The Journal of Systems & Software 190 (2022) 111304

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Data Augmentation by Program Transformation☆

Shiwen Yu, Ting Wang, Ji Wang *

Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer Science and Technology, National University of Defense Technology, Changsha, China

ARTICLE INFO

Article history:
Received 22 June 2021
Received in revised form 11 March 2022
Accepted 13 March 2022
Available online 26 March 2022

Keywords:
Data Augmentation
Big Code
Program Transformation

ABSTRACT

Data Augmentation has been recognized as one of the main techniques to improve deep learning models' generalization ability. However, it has not been widely leveraged in big code tasks due to the essential difficulties of manipulating source code to generate new labeled data of high quality. In this paper, we propose a general data augmentation method based on program transformation. The idea is to extend big code datasets by a set of source-to-source transformation rules that preserve not only the semantics but also the syntactic naturalness of programs. Through controlled experiments, we demonstrated that semantic and syntax-naturalness preserving are the expected properties for a transformation rule to be effective in data augmentation. We designed 18 transformation rules that are proved semantic-preserving and tested syntax-naturalness-preserving. We also implemented and open-sourced a partial program transformation tool for Java based on the rules, named SPAT, whose effectiveness for data augmentation is validated in three big code tasks: method naming, code commenting, and code clone detection.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

With the advent of large repositories of source code and scalable deep learning methods, there has been an increasing interest in applying deep learning to solve software engineering tasks, forming the research field of big code. Previous studies have achieved promising results by getting high scores on limited test sets. However, there is often a lack of consideration for the generalization problem. We observed severe function failures of some big code deep learning models when they have to deal with data that is unseen in their training sets. For instance, in the big code task Method Name Prediction (i.e. predict method name by the method body), the popular source code embedding model code2vec (Alon et al., 2019) will predict wrong method names when only the two sides of an equation are switched (see Table 1.[1] line 6). Also, we find that code clone detection models that can get F1 scores over 0.95 in the test set experience a severe degradation when tested on a different dataset.

On the one hand, current big code datasets collected from software repositories are strongly limited by human labor and the boundedness of auto-labeling tools (auto-labeling tools can only collect a specific part of the population, producing certain kinds of bias (Bird et al., 2009)). On the other hand, deep learning models need much more (and diverse) data to distinguish the complicated connections between the various syntactic forms and the latent semantics of programs. Hence, big code models trained on limited datasets often suffer from data shortage or data bias and fail to generalize.

To deal with the challenge brought by data shortage and bias, many data augmentation approaches have been proposed in fields like Image Processing, such as random cropping (Krizhevsky et al., 2012) and flipping (Simonyan and Zisserman, 2015). Inspired by them, we notice that data augmentation can also be adopted in big code: a source code snippet can be seen as an image. Then, the flipping, rotation, and cropping of the image are like transforming the code into alternative forms, by which we expect to enrich the original datasets, forcing the models to learn more generalized relationships between syntax and semantics.

However, different from manipulating an image, programs have strict restrictions and specifications. A random perturbation in the code form could change the semantics totally, (most likely) introduce syntax error, or generate a too strange code to be written by humans and thus be useless for data augmentation. Therefore, in order to adopt data augmentation in big code, we

☆ Editor: Earl Barr.
* Corresponding author.
E-mail addresses: yushiwen14@nudt.edu.cn (S. Yu), tingwang@nudt.edu.cn (T. Wang), wj@nudt.edu.cn (J. Wang).
[1] This experiment was conducted on the website of code2vec (https://code2vec.org/) at 6/14/2020. More examples can be seen in Appendix I.

Table 1
An example of Code2Vec on method name prediction.

Before Transformation	After Transformation
`boolean f(int n) {`	`boolean f(int n) {`
`  if (n <= 1)`	`  if (n <= 1)`
`  return false;`	`  return false;`
`  for (int i = 2;`	`  for (int i = 2;`
`  i * i <= n; i++)`	`  i * i <= n; i++)`
`  {if (n % i == 0)`	`  {if (0 == n % i)`
`    return false;`	`    return false;`
`  }`	`  }`
`  return true;`	`  return true;`
`}`	`}`
Prediction: **isPrime** (✓)	Prediction: **has** (✗)

need to control the perturbation in a particular way such that the codes after perturbation can not only share the labels with the original ones (and without syntax error) but also be useful in improving the trained models' generalization ability.

We propose to fulfill the two requirements by carefully designing program transformation rules that can preserve both the semantics and the syntactic naturalness of code snippets. We hypothesize that if two code snippets are semantically equal, they can share the same label in most big code tasks. Also, if the transformed code is very likely to be written by programmers "naturally", it is highly possible to help models generalize better to the real-world population (at least not counterproductive).

In this paper, we first propose a general big code data augmentation framework based on our hypothesis, including rule design, rule evaluation, and rule selection. We then designed 18 transformation rules, such as LocalVarRenaming, For2While, VarDeclarationMerging. These rules are formally proved semantic-preserving by denotational semantics and tested syntax-naturalness-preserving by a probabilistic language model. Then, we use controlled experiments to demonstrate that semantic and syntax-naturalness-preserving are the expected properties for a transformation rule to be effective for big code data augmentation.

For the convenience of following researches, we also implemented and open-sourced a lightweight rule-based **s**emantic-and-naturalness **p**reserving **a**uto-**t**ransformation tool (SPAT) based on the designed rules. SPAT can handle dependency-missing programs (which are the majority in big code datasets) and is evaluated effective as an offline data augmentation tool in three big code tasks: code clone detection, method name prediction, and code commenting.

In summary, this paper makes the following contributions:

- A data augmentation framework for big code, which is based on semantic-and-naturalness preserving program transformation. It can be integrated with various data-driven models without changing the learning strategy.
- 18 transformation rules that are proved to be semantic-preserving by denotational semantics and tested to be syntax-naturalness-preserving with a probabilistic language model.
- A set of controlled experiments, through which we demonstrate that if a transformation rule is semantic and syntax-naturalness-preserving, it is highly possible to be effective for big code data augmentation and vice-verse. The experiments covered three big code tasks, four state-of-the-art deep learning models, and six datasets.
- An implementation of the proposed method and rules for Java, called SPAT. It can work on dependency-missing code fragments (partial programs) and transform 1,618 source code snippets per second on average.

This paper is organized as follows. Section 2 introduces two background theories that our study relies on. Section 3 describes our data augmentation technique, including the hypothesis of effectiveness (semantic-equivalence and syntax-naturalness), rule designing, rule evaluation, and rule selection. Section 4 shows the experiments we conducted to demonstrate the test of our hypothesis. Section 5 illustrates the ablation experiments to show what if a transformation does not have the two expected properties. Section 6 declares the threats to the validity of our study and their corresponding reasons. Section 7 introduces studies related to our work. We made the conclusion in Section 8.

2. Background

Before presenting our approach, it is necessary to introduce two fundamental theories: denotational semantics (Jung et al., 1996; Gordon, 1979) and language model (Bengio et al., 2000). We use denotational semantics to formalize the concept of "semantic equivalence" and prove the "semantic-preserving" of transformation rules. Also, we adopt a probabilistic language model to measure the syntax naturalness of code.

