Fold2Vec: Towards a Statement-Based Representation of Code for Code Comprehension

FRANCESCO BERTOLOTTI and WALTER CAZZOLA, Università degli Studi di Milano, Italy

We introduce a novel approach to source code representation to be used in combination with neural networks. Such a representation is designed to permit the production of a continuous vector for each code statement. In particular, we present how the representation is produced in the case of Java source code. We test our representation for three tasks: *code summarization*, *statement separation*, and *code search*. We compare with the state-of-the-art *non-autoregressive* and *end-to-end* models for these tasks. We conclude that all tasks benefit from the proposed representation to boost their performance in terms of F1-score, accuracy, and mean reciprocal rank, respectively. Moreover, we show how models trained on code summarization and models trained on statement separation can be combined to address methods with tangled responsibilities, meaning that these models can be used to detect code misconduct.

CCS Concepts: • **Computing methodologies → Neural networks;**

Additional Key Words and Phrases: Big code, learning representations, method name suggestion, intent identification

ACM Reference format:
Francesco Bertolotti and Walter Cazzola. 2023. Fold2Vec: Towards a Statement-Based Representation of Code for Code Comprehension. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 6 (February 2023), 31 pages.
https://doi.org/10.1145/3514232

1 INTRODUCTION

Premise. Code comprehension is one of the most challenging and time-consuming tasks in software development. Often code is poorly documented, and its functionalities are tangled and scattered throughout the whole codebase. This makes tasks like debugging and maintenance difficult and time consuming. Therefore, the need for tools to support the developer in code comprehension is becoming critical. This need is also reflected by recent research trends where the focus is on *automatic code generation* [8, 46, 48, 55], *automatic code completion* [52, 66], *code summarization* [1, 5–7, 65], and *automatic test generation* [16, 23].

Tangled intents. One aspect that renders code comprehension difficult is that the code of a single functionality is often polluted by the code of other functionalities. We refer to this phenomenon as tangled intents. This is where most of the models based on code comprehension have limits. Comprehension occurs at coarse grain usually set at the method/function level [1, 5–7, 65], which limits

This work was partially funded by the MUR projects "DSURF" (PRIN 2015B8TRFM) and "T-LADIES" (PRIN 2020TL3X8X).
Authors' address: F. Bertolotti and W. Cazzola, Università degli Studi di Milano, Computer Science Department, Via Celoria 18, 20135, Milan Italy; emails: francesco.bertolotti@unimi.it, cazzola@di.unimi.it.

1049-331X/2023/02-ART6 $15.00
https://doi.org/10.1145/3514232

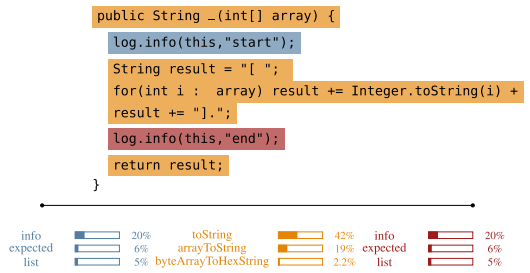


Fig. 1. Java tangled method with intent classifications (orange, blue, and red) by three separate runs of `fold2vec`.

their applicability. Let us demonstrate the problem on an example involving intent identification. Figure 1 shows the method whose behavior we have to guess by giving it a proper description. Even if the code is quite short, it suffers from the preceding entangled intents—the code belonging to different intents has a different colored background. To tell apart and name the code of every single intent is difficult and often ambiguous. Moreover, a wrong classification presents obstacles to proper reuse, refactoring, and maintenance, among others.

