 |
| 3 | 0.7690 | 0.6584 | 0.5836 | 0.6489 | 0.6314 | 0.38 |
| 4 | 0.7632 | 0.7891 | 0.5307 | 1.0395 | 0.6244 | 0.3933 |
| 5 | 0.7142 | 0.6524 | 0.5504 | 0.8576 | 0.5713 | 0.5094 |
| 6 | 0.7632 | 0.6772 | 0.5817 | 0.6259 | 0.5847 | 0.3387 |
| 7 | 0.7311 | 0.8467 | 0.5178 | 0.6546 | 0.5943 | 0.8573 |
| 8 | 0.7690 | 0.7013 | 0.5712 | 0.9185 | 0.6637 | 0.4267 |
| 9 | 0.7196 | 0.729 | 0.5712 | 0.9185 | 0.6980 | 0.3818 |
| 10 | 0.7196 | 0.7843 | 0.5367 | 0.6967 | 0.6659 | 0.4525 |

For most cases, the metric *Prob* have good results, as can be seen on charts 3a, 4a and 5, capturing the loss of naturalness of variants. The metric *Dist* is able to capture the loss of naturalness of variants because it obtained linear behavior with the increase of the level of mutation for all problems.

In addition to these experiments, we run GenProg, an automated software repair method, to generate smallest problem fixes called variant A and variant B as can be seen in Figure 6. The purpose of this experiment is to verify whether *Prob* and *Dist* metrics can distinguish variants that have alike or equal fitness values measured only by test case coverage. The metric *Dist* shows how semantically far the candidate to fix is from a correct program that was selected from known fixes.

In a qualitative analysis of the codes, we notice the variant B is closer to the correct program for smallest since the variant A has the statement *return* within blocks of if-clauses. It is not common for this statement to be present within conditional blocks in the C language, thus, the metric *Prob* was able to capture that the co-occurrence between the neighboring statements of the statement *return (0)* is not common.

Whereas the *Prob* value for variant B is 0.74, indicating that it is more likely to be a fix, being each of its pair of subsequent lines more similar to the sequences in the correct code (Figure 6). Regarding the *Dist* values, the variant B presents a lower value than variant A. This is natural due to the later has a programming structure closer to the correct program.

Therefore, we can argue the proposal is able to deal with fitness plateaus, that is a huge problem for some automated software repair methods, like GenProg.

Thus, we can answer the research questions:

- **RQ1)** Yes, the metric *Prob* can capture the loss of naturalness of variants, since for most of the cases this metric decreased as the perturbation level increases. Finally, the metric helps to avoid the plateaus, since in only two cases, the generated variants obtained the same score.
- **RQ2)** Yes, the metric *Dist* is also able to capture the loss of naturalness of variants, since almost all cases this metric increased as the perturbation level increased. The exception was for the swap operator because the tokens remain the same in the variant after the operation.

## VI. RELATED WORK

Many works have focused on proposing and evolving techniques to address automated software repair. As we discussed before, almost all of these approaches are based on software specification, for example [21], [22], [23] or test cases [24], [25], [26], [27], [28], [29], [30], [31] and [32]. In other words, these techniques are vulnerable to partial specifications, pool test suites and time concerns, because typically it is necessary to run or, at least, simulate all tests for each variant under evaluation. Especially when evolutionary techniques are used, executing a large number of test cases may be infeasible or their absence may cause weakness in the fitness evaluation.

Our proposal performs a static analysis by applying the learned model, thus, it does not require to execute test case or even original program and variants for evaluating a patch.

Recently, some works proposed metrics or mechanisms to alleviate test cases dependence. In [33] is introduced a re-usability metric to help on repairing programs using fix ingredients from similar code extract from correct programs. To compound the metric, the proposal analyzes ASTs (Abstract Syntax Trees) and defines a level of similarity and difference between a buggy program and a fix candidate fragment of an external code. The authors argue that the similarity component is used to find fix ingredients whereas the difference component prevents picking up a code with the same bug under repairing. In opposition to our method, the metric re-usability is not able to rank a set of candidate patches, but only create a threshold through which it selects a set of promising variants.