2.1. Denotational semantics

Denotational semantics aims to define the semantics of programming statements as denotations: functions. We first introduce a simple programming language named "FLOW" and then define denotational semantics upon it.

2.1.1. FLOW

In order to keep our study as general as possible, we use FLOW to discuss program semantics. FLOW is a simple programming language that can describe basic calculating and controlling. A legal FLOW program is like the following (ignore the linefeed):

```
x := 10;
y := 2;
(while (0 < y)
do   x := (x * y);
     y := (y - 1)
)
```

The formal grammar of FLOW is listed in Appendix 2.1. FLOW can only support the integer data type and does not support data structure or function definition. It also only supports limited variable identifiers. Nevertheless, it is enough to capture the basic concepts of programming language.

2.1.2. Denotational semantics of FLOW

The denotational semantics of a FLOW program (denoted as S) is defined as a function (the denotation $\mathbb{S}(S)$) which inputs a program state and outputs another program state. A program state is also treated as a function, which maps variable strings (*Var*) to integer values (*Value*). The mathematical foundation of denotational semantics is domain theory (Jung et al., 1996). We list the formal definitions details of domains, denotation sets, semantic functions and the proof of their legitimacy to Appendix 2.1. We denote the domain of program states as *State*, then the denotational semantics of a FLOW program ($\mathbb{S}(S)$) can be written as:

$$\begin{aligned} &State : Var \rightarrow Value \\ &\mathbb{S}(S) \in \ State \rightarrow State \end{aligned} \tag{1}$$

Similarly, we denote the domain of programs as *Sts*. *Sts* is the domain of all legal program syntax strings. We have $S \in Sts$. Then, the transformation rule R is defined as a function $R \in Sts \rightarrow Sts$. For each rule, we also have a condition ($C \in Sts \rightarrow \{true|false\}$) only under which ($C(S) = true$) will we apply the rule. The conditions are formalized as predicates in first-order logic. Therefore, that two programs (S and S') are semantically equal can be defined as:

$$\mathbb{S}(S) = \mathbb{S}(S') \tag{2}$$

and that a rule is semantic-preserving can be defined as:

$$\forall S \quad C(S) \Longrightarrow (\mathbb{S}(R(S)) = \mathbb{S}(S)) \tag{3}$$

2.2. Probabilistic language model

In this paper, we define the syntax naturalness of a program as the probability of it being in a large corpus of human-written programs. We believe the language model (Bengio et al., 2000) is a good way to measure it. A natural language model can be seen as a function that inputs a sequence of text and outputs its probability of being an actual natural language sequence. A programming language model works similarly. Given a program, it outputs the probability of the program being a "real" program. There are many ways to treat a program. We choose to treat it as a sequence of tokens because it is similar to how we read source code: from line to line and from token to token. This idea is also adopted by many big code researches for different purposes (Lin et al., 2019; Tu et al., 2014). They used a neural network as the language model. We directly use their measurement of syntax naturalness (cross-entropy):

$$N(S) = -\frac{1}{n}\sum_{i=1}^{n} logP(w_i|h) \tag{4}$$

where $S = w_1\ w_2\ \dots\ w_n$ (split by space). $P(w_i|h)$ is the probability estimated by the language model. w_i is the ith token to be predicted, and h is the hidden state calculated by an LSTM module with preceding tokens followed by w_i. The less $N(S)$, the more natural S. Eq. (4) can also be used directly as the loss function when training. After we train the model on a large corpus, $N(S)$ is expected to measure the probability of S being in the corpus.

Ideally, we want programs before and after transformation to share the same probability. But practically, we allow a tiny gap between them:

$$P(N(R(S_c)) - N(S_c) \leq \epsilon) \geq p \tag{5}$$

where $S_c \in \{S|C(S)\}$ (S_c satisfies the condition C). ϵ is the threshold, and p is the threshold probability. ($\epsilon = 0, p = 1$) indicates R is strictly syntax-naturalness-preserving. Practically, we set $p = 0.9$ and solve the least ϵ, and this ϵ represents how good the rule is at preserving the syntax naturalness, the less, the better.

3. Our approach

This section presents the semantic-and-naturalness preserving program transformation method for big code data augmentation. We first introduce the hypothesis of our method. Then, we explain the process of rule designing and rule evaluation. In the end, we describe how we select rules for specific tasks.

3.1. Hypothesis of effectiveness

As in Section 2.1.2, we regard a program transformation as a function R which inputs the original source code snippet S and outputs the transformed code snippet $R(S)$. Two requirements must be satisfied if we want $R(S)$ to be useful in data augmentation: (1) $R(S)$ must be "natural" in syntax (see Eq. (5)); (2) $R(S)$ and S must be semantically equal (see Eq. (3)). Note that "semantics" in this paper refers to denotational semantics. As we mentioned above, "syntax natural" means $R(S)$ is very likely to be written by programmers naturally. Transformed source code snippets that only exist as artificial products will not help a learning model generalize to the real-world data population. For instance, code obfuscation or high-level code optimization is not suitable for data augmentation since they usually destroy syntax naturalness. Furthermore, "semantic equivalence" is used to ensure the transformed code snippets can share the same label with the originals. We should point out that the second requirement only works for big code tasks that focus on understanding the code semantics, such as code search, code commenting, and semantic clone detection. It does not work well in fields like bug or vulnerability detection where $R(S)$ needs to contain the same bug as S rather than be semantically equal. For example, two code snippets involving the bug "Divide By Zero" do not have to be semantically equal. In this paper, we only consider big code tasks concentrating on code semantics (denotational semantics).

The overall process of data augmentation is simple: after we obtain a set of R that can satisfy the two requirements, we apply them on all transformable ($C(S)$ is true) code snippets S in the original dataset (D). Then, all generated (transformed) code snippets $R(S)$, together with all the original data S, are included in the augmented dataset($D+$). We expect models trained on $D+$ can generalize better.

3.2. Rule design

We abstract transformations by manually observing a set of programs that serve the same purpose (code clones in BCB (Svajlenko et al., 2014) and OJ (Mou et al., 2016)). The code clones are pairs of programs that serve the same purpose, such as different array sorting functions. We observe these pairs, and think about the inner reason why they can serve the same purpose with different syntax structure. For example, some code clones are made with the reason that one uses "for-statement" and the other uses "while-statement". Of course, they have also other different parts. But we only abstract one factor. The abstraction is then a mapping that maps one syntax form into another syntax form. Next, we have to think about three things: (1) if this mapping can be implemented with grammar tree manipulating and no API or global variable is involved; (2) if this mapping is semantic-preserving. And if it is we have to prove it formally by denotational semantics; (3) if this mapping is naturalness-preserving. If any one of them is negative, we cannot produce the abstraction as a transformation rule. We do not want a transformation to be too complicated for learning models to learn the syntax-semantics connections. Therefore, simple but important syntax variance is our target.