Motivation. Models like `code2seq` [5] and `code2vec` [7] have demonstrated that training a **neural network (NN)** with specifically designed representations for code helps to achieve better results with respect to standard *natural language processing* techniques. However, both `code2seq` and `code2vec` are bounded by their representation, leaf-to-leaf **abstract syntax tree (AST)** path. In our view, a fine-grained prediction that considers potentially non-consecutive statements should be designed to deal with disconnected sub-trees of the AST. In these cases, leaves from different sub-trees cannot be connected by leaf-to-leaf paths. This limitation leads to shorter and narrower paths, for which it has been shown to hurt the performance [6]. These motivations lead us to design a novel representation that aggregates all of the information available from the different sub-trees. This information can be used to gather statements into coherent groups. These groups, called *intents*, can be used to detect bad code practices.

Research questions. To address the mentioned issues and limitations, we try to answer the research questions:

***RQ*$_1$:** Can NNs for code comprehension benefit from a statement-based code representation?
***RQ*$_2$:** Can such a code representation enable a NN to detect code misconduct?
***RQ*$_3$:** Which neural component with which neural representation behaves the best?

Statement-based representation. Kiros et al. [45] and Dwivedi and Shrivastava [24] build a hidden representation aligned with its semantics for each natural language phrase of the document. All of the hidden representations contribute together to perform a specific task on the whole document. In our view, a similar approach could be exploited to improve code comprehension and provide an answer to RQ$_1$ where the analogous of a natural language phrase is a code statement. Our NN, called `fold2vec`, builds a hidden representation for each code statement separately from the others. Then, the combined hidden representations contribute to the classification of the method. In this direction, our main contribution is a novel code representation. Given a code snippet, its AST is considered. We split the AST into sub-trees, each representing a single code statement. Each sub-tree is folded into two sequences of tokens—one for the terminals and one for the non-terminals—via a pre-order visit. All token sequences are then fed to the NN, and they contribute to the classification.

Code summarization and code search. To answer RQ_1, we show that our code representation improves the state of the art for the code summarization and code search tasks. It is worth noting that for code summarization, the dataset used to perform this task exploits the fact that method names often summarize the method behavior. This assumption can be seen as a rough approximation, but it permits to access a vast amount of available data on which the training can be performed. Usually, more data lead to better and more robust results [29].

Statement separation. An intent can be any subset of statements, not necessarily adjacent in the code, pursuing the same goal. In our view, intent identification could be possible by applying a NN specifically trained for the task of statement separation. This NN can be used to induce a metric over statements regarding the statement topics, meaning that statements with the same topic would be close to each other, and statements with different topics would end up far apart from each other. With an hierarchical clustering algorithm, the induced metric can be used to cluster statements that share the same topic. To answer RQ_2, we train `fold2vec` for the statement separation task. We call this model `stmt-fold2vec`. However, the novelty of the task limits the availability of baselines. To address this issue, we adapt the previously mentioned work (`code2vec`) for the statement separation task. We demonstrate that the novel code representation leads to improvements also for this task.

Overview. The rest of the article is organized as follows. Section 2 introduces the terminology. Section 3 illustrates the design of our source code representation and the NN architecture used on top of it. Section 4 discusses and compares `fold2vec` results to the state-of-the-art models for the code summarization and code search. It also brings some evidence that `fold2vec` can classify single or groups of statements to enable intent identification. We present our ablation study in Section 5 and discussion in Section 6. Finally, in Sections 7 and 8, we examine some related work and draw our conclusions, respectively.

2 BACKGROUND

The aim of this section is to set a common terminology used in this article by recalling the definition of some core concepts as NN layers and AST.

Neural network. A NN is "a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs [21]". Usually, NNs are organized in layers. In a *supervised learning* environment, the NN is trained through *gradient descent* techniques to minimize a loss from a set of examples called the *training set*. Additionally, a *validation set* and *test set* are used to evaluate the NN in a developing phase and at the end of the process, respectively. These phases are meant to limit the effect of *overfitting*.

Autoregressive architecture. A neural architecture is said to be autoregressive when each prediction is part of several single classifications, each based on the previous ones. For example, while predicting how a text continues, an autoregressive architecture predicts the next token by considering also the previous predictions.