The work proposed by [34] applies Genetic Programming to repair bugs, but it uses a model checking instead of a test suite to evaluate program variants. In summary, there is a module with a model checking that receives the candidate program states and measures a fitness level according to the number and type of the errors found while traversing all model paths. Despite the fitness, the evaluation does not depend on test

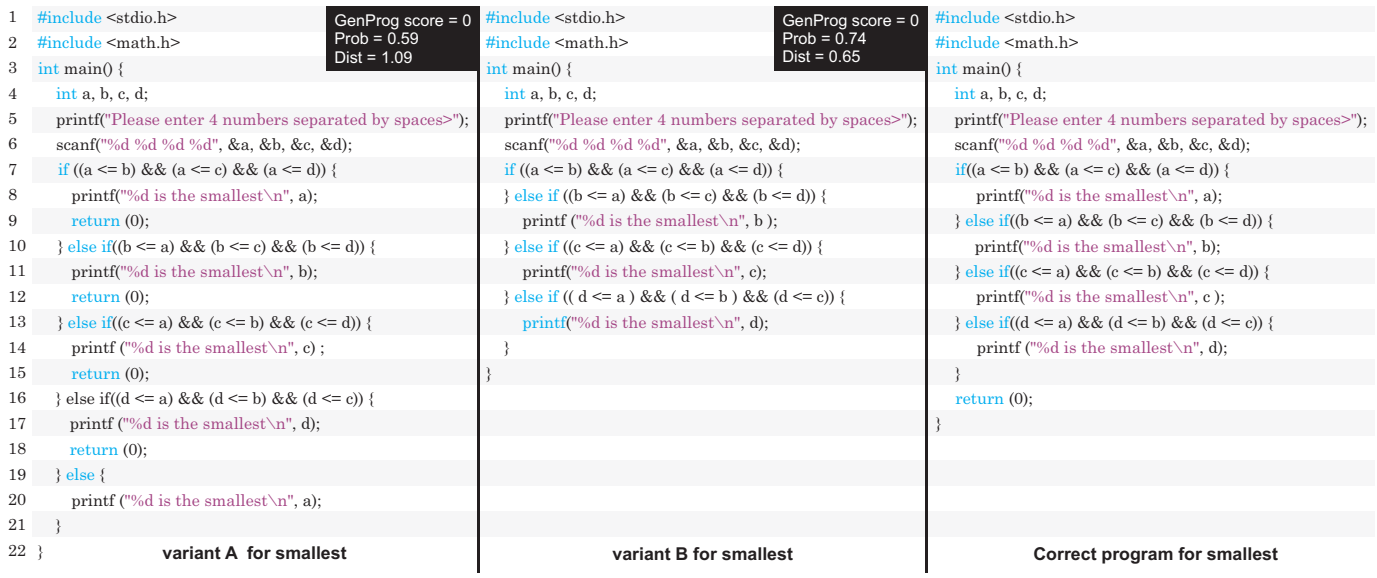


Fig. 6. Examples of two canditate variants obtained from GenProg and one correct code for smallest problem.

cases, the model checking requires a set of properties to be verified, for example, built-in properties or custom assertions. Meanwhile, our proposal does not require any predefined properties because our assumption is that they can be captured automatically by the learning model.

It is known that, in practice, a test Oracle may not be available or its usage is too expensive. Thus, [35] applied the concept of Metamorphic Test (MT) aiming to alleviate the test Oracle problem for generating program repair. MT verifies the relations among multiple test cases and their outputs instead of checking individual test cases correctness. Despite that, the proposal does not require an oracle to guide the search process, it is necessary to run a set of MT to evaluate a program variant.

Hence, one alternative for costly test suite-based evaluation approach is the static analysis of "naturalness". The "naturalness" of software was studied in [10], where the authors investigate the assumption that codes of programs are likely to be repetitive and predicted, as the natural languages. Using a common language model called n-grams, it was possible to capture regularity on the tokens in Java and C corpus of code. Based on the previous discovery, it was implemented an Eclipse plugin to improve the code completion engine of the Eclipse IDE. By considering a training corpus of Java projects the new plugin outperformed the native one especially for predicting little tokens. So the n-gram frequency is associated with the "naturalness".

Inspired by great results of language models and the embedding words for text mining, we speculated that these methods can also help the evaluation process in the field of automatic program repair. So we proposed our method based on Word2vec. The Word2vec has been used on many problems, for instance, it was used in [12] to support predictions on Name Entity Recognition problem. That is, given a word, it must be identified as a location, person or organization. The authors report interesting results by combining Word2vec and Linear Support Vector Classification algorithm on the task of classifying words from a corpus of newspaper articles.

Although some works try to evaluate variants with lower cost and greater effectiveness, we did not find any work that used sequence-to-sequence model or embedded words for this.