The motivation is straightforward: suppose the unseen instance can be transformed from the instances in the training set through our designed transformations, such as the example in Table 1, we can adopt the transformation of "equation switching" to augment the training set. Then, the trained model will learn a connection from source code syntax to the method names that treat the two kind of equation syntax as the same, resulting in a stronger generalization ability.

We formalize a transformation rule into two functions: $C \in Sts \rightarrow \{true|false\}$, the conditions only under which we will apply

Table 2
Descriptions of 18 transformation rules.

ID	Rule name	Description
Transformation Rules that may cause a different execution trace.		
3	ReverseIfElse	Switch the two code blocks in the if statement and the corresponding else statement.
4	SingleIF2ConditionalExp	Change a single if statement into a conditional expression statement.
5	ConditionalExp2SingleIF	Change a conditional expression statement into a single if statement.
8	InfixExpressionDividing	Divide a infix expression into two expressions whose values are stored in temporary variables.
9	IfDividing	Divide a if statement with a compound condition ($\wedge$, $\vee$, or $\neg$) into two nested if statements.
10	StatementsOrderRearrangement	Switch the places of two adjacent statements in a code block, where the former statement has no shared variable with the latter statement.
11	LoopIfContinue2Else	Replace the if-continue statement in a loop block with if-else statement.
14	SwitchEqualSides	Switch the two expressions on both sides of the infix expression whose operator is =.
15	SwitchStringEqual	Switch the two string objects on both sides of the $String.equals()$ function.
16	PrePostFixExpressionDividing	Divide a pre-fix or post-fix expression into two expressions whose values are stored in temporary variables.
17	Case2IfElse	Transform the “Switch-Case” statement into the corresponding “If-Else” statement.
Transformations Rules that cause the same execution trace.		
ID	Rule name	Description
0	LocalVarRenaming	Replace the local variables’ identifiers with new non-repeated identifiers.
1	For2While	Replace the for statement with an semantic-equivalent while statement.
2	While2For	Replace the while statement with an semantic-equivalent for statement.
6	PP2AddAssignment	Change the assignment $x++$ into x+=1.
7	AddAssignemnt2EqualAssignment	Change the assignment x+=1 into $x := x + 1$.
12	VarDeclarationMerging	Merge the declaration statements into a single composite declaration statement.
13	VarDeclarationDividing	Divide the composite declaration statement into separated declaration statements.

the transformation; and $R \in Sts \rightarrow Sts$, the transformation function. The R function only changes the sub-tree S where $C(S)$ is true. As an example, the rule While2For can be written as:

$$C(S) =_{df} (\exists B, S_1 \quad S = (\text{while } B \text{ do } S_1))$$
$$R((\text{while } B \text{ do } S_1)) =_{df} (\text{for (skip; } B\text{; skip) do } S_1)$$

Where B represents a boolean expression, and S stands for program statements. In fact, the definition above is simplified. We need to define the R for all $S \in Sts$, not just the while-loop statement. We use inductive definition to manage it. The formal definitions of all rules can be checked in Appendix 2.3. We put the natural language description of the 18 rules in Table 2.

After we formalized the transformation rule as C and R, we argue if this transformation can satisfy the two requirements: semantic and syntax-naturalness preserving. That is, Eq. (3) is true and Eq. (5) has a small ϵ.

3.3. Rule evaluation

For each rule, we first prove that it is semantic-preserving. Then, we will implement the rule for Java and prepare a benchmark, on which we measure how good it is at preserving syntax-naturalness, together with its applicability and speed.

3.3.1. Proof of semantic-preserving

We continue using the rule While2For as an example. Our proving target is Eq. (3). That is, we need to prove the following equation:

$$\mathbb{S}((\text{while } B \text{ do } S_1)) = \mathbb{S}((\text{for (skip; } B\text{; skip) do } S_1)) \tag{6}$$

By utilizing the definitions of denotational semantics, we can formally prove this equation. The proving process, together with proofs of other rules, are put to Appendix 2.3 due to space and easy-to-read consideration.

3.3.2. Syntax naturalness

As mentioned in Section 2.2, we use a probabilistic language model to measure the syntax-naturalness of a code snippet. The degree of syntax-naturalness-preserving of a rule is defined by ϵ in Eq. (5) when p is set to 0.9.

First, we need to implement the rules designed on FLOW to a practical programming language: Java. There are two reasons for us to do so: (1) Eclipse JDT provides powerful APIs supporting Java program rewriting; (2) many big code datasets contain Java code rather than FLOW code. The details of implementation can be checked in Appendix 3. The implemented tool (SPAT) is open-sourced.[2]

[2] https://github.com/Santiago-Yu/SPAT

Table 3
Syntax-naturalness-preserving, Applicability, and Speed of 18 Rules.

Rid	Name	ϵ $(p = 0.9)$	Applicability		#/sec
0	LocalVarRenaming(a → b)	$\epsilon = -3.2\text{e}{-04}$	9133/9133	1	1,615
1	For2While	ϵ = 1.6e-04	1684/9133	0.18	1,611
2	While2For	$\epsilon = -1.5\text{e}{-06}$	3194/9133	0.35	1,607
3	ReverseIfElse	ϵ = 4.3e-06	5998/9133	0.66	1,458
4	SingleIF2ConditionalExp	ϵ = 4.4e-04	1544/9133	0.17	1,603
5	ConditionalExp2SingleIF	ϵ = 2.5e-04	340/9133	0.04	1,682
6	PP2AddAssignment	ϵ = 2.7e-04	365/9133	0.04	1,695
7	AddAssignemnt2EqualAssignment	$\epsilon = -2.4\text{e}{-05}$	870/9133	0.10	1,683
8	InfixExpressionDividing	ϵ = 9.1e-06	1470/9133	0.16	1,611
9	IfDividing	ϵ = 2.3e-07	8/9133	0.001	1,551
10	StatementsOrderRearrangement	ϵ = 3.4e-04	6445/9133	0.71	1,326
11	LoopIfContinue2Else	$\epsilon = -1.8\text{e}{-06}$	201/9133	0.02	1,675
12	VarDeclarationMerging	ϵ = 1.5e-04	1471/9133	0.16	1,689
13	VarDeclarationDividing	ϵ = 3.2e-04	378/9133	0.04	1,688
14	SwitchEqualSides	ϵ = 7.4e-05	2586/9133	0.28	1,691
15	SwitchStringEqual	ϵ = 6.7e-05	596/9133	0.07	1,558
16	PrePostFixExpressionDividing	ϵ = 2.1e-04	73/9133	0.007	1,602
17	Case2IfElse	ϵ = 1.4e-04	89/9133	0.01	1,581

Algorithm 1 ϵ Estimation

```
Input: Bnc: benchmark, p: threshold probability,
       N: trained programming language model,
       C: transformation condition function,
       R: transformation rule function.
Output: ε: indicator of syntax-naturalness-preserving.
1: array := []
2: for S in Bnc do
3:     if C(S) then
4:         eps := N(R(S)) - N(S)
5:         array.add(eps)
6: array.sort_ascending()
7: index = array.len() * p
8: ε = array[index]
9: return ε
```

Next, we prepare a benchmark (Bnc) of 9133 source code snippets from BCB (Svajlenko et al., 2014) (provided in the supplementary material). Then, we train the language model NPLM[3] proposed by Bengio et al. Bengio et al. (2000) on a corpus of 1,143,729 code snippets (not crossed with the benchmark) collected from 262 GitHub projects (Svajlenko et al., 2014). The loss function we use is Eq. (4).