Pre-trained and end-to-end models. A model is said to be pre-trained when it was already trained on a specific task. These models can exploit a task for which there is a high availability of data to learn intrinsic features of the media (text, code, images, videos, etc.) that can be useful in a wide range of tasks. The learned features can then be reused and fine-tuned for tasks of which there is less availability of data. Pre-trained models are usually trained on extremely large datasets that require expensive hardware during training. In contrast, end-to-end models are trained from

scratch directly on a task of interest. These models are usually smaller, customizable, and easily reproducible. Using end-to-end models is convenient when the task of interest already has a large dataset available.

Multi-layer perceptron. The **multi-layer perceptron (MLP)** is the composition of several *dense layer*s. A dense layer applies to a real vector $\vec{x} \in \mathbb{R}^{1\times n}$ the transformation:

$$\vec{y} = \mathbf{W}\vec{x} + \vec{b},$$

where $\mathbf{W} \in \mathbb{R}^{n\times m}$ and $b \in \mathbb{R}^m$ are trainable parameters randomly initialized. Usually, a non-linear activation function is applied to the output $\vec{y}$ (e.g., $\vec{z} = \tanh(\vec{y})$).

Embeddings layer. The *embeddings layer* maps tokens to real vectors. This kind of layer is a matrix $\mathbf{E} \in \mathbb{R}^{t\times d}$, where t is the number of tokens to be considered and d is the size of each embedding. Pennington et al. [60] showed that values for d between 50 and 500 are a good fit in most cases. Each token, indexed by i such that $0 \leq i < t$ is mapped to the i-th row of $\mathbf{E}$. $\mathbf{E}$ is made up of trainable parameters randomly initialized. $\mathbf{E}$ can be either pre-trained as in the work of Mikolov et al. [51] and Pennington et al. [60] or learned during network training like in the work of Alon et al. [5, 7].

Layer normalization. Layer normalization [9] is a regularization technique used to improve the network training time and *test error*. Apart from the *input layer*, each layer learns its input distribution from the output of the preceding layer. During training, the output distribution from a layer fed to another is constantly changing as the NN is updated; this is known as the *covariate shift* problem. Given a set of features, layer normalization computes the mean and standard deviation, then re-centers and re-scales the features using an extra normalization parameter. By normalizing the distribution, the *covariate shift* is reduced and the network learns faster.

Recurrent neural network. A **recurrent neural network (RNN)** is the neural architecture commonly used when processing input sequences. It processes each element of the input sequence considering the output of the preceding computation. Given the input sequence $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$, it computes the t-th stage as

$$\vec{h}_t = \texttt{RNN}(\vec{h}_{t-1}, \vec{x}_t, \theta),$$

where, $\vec{h}_{t-1}$ is a hidden state and θ represents the network parameters. The hidden state is a sort of network memory updated as an input element is processed. RNN training can suffer from both the *vanishing* and the *exploding gradient problem* as explained in [12, 59]—that is, parameter updates can become either too small or too big, respectively.

Long short-term memory. **Long short-term memory (LSTM)** [26] is a type of RNN that deals with the *gradient problem* [59] so that the network can make long-term associations. Each layer can also use two LSTMs at the same time, where one consumes the input sequence and the other the reversed input sequence; this configuration is called **bidirectional long short-term memory (BiLSTM)**. As well, an additional LSTM layer can consume the output of a previous LSTM layer; this configuration is called *stacked LSTM*.

Attention mechanism. An *attention mechanism* helps the network focus on the input portion that should be more relevant for its task. For example, articles are less informative than nouns when you are trying to understand a sentence. The attention mechanisms used in this article are **self-attention (SA)** [72] and **global attention (GA)** [47]:

- *Global attention*: Given the input sequence $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n \in \mathbb{R}^{1\times d}$ and a parameter $\vec{a} \in \mathbb{R}^{d\times 1}$ randomly initialized, the GA masks the useless information by computing some input scores

$s_i = \vec{x}_i \cdot \vec{a}$ that are normalized through a `softmax` function:

$$\alpha_i = \texttt{softmax}(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}} \qquad \hat{x} = \sum_{i=1}^{n} \alpha_i \vec{x}_i,$$

where α_i are real values between 0 and 1 whose sum is 1. The real vector $\hat{x}$ summarizes the inputs with a weighted sum. Each weight is a normalized relevance score. $\vec{x}_i$ has a low relevance and does not contribute to $\hat{x}$ when α_i is close to 0. Conversely, $\vec{x}_i$ has a high relevance when α_i is close to 1. The GA process can be applied multiple times on the same input sequence by training multiple a vectors. In this case, we will use the term *multi-head GA*.