## VII. Conclusion

Assessing fixes quality generated by Automated Program Repair methods are challenging mainly due to the lack of compliance information, the cost of the task and the number of variables. Several techniques have been applied to evaluate the quality of a fix. The most used is based on test suite execution, especially by search-based program repair methods. But it is costly and imprecise since it generates plateaus, decreasing the search performance. Thus, we proposed a method based on word embeddings to analyze statically the generated variants as to its naturalness and consequently evaluate its quality. To validate our proposal, we used corpus of fixes from repositories of IntroClass Benchmark. We simulated the variants changing a RefCode in different ways. For each variant, two scores were assigned: *Prob* and *Dist*. Fixes with higher *Prob* and lower *Dist* are considered more natural.

Overall, our experiments show promising results, endorsing our hypothesis that embedded words and softmax output from Word2vec can be used as metrics of quality of variants.

Therefore, our main contribution is a method to evaluate statically the variants generated by automated program repair methods. Calculating the metrics for a variant may require more computation as several variants and larger codes need to be evaluated. So a parallel version of this algorithm can be implemented as a future work. Furthermore, the word vectors generated for the tokens can also be used to build variants that can repair bugs or help on it, once the model learned the token sequence.

## REFERENCES


[1] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 73–87. [Online]. Available: http://doi.acm.org/10.1145/336512.336534

[2] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.13

[3] Tricentis. (2017) Software fail watch: 5th edition. [Online]. Available: https://www.tricentis.com/software-fail-watch/

[4] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 2008, pp. 162–168.

[5] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337225

[6] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.60

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884807

[8] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837617

[9] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the 31ft AAAI Conference on Artificial Intelligence*, 2017.

[10] P. Devanbu, "On the naturalness of software," in *Proceedings of the 6th India Software Engineering Conference*, ser. ISEC '13. New York, NY, USA: ACM, 2013, pp. 61–61. [Online]. Available: http://doi.acm.org/10.1145/2442754.2442763

[11] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[12] S. K. Siencnik, "Adapting word2vec to named entity recognition," in *NODALIDA*, 2015.

[13] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[14] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015, http://dx.doi.org/10.1109/TSE.2015.2454513 DOI: 10.1109/TSE.2015.2454513.

[15] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053

[16] O. Levy, Y. Goldberg, and I. Ramat-Gan, "Linguistic regularities in sparse and explicit word representations," in *Proceedings of the Eighteenth Conference on Computational Natural Language Learning*. Ann Arbor, Michigan: Association for Computational Linguistics, June 2014, pp. 171–180. [Online]. Available: http://www.aclweb.org/anthology/W/W14/W14-1618

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1301.html#abs-1301-3781

[18] Z. Harris, "Distributional structure," *Word*, vol. 10, no. 23, pp. 146–162, 1954.

[19] X. Rong, "word2vec parameter learning explained," *CoRR*, vol. abs/1411.2738, 2014. [Online]. Available: http://arxiv.org/abs/1411.2738

[20] M. Kusner, Y. Sun, N. Kolkin, and K. Q. Weinberger, "From word embeddings to document distances," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, D. Blei and F. Bach, Eds. JMLR Workshop and Conference Proceedings, 2015, pp. 957–966. [Online]. Available: http://jmlr.org/proceedings/papers/v37/kusnerb15.pdf

[21] D. Gopinath, M. Z. Malik, and S. Khurshid, *Specification-Based Program Repair Using SAT*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 173–188. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19835-9_15

[22] B. Jobstmann, A. Griesmayer, and R. Bloem, *Program Repair as a Game*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 226–238. [Online]. Available: http://dx.doi.org/10.1007/11513988_23

[23] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 162–168.

[24] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494 – 3514, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1568494611000330

[25] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 947–954. [Online]. Available: http://doi.acm.org/10.1145/1569901.1570031

[26] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[27] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 448–458.

[28] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 772–781.

[29] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, May 2014.

[30] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254

[31] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 471–482.

[32] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 727–738. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950295

[33] T. Ji, L. Chen, X. Mao, and X. Yi, "Automated program repair by using similar code containing fix ingredients," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, June 2016, pp. 197–202.

[34] Z. Zojaji, B. T. Ladani, and A. Khalilian, "Automated program repair using genetic programming and model checking," *Applied Intelligence*, vol. 45, no. 4, pp. 1066–1088, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10489-016-0804-0

[35] M. Jiang, T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Ding, "A metamorphic testing approach for supporting program repair without the need for a test oracle," *Journal of Systems and Software*, vol. 126, pp. 127 – 140, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216300206