Finally, we use the trained NLPM as P in (4) to calculate $N(S_c)$ and $N(R(S_c))$ for all S_c in Bnc. We use maximum likelihood estimation to estimate the left hand of Eq. (5) and solve the least ϵ when p is set to 0.9. This process is described in Algorithm 1.

The ϵ of all rules can be checked in Table 3. By contrast, we also measured this indicator for program obfuscation,[4] which is $\epsilon = 0.013$, bigger than the rules for two orders of magnitude. Since the indicators of 18 rules are relatively small, we can claim that these rules are syntax-naturalness preserving.

For a visual explanation, we depict the distribution of syntax-naturalness of programs before and after transformation as box plots in Fig. 1. As can be seen, NPLM is able to distinguish obfuscated code, but it cannot tell the difference of syntax brought by SPAT rules. An extra discussion can be checked in Section 6.

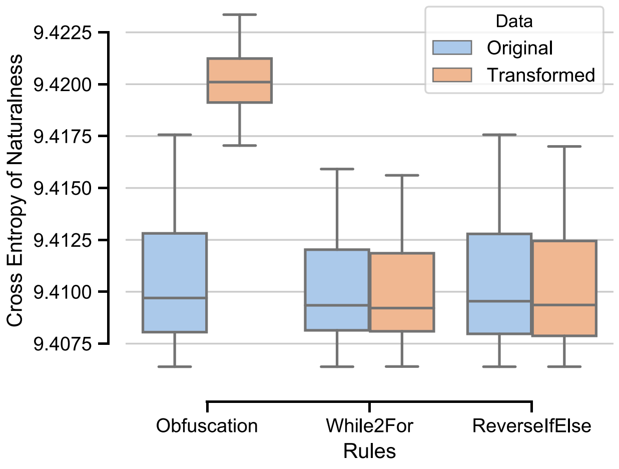

Fig. 1. The grouped box plots of the naturalness distribution of transformation rules and code obfuscation.

3.3.3. Applicability and speed

We measure the applicability of a rule with $\frac{\#\{S_c\}}{\#Bnc}$, i.e., the percentage of transformable S in the benchmark, and the speed as the number of code snippets it can process in a second. The test results are shown in Table 3.

The rule LocalVarRenaming has the strongest applicability of 1 because all code snippets in the benchmark have declared local variables. On the contrary, only eight code snippets can be transformed by the rule IfDividing. In addition, all rules can process about 1500 code snippets per second (1 million in 10 min) by parallelizing on *AMD 3970x* CPU.

3.4. Rule selection

Although all rules are proved semantic-preserving and tested syntax-naturalness-preserving, and we believe the qualified rules can enrich the data diversity and improve trained models' generalization ability, it does not ensure every rule can increase models' grade on all test set. In fact, not every rule is effective in all big code tasks. The effective rules vary between tasks, models, and even datasets, and sometimes the more general model may get lower grades on a biased test set.

In practice, there are many reasons that a qualified transformation rule brings no improvement. For example: (1) the code representation of the model ignored the corresponding difference

[3] We chose the implementation in https://github.com/chiaminchuang/A-Neural-Probabilistic-Language-Model.

[4] We use "Jobfuscate", which can be downloaded in https://www.duckware.com/jobfuscate/index.html.

brought by the rule; (2) the training set already contains enough samples for the learning model to distinguish the corresponding difference; (3) the diversity brought by the rule is not relevant to the task; (4) the search space of the learning model does not contain the desired relationship; (5) the test set itself is biased. As a result, the variance brought by data augmentation reduce the test grades.

In order to let the models get better grades, we often choose to apply rules selectively. This rule selection process is like tuning a hyperparameter on the validation set. We first augment the original training set with one rule at a time, train the model on each augmented training set, and test the trained models on the validation set. Rules that can bring improvements to the model on the validation set will be selected. After the selection, we augment the training set with all the selected rules. However, in our view, this process is more like to “cater to the dataset’s favor" since we need to get higher points on the limited test sets. Hence, we really suggest training with more diverse data (that means augmenting with all qualified rules) even facing a slight degradation on the test set.

In this paper, we adopted rule selection on two models: Code2vec (Alon et al., 2019) and DeepCom (Hu et al., 2018). The selection process eventually selected 10 to 14 of the 18 rules for different learning models on different datasets. The experiment result can be checked in Appendix 4.1. As a matter of fact, adopting all 18 rules will not significantly decrease the best performance with rule selection (the maximal degradation is 1.9% on Code2Vec and on the *small* dataset). The two properties can ensure that the augmented data will not be counter-productive.

4. Experiments evaluation

In this section, we are going to test our hypothesis. That is, we evaluate our rules’ effectiveness as a data augmentation tool for big code with three different tasks. We have selected the representative models in Method Name Prediction (Code2Vec (Alon et al., 2019)), Code Commenting (DeepCom (Hu et al., 2018)), and Code Clone Detection (ASTNN (Zhang et al., 2019) and TBCCD (Yu et al., 2019)). The three tasks are popular big code tasks with standard benchmarks and evaluation metrics. They are also easy to understand. The test sets, validation sets, hyperparameters, and training strategies of all models are kept the same as their original settings. Transformations will be applied to the training sets only. If the models trained on the augmented training sets have better performance than the originals, we can justify that semantic and syntax-naturalness preserving transformations are effective for data augmentation in big code. The Open Science material to reproduce the experiment results in this paper is put on Zenodo https://doi.org/10.5281/zenodo.5860659.

4.1. Method name prediction

Method name prediction is to predict the method name given the method body. The motivation of this task is to treat the method names as “semantic labels”. By training a model to predict the “semantic labels”, we want to get the continuous vectors for code semantic representation.

4.1.1. Code2vec

Code2vec[5] (Alon et al., 2019) is one of the most popular deep learning models for method name prediction. The main idea of Code2vec is to use a bag of “contexts" to represent a code snippet. Each “context" is a sequence of AST nodes connected by the “father and son" relationship extracted from the source code’s AST. Attention mechanism (Mnih et al., 2014) is adopted to aggregate the bag of “contexts" into a single continuous vector (called code vector). The word with an embedding vector nearest to the code vector is chosen as the prediction. After the training is finished, the Code2vec model can also be used as a pre-trained code representation model for other big code tasks such as code search and code completion.

[5] Its code and dataset are available at https://github.com/tech-srl/code2vec.

Table 4
Evaluation comparison between augmented training sets and original training sets. The numbers in brackets are the sizes of training sets.

Training set	Precision	Recall	F1	△
small(0.7 m)	0.233	0.211	0.221	
small-aug(2.4 m)	0.279	0.241	**0.259**	17.0%
med(3.0 m)	0.440	0.320	0.370	
med-aug(10.1 m)	0.452	0.361	**0.401**	8.5%
large(12 m)	0.631	0.544	0.584	
large-aug(50 m)	0.639	0.557	**0.595**	1.9%

4.1.2. Experiment setting

The datasets Code2vec uses are collected from 10,072 Java GitHub repositories (Alon et al., 2018). The data in training sets, validation sets, and test sets are chosen from different code repositories. Three datasets are built (*Small*, *Med*, *Large*). We kept all the settings the same as the original and adopted SPAT with rule selection to augment the training set. *Small* dataset has 664,821 methods for training, 23,498 for validating, 56,096 for testing. *Med* dataset has 3,002,033 methods for training, 410,616 for validating, 411,583 for testing. *Large* dataset has 12,636,998 methods for training, 371,364 for validating, 368,445 for testing. Adam optimizer is adopted to adjust the learning rate. The details can be checked in Appendix 4.1.

4.1.3. Evaluation

We evaluate Code2vec with precision and recall (the same as its original setting). The method names are treated as bags of case-insensitive sub-tokens W. The ground truth method name is marked as W_g, and the prediction is marked as W_p. Then, the precision for one prediction is calculated as $\mid W_g \cap W_p \mid / \mid W_p \mid$. Similarly, The recall for one prediction is calculated as $\mid W_g \cap W_p \mid / \mid W_g \mid$. The average precision and recall of all predictions are used to calculate the F1 score. The experiment results are listed in Table 4. Note that code2vec models trained on augmented training sets and original training sets are tested on the same test sets (which is the original test set).

As shown in Table 4, SPAT has improved the performances of Code2vec in all three datasets, showing its effectiveness as a data augmentation tool. One interesting phenomenon is that with the increasing of dataset size, the degree of improvement is weakened. It is most likely because, with the number of collected code snippets increasing, more and more variations of code forms are “naturally" included in the dataset. In other words, data augmentation will be unnecessary if we collected enough (diverse) data. However, method name prediction can be regarded as a self-supervised task: the data can be automatically collected and labeled from software repositories nearly without bias. Many big code tasks, on the contrary, have only limited datasets, and our method will be helpful.

4.2. Code commenting

Code commenting aims to generate comments in natural language for source code automatically. Code comments can save developers’ time in writing comments and help in understanding the source code.

Table 5
Evaluation comparison in code commenting.

Methods	BLEU-4	△
DeepCom (55766)	0.3817	
DeepCom+ (289622)	0.3894	2.0%
Hybrid-DeepCom (445812)	0.3951	3.5%
Hybrid-DeepCom+ (1570225)	**0.4017**	5.2%

4.2.1. DeepCom

DeepCom[6] (Hu et al., 2018) treats code commenting as a variant of Natural Language Translation. It is based on Neural Machine Translation (NMT) (Bahdanau et al., 2015), and focused on method-level commenting for Java. DeepCom consists of three parts: (1) encoder, (2) attention mechanism, and (3) decoder. The encoder uses LSTM to encode the sequence of tokens converted from AST. Then, the attention mechanism compresses hidden states outputted by the LSTM into context vectors. In the end, context vectors are used to calculate the comments in decoding. In 2020, the authors of DeepCom has updated their model (Hu et al., 2020), named "Hybrid-DeepCom". We carried out data augmentation on both DeepCom and Hybrid-DeepCom.

4.2.2. Experiment setting

The dataset[7] used in this work is the same as in DeepCom. They first prepared a large-scale Java corpus built from 9,714 open source projects from GitHub. And they use the Eclipse's JDT compiler to parse the Java methods into ASTs and extract corresponding Javadoc comments which are standard comments for Java methods. The methods without Javadoc are omitted. In the end, it consists of 69,708 <Java method, comment> pairs, of which 80% are for training, 10% for validation, and 10% for testing. All settings are kept the same as its original excepting the training set is augmented by SPAT with rule selection. The dataset prepared for Hybrid-DeepCom is bigger. It has 485,812 pairs, in which 445,812 are used for training, 20,000 for validation, and 20,000 for testing. Details can be checked in Appendix 4.2.1.

4.2.3. Evaluation

DeepCom uses machine translation evaluation metrics BLEU-4 score (Papineni et al., 2002) to measure the quality of generated comments. The BLEU-4 score ranges from 1 to 100 as a percentage value. The higher the BLEU-4, the closer the generated comment is to the ground truth comment. Table 5 lists the experiment result. We can see that SPAT also brings an improvement to DeepCom by augmenting its training set. After augmented, the size of the training set quintupled, and the BLEU-4 score of DeepCom is improved by 2.0%. The Hybrid-DeepCom can get a BLEU-4 score as 0.3951, 3.5% higher than the original DeepCom. The augmented Hybrid-DeepCom can get a BLEU-4 score as 0.4017, with an improvement as 5.2%.

4.3. Code clone detection

We only consider semantic code clones: pairs of code snippets that serve as the same function. Besides the need for software maintenance, code clone detection can also be used as code representation learning for downstream tasks. It can be seen as a binary classification problem: given two code snippets, a model is to predict if they are clones.

[6] Its code is available at https://github.com/xing-hu/EMSE-DeepCom.
[7] The dataset is available at https://github.com/xing-hu/DeepCom.

Table 6
Evaluation comparison of ASTNN(A) and TBCCD(T) trained on BCB and tested on Educoder.

Method	Precision	Recall	F1	△
A(0.06 m)	0.503	0.997	0.668	
A+(0.3 m)	0.523	0.976	**0.681**	2.0%
T(1.6 m)	0.944	0.688	0.796	
T+(6.9 m)	0.925	0.801	**0.859**	7.9%

4.3.1. ASTNN and TBCCD

Both ASTNN (Zhang et al., 2019) and TBCCD (Yu et al., 2019)[8] choose Abstract Syntax Tree as the input representation of code snippets. Their difference mainly lies in the methods used to aggregate the AST into one single vector. ASTNN first split the AST into a sequence of "statement" sub-trees (ST-trees) and then calculate the vectors of ST-trees by recursively pooling from their own sub-trees. Then, ASTNN uses LSTM to aggregate the statement vector sequence into one code vector. TBCCD, on the other hand, chooses tree-based convolution (Mou et al., 2016) as its basic module: a "continuous binary tree" convolution unit that can handle children nodes of different numbers. Cross entropy and cosine similarity are used to measure the semantic distance between two code vectors in the end.

4.3.2. Experiment setting

Unlike the previous two tasks where we train and test models within one dataset, we decide to evaluate ASTNN and TBCCD across different datasets because the two models have already achieved very high points on the original dataset BCB (Svajlenko et al., 2014) (both approximating 100%). Hence, it becomes difficult to tell the difference brought by SPAT on generalization ability. We choose to train and validate models on BCB and test them on another dataset: a dataset we collected from an open judgment website for programming called "Educoder". We prepare 74,323 records from "Educoder" as the test set *Educoder*[9].

Although ASTNN and TBCCD used the same dataset, they selected different training and validation sets from it. ASTNN only extracted 97,535 data records from BCB, 60% for training, 20% for validation. TBCCD extracted 1,583,930 pairs of code fragments for training and 416,328 for validation. The original test sets are not used.

Because the test sets and validation sets are from different datasets, selecting rules on the validation sets are not reliable. Therefore, we choose to use all rules without selection. Other training settings are the same as their originals and are detailed in Appendix 4.3.

4.3.3. Evaluation

Both ASTNN and TBCCD choose precision, recall, and F1 score to measure the performance. Since code clone detection is a binary classification problem, precision is calculated as $tp/(fp+tp)$ and recall is $tp/(fn+tp)$ where tp is true positive prediction, fp is false positive, and fn is false negative. We list the experiment results in Table 6. We can see that SPAT has brought improvements to the generalization ability of both ASTNN and TBCCD trained on the augmented training sets. After augmented by all rules, the sizes of the training sets quadrupled. The F1 score of the augmented ASTNN (A+) is improved by 2.0% on the test set *Educoder*, and the augmented TBCCD (T+) is improved by 7.9%.

[8] Their datasets and codes are available at https://github.com/yh1105/datasetforTBCCD and https://github.com/zhangj111/astnn.
[9] The dataset built by us is on https://github.com/Santiago-Yu/SPAT

5. Ablation study

In the previous experiments, we demonstrate that by augmenting training sets with semantic and syntax-naturalness preserving transformations, we could improve the trained learning models' generalization ability. This section focuses on what will happen if we augment the training sets with transformations that are not semantic-preserving or syntax-naturalness-preserving.

5.1. Non-semantic-preserving transformation

To control the variance to the minimal, we select four most applicable rules and introduce bugs into them: (1) LocalVarRenaming (rule 0) where the variable usages and declarations are mismatched; (2) For2While (rule 1) where the updating statements are forgotten; (3) ReverseIfElse (rule 3 where we forget to add the logical negation on the condition expression; (4) StatementsOrderRearrangement (rule 10) where we no more consider the switch conditions of adjacent statements. The buggy rules are not semantic-preserving anymore, but the syntax-variance brought by them is very close to the originals. As for the model and dataset, we choose Code2vec as the model and use the training set and evaluation set in the dataset *Small* and the test set from the dataset *Large*.

We depict the results in Fig. 2 . As can be seen, three out of four rules face degradation of effectiveness when a bug is introduced. However, we should notice that buggy transformation rules (rule 0 and rule 10) that do not preserve semantics anymore can sometimes still improve the model's performance. The buggy version of rule 10 even outperforms its correct version. We blame this phenomenon to the effect of "Adding Noise" (Bishop, 1995; An, 1996; Neelakantan et al., 2015). That is, for small training sets, adding noisy data can help training models to avoid over-fitting. It is also regarded as a kind of data augmentation.

However, whether noise addition can be helpful depends on a lot of factors: the dataset, the model design, and even the task itself. It is also unknown where and how to add what kind of noise. For example, when we add completely meaningless noise into the *Small* training set, the trained Code2vec model will face severe degradation (see Fig. 2). Specifically, we add data records whose method name and method body are mismatched into the training set as noise. Apparently, the model's performance monotonously decreases with the size of mismatched data increasing. In other words, completely meaningless noise will not help the model to learn the right relationship.

In conclusion, compared to the noise addition, semantic preserving transformation is at least not counterproductive since it generates correctly labeled data. Also, according to our experiments, models trained with semantic-preserving augmented data are more likely to generalize better.

5.2. Non-syntax-naturalness-preserving transformation

We use code obfuscation as the representative of non-syntax-naturalness-preserving transformation. Code obfuscation is semantic-preserving but cannot keep the syntax naturalness (as we measured its ϵ in Section 3.3.2). We adopted code obfuscation[10] as a data augmentation method on the three tasks. Like before, we keep all experiment settings as their originals and only increase the training sets with obfuscated code. The results are listed in Table 7.

All models are facing severe degradation when augmented with obfuscated code. It is reasonable because the training sets are very different from the test sets due to the added obfuscated source code. The learning models are not likely to overcome such an obstacle of the wide gap between syntax variance and semantics.

Through the two experiments above, we have tested the effectiveness of the two properties: "semantic-preserving" and "syntax-naturalness-preserving". Our hypothesis at Section 3.1 therefore holds.

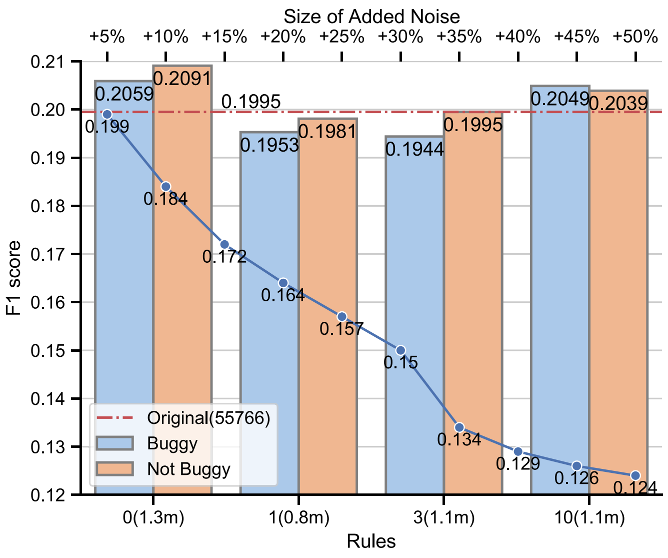

Fig. 2. The bar plot is the evaluation comparison between Code2vec trained on training sets augmented by four buggy and not buggy transformations. The red line is the F1 score of Code2vec trained on the original training set, and the numbers in brackets are the sizes of augmented training sets. The blue line plot is the F1 scores obtained by adding different size of meaningless noise into the original training set. +x% represents adding x% mismatched data of the training set's original size. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 7
Evaluation comparison before and after augmentation with code obfuscation.

Task	Method	Scores	▽
Method Name Prediction	code2vec_small(0.7 m)	0.22	
	code2vec_small+(1.3 m)	**0.18**	18.1%
	code2vec_med(3.0 m)	0.37	
	code2vec_med+(6.0 m)	**0.27**	25.9%
	code2vec_large(12 m)	0.58	
	code2vec_large+(25 m)	**0.41**	30.7%
Code Commenting	DeepCom(56k)	0.38	
	DeepCom+(112k)	**0.29**	23.7%
Code Clone Detection	ASTNN(0.06 m)	0.67	
	ASTNN+(0.12 m)	**0.52**	22.9%
	TBCCD(1.6 m)	0.80	
	TBCCD+(3.2 m)	**0.60**	25.0%

6. Threats to validity

Because this is the first systematic study of data augmentation in big code by program transformation to our knowledge, three considerable threats may harm our claims' validity.

First, the measurement of naturalness is worth further discussion: it is relatively easy for a programmer to judge if a program is written by a human, but we lack a perfect algorithm to judge it accurately. Therefore, utilizing the language model is the most practical and suitable way we can take. However, a language model is to measure the degree of similarity between a code snippet (c) and the training corpus rather than the naturalness of c, meaning there may be mistakes. According to our observation, the language model trained in this paper is at least able to

10 Jobfuscator https://www.pelock.com/products/jobfuscator.

distinguish natural source code snippets from obfuscated ones. With the progress of programming language models, we can have a more accurate measurement of naturalness.

Second, although we proved the transformation rules' semantic preserving property formally on FLOW, we did not prove they are consistent when we implement the rules for Java. On the one hand, FLOW and Java are similar in many places. Therefore many of our proof on FLOW still work for Java. On the other hand, proving the denotational semantic-preserving of transformation for a language like Java is much more complicated than for FLOW: method calls, class structure, global variables, various data types, etc. However, when we implement the rules, we will try to customize the rules to remain semantic-preserving on Java. Take StatementsOrderRearrangement as an example. We need to consider extra factors to keep the semantic-preserving proof in FLOW still available in Java. We usually tighten the transformation condition C to keep the validity. In the case of StatementsOrderRearrangement, in addition to "the variables in the adjacent statements are not overlapped", we also ask "the two adjacent statements do not involve method calling", where "method calling" do not exist in FLOW. By setting stricter transformation conditions, we can keep the validity of proof in FLOW with the cost of losing a certain degree of applicability. Details of implementation can be checked in Appendix 3.

Last, for a general data augmentation method, the variety of augmented data is always wanted, and the saturation points are different for different tasks, datasets, and even learning models. More rules will bring better performance on data augmentation. However, designing, proving, implementing, and testing a rule are difficult and time-consuming. We believe the second step in this direction would be "(semi-)automatically designed transformation". For example, one direction is to design a set of basic edit rules and combine them automatically by reinforcement learning to form transformations that are both semantic-preserving and syntax-naturalness-keeping.

7. Related work

7.1. Big code

An increasing number of studies focus on utilizing massive online software repositories like GitHub to build data-driven models to solve software engineering tasks. We evaluated four state-of-the-art open-source models in this paper: Code2vec (Alon et al., 2019), DeepCom (Hu et al., 2018), ASTNN (Zhang et al., 2019), and TBCCD (Yu et al., 2019). The task of Code2vec is to predict the name of a source code method given its body, by doing which it aims to learn a distributed semantic representation of programs. DeepCom is an end-to-end deep learning model whose task is to generate code comments from source codes. ASTNN and TBCCD are also deep learning models and aim at detecting source code pairs that are semantically similar (Type 4 Clone). This paper uses these models and their datasets as the experiment objects to verify our data augmentation method's effectiveness.

7.2. Program transformation

The key challenge of Program Transformation (PT) is to decide how to generalize concrete changes into an abstract transformation. To address this challenge, existing researches propose to utilize two different strategies: example-based and rule-based. Example-based PT synthesizes practical transformation from a set of examples with similar code changes (Jiang et al., 2019; Molderez and Roover, 2016). It can be very convenient since we only need to provide a few examples and expect the tools to capture the desired transformation. However, we find that the transformation synthesized by current example-based PT tools is often not consistent with the desired one. On the other hand, rule-based PT chooses to define transformation manually (Kim et al., 2013; Liu and Zhong, 2018). Manually written PT tools can be very accurate on specific transformations like certain types of bug repairing, but they often lack flexibility and need long-term development with expert assistance. The current rule-based PT tools usually serve specific purposes like program migration, optimization, obfuscation, and API updating. Hence, they are not specially designed to increase the natural diversity of code forms for big code datasets. They also do not consider partial programs (source code snippets that are not compile-able because their dependency information is missing). However, most big code datasets only contain partial programs due to labeling efficiency and limited storage. Therefore, we developed a partial program manageable rule-based PT tool for data augmentation from scratch.

7.3. Data augmentation

Data augmentation aims at artificially enlarging the training dataset from existing data using various transformations, such as translation (Wei et al., 2019), rotation (Shorten and Khoshgoftaar, 2019), flipping (Simonyan and Zisserman, 2015), cropping (Krizhevsky et al., 2012), and random erasing (Zhong et al., 2020). They have experimentally been proved useful to improve deep learning models' generalization ability in a wide range of tasks. Most of the data augmentation studies focus on multimedia processing. To our knowledge, there is no data augmentation method proposed in big code yet. Compared to the manipulations of an image, transforming a source code snippet is much more complex.

7.4. Similar works based on program transformation

Several studies adopted similar but different ideas as our work. Compton et al. (2020) propose transforming the code snippets by obfuscating the variable names to increase the quality of code embedding. However, it trains the deep learning model with only the obfuscated code, and the trained model shows worse performance on the original test set. Zhang et al. (2020) and Yefet et al. (2020) focus on attacking the model by transforming code snippets into alternative forms. They have shown that models trained with adversarial examples can resist their attack, but they did not prove that the adversarial examples can help models generalize better to the real-world population, which is our paper's focus. DeepBugs (Pradel and Sen, 2018) uses rule-based transformations to build a dataset rather than augmenting one. The transformation rules of DeepBugs are bug-introducing patterns. Therefore, they want their model to learn the designed transformation patterns, while we do not wish our models to learn or over-fit the transformations (Our test sets contain no transformed data). Rabin et al. (2020) propose to test the generalization ability of deep learning models by semantic preserving transformations. Their work is actually "Stability Testing" (which measures the robustness of learning models by transforming the data in the test set, see Wang and Su (2019)), while we are investigating the possibility of data augmentation (which increases the size of the training set).

Unlike these works, we propose a general method for big code data augmentation by rule-based program transformation, which involves an augmentation framework and strategies for designing, evaluating, and selecting transformation rules for data augmentation.

8. Conclusion

This paper systematically investigated the application of data augmentation to big code data by program transformation. We demonstrate that a transformation needs to preserve both the semantics and syntax naturalness of code snippets to be helpful as a data augmentation method and vice verse through experiments. Nevertheless, like the other data augmentation techniques, we cannot ensure that an improvement must be brought by data augmentation for every big code task. But semantic and syntax-naturalness preserving transformations can be at least not counter-productive and are very likely to be effective since they improved the generalization-abilities of all models in our experiments. During the research, we designed 18 transformation rules that are proved semantic-preserving and tested syntax-naturalness-preserving. We also implemented a partial program handle-able Program Transformation tool named SPAT for Java and evaluated its effectiveness as a data augmentation tool on three big code tasks.

CRediT authorship contribution statement

Shiwen Yu: Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Visualization. **Ting Wang:** Conceptualization, Supervision, Writing – review & editing. **Ji Wang:** Conceptualization, Methodology, Software, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research has been supported by the National Natural Science Foundation of China under Grant No. 62032024.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2022.111304.

References

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2018. A general path-based representation for predicting program properties. In: Foster, J.S., Grossman, D. (Eds.), Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. ACM, pp. 404–419. http://dx.doi.org/10.1145/3192366.3192412.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), http://dx.doi.org/10.1145/3290353.

An, G., 1996. The effects of adding noise during backpropagation training on a generalization performance. Neural Comput. 8 (3), 643–674.

Bahdanau, D., Cho, K., Bengio, Y., 2015. Neural machine translation by jointly learning to align and translate. In: Bengio, Y., LeCun, Y. (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. URL http://arxiv.org/abs/1409.0473.

Bengio, Y., Ducharme, R., Vincent, P., 2000. A neural probabilistic language model. In: Leen, T.K., Dietterich, T.G., Tresp, V. (Eds.), Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA. MIT Press, pp. 932–938, URL http://papers.nips.cc/paper/1839-a-neural-probabilistic-language-model.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P., 2009. Fair and balanced? Bias in bug-fix datasets. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE '09, Association for Computing Machinery, New York, NY, USA, ISBN: 9781605580012, pp. 121–130. http://dx.doi.org/10.1145/1595696.1595716.

Bishop, C.M., 1995. Training with noise is equivalent to tikhonov regularization. Neural Comput. 7 (1), 108–116.

Compton, R., Frank, E., Patros, P., Koay, A., 2020. Embedding java classes with code2vec: Improvements from variable obfuscation. CoRR arXiv:2004.02942.

Gordon, M.J.C., 1979. The Denotational Description of Programming Languages - An Introduction. Springer, ISBN: 978-3-540-90433-5.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Khomh, F., Roy, C.K., Siegmund, J. (Eds.), Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018. ACM, pp. 200–210. http://dx.doi.org/10.1145/3196321.3196334.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2020. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 25 (3), 2179–2217.

Jiang, J., Ren, L., Xiong, Y., Zhang, L., 2019. Inferring program transformations from singular examples via big code. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE, pp. 255–266. http://dx.doi.org/10.1109/ASE.2019.00033.

Jung, A., Fiore, M., Moggi, E., O'Hearn, P.W., Riecke, J.G., Rosolini, G., Stark, I., 1996. Domains and denotational semantics: History, accomplishments and open problems. School of Computer Science Research Reports-University of Birmingham CSR.

Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches. In: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, pp. 802–811. http://dx.doi.org/10.1109/ICSE.2013.6606626.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems. pp. 1097–1105.

Lin, B., Nagy, C., Bavota, G., Lanza, M., 2019. On the impact of refactoring operations on code naturalness. In: Wang, X., Lo, D., Shihab, E. (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019. IEEE, pp. 594–598. http://dx.doi.org/10.1109/SANER.2019.8667992.

Liu, X., Zhong, H., 2018. Mining stackoverflow for program repair. In: Oliveto, R., Penta, M.D., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018. IEEE Computer Society, pp. 118–129. http://dx.doi.org/10.1109/SANER.2018.8330202.

Mnih, V., Heess, N., Graves, A., Kavukcuoglu, K., 2014. Recurrent models of visual attention. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada. pp. 2204–2212, URL http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.

Molderez, T., Roover, C.D., 2016. Search-based generalization and refinement of code templates. In: Sarro, F., Deb, K. (Eds.), Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings. In: Lecture Notes in Computer Science, 9962, pp. 192–208. http://dx.doi.org/10.1007/978-3-319-47106-8_13.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. AAAI'16, AAAI Press, pp. 1287–1293.

Neelakantan, A., Vilnis, L., Le, Q.V., Sutskever, I., Kaiser, L., Kurach, K., Martens, J., 2015. Adding gradient noise improves learning for very deep networks. arXiv preprint arXiv:1511.06807.

Papineni, K., Roukos, S., Ward, T., Zhu, W., 2002. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA. ACL, pp. 311–318. http://dx.doi.org/10.3115/1073083.1073135, URL https://www.aclweb.org/anthology/P02-1040/.

Pradel, M., Sen, K., 2018. DeepBugs: A Learning approach to name-based bug detection. In: Proceedings of the ACM on Programming Languages, vol. 2, OOPSLA. ACM New York, NY, USA, pp. 1–25.

Rabin, M., Islam, R., Bui, N.D., Yu, Y., Jiang, L., Alipour, M.A., 2020. On the generalizability of neural program analyzers with respect to semantic-preserving program transformations. arXiv preprint arXiv:2008.01566.

Shorten, C., Khoshgoftaar, T.M., 2019. A survey on image data augmentation for deep learning. J. Big Data 6, 60. http://dx.doi.org/10.1186/s40537-019-0197-0.

Simonyan, K., Zisserman, A., 2015. Very deep convolutional networks for large-scale image recognition. In: Bengio, Y., LeCun, Y. (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. URL http://arxiv.org/abs/1409.1556.

Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M., 2014. Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 476–480. http://dx.doi.org/10.1109/ICSME.2014.77.

Tu, Z., Su, Z., Devanbu, P.T., 2014. On the localness of software. In: Cheung, S., Orso, A., Storey, M.D. (Eds.), Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. ACM, pp. 269–280. http://dx.doi.org/10.1145/2635868.2635875.

Wang, K., Su, Z., 2019. Learning blended, precise semantic program embeddings. CoRR arXiv:1907.02136.

Wei, J.W., Suriawinata, A.A., Vaickus, L.J., Ren, B., Liu, X., Wei, J., Hassanpour, S., 2019. Generative image translation for data augmentation in colorectal histopathology images. In: Dalca, A.V., McDermott, M.B.A., Alsentzer, E., Finlayson, S.G., Oberst, M., Falck, F., Beaulieu-Jones, B.K. (Eds.), Machine Learning for Health Workshop, ML4H@NeurIPS 2019, Vancouver, BC, Canada, 13 December 2019. In: Proceedings of Machine Learning Research, vol. 116, PMLR, pp. 10–24, URL http://proceedings.mlr.press/v116/wei20a.html.

Yefet, N., Alon, U., Yahav, E., 2020. Adversarial examples for models of code. In: Proceedings of the ACM on Programming Languages, vol. 4, OOPSLA. ACM New York, NY, USA, pp. 1–30.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: Proceedings of the 27th International Conference on Program Comprehension. ICPC '19, IEEE Press, pp. 70–80. http://dx.doi.org/10.1109/ICPC.2019.00021.

Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., Jin, Z., 2020. Generating adversarial examples for holding robustness of source code processing models. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, the Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, the Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. AAAI Press, pp. 1169–1176, URL https://aaai.org/ojs/index.php/AAAI/article/view/5469.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the 41st International Conference on Software Engineering. ICSE '19, IEEE Press, pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zhong, Z., Zheng, L., Kang, G., Li, S., Yang, Y., 2020. Random erasing data augmentation. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, the Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, the Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. AAAI Press, pp. 13001–13008, URL https://aaai.org/ojs/index.php/AAAI/article/view/7000.