- *Self-attention*: In the case of SA, each input element $\vec{x}_i$ is scored with respect to all of the other elements instead of using a global vector. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the matrix made up of the row vectors $\vec{x}_i$, and $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V \in \mathbb{R}^{d \times d}$ are some parameters randomly initialized. Three linear transformations are applied to the input sequence:

$$\mathbf{Q} = \mathbf{X} \cdot \mathbf{W}_Q \qquad \mathbf{K} = \mathbf{X} \cdot \mathbf{W}_K \qquad \mathbf{V} = \mathbf{X} \cdot \mathbf{W}_V,$$

where the i-th rows of $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are different linear transformations of $\vec{x}_i$. $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are combined into $\mathbf{Z}$:

$$\mathbf{S} = \mathbf{Q} \cdot \mathbf{K}^{\mathsf{T}} \qquad \mathbf{Z} = \texttt{softmax}(\mathbf{S}) \cdot \mathbf{V}.$$

The score s_{ij} of x_i with respect to $\vec{x}_j$ is given by the dot product of the i-th row and the j-th column of $\mathbf{Q}$ and $\mathbf{K}^{\mathsf{T}}$, respectively. The `softmax` function applied by rows normalizes the scores so that the i-th row of $\texttt{softmax}(\mathbf{S})$ contains the weights for $\vec{x}_i$ with respect to all other elements. $\mathbf{Z}$ is the weighted sum obtained by multiplying the normalized scores and $\mathbf{V}$. Storing S in memory has a cost of $O(n^2)$, which often limits the applicability of this mechanism.

Abstract syntax tree. An AST is a unique tree-shaped representation of a program. Formally, it is a quintuple $(N, T, X, s, \delta, \psi)$, where the following apply:

- N is the set of internal nodes of the AST. It is split into N_S and N_E, where the former contains only statement-like nodes and the latter only expression-like nodes.
- T is the set of leaf nodes of the AST.
- X is a set of values that the leaf nodes can assume.
- $s \in N$ is the root node of the AST.
- $\delta : N \to (N \cup T)^+$ is the function that maps each node $n \in N$ to a list of its children.
- $\psi : T \to X$ is a function that maps each leaf node $n \in T$ to its actual values.

3 FOLD2VEC

In this section, we present the proposed model, named `fold2vec`, and the steps needed to train and test it. In Section 3.1, we explain how to extract the features to feed the NN. In Section 3.2, we describe how the neural architecture is composed and how its components interact. In Section 3.3, we discuss the tuning process of `fold2vec`.

3.1 Code Feature Extraction

Overview. As mentioned previously, `fold2vec` adopts a source code representation based on linearizations of the method AST. Before diving into the details, let us summarize how these linearizations are extracted:

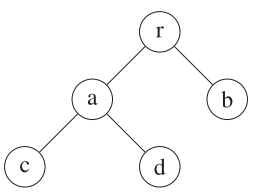


Fig. 2. Two possible unfoldings: $f(r)$ applies a pre-order visit starting from the node r; f' returns only leaves starting from the node r.

(1) Given a method, we use a parser to obtain its AST. For example, step ① in Figure 3 (presented later) shows a method, and step ② in the figure shows its AST.
(2) We split the AST into sub-trees such that each sub-tree represents a statement in the method. For example, step ① in Figure 3 has three statements colored, and step ② in the figure has the respective sub-trees colored in the same way.
(3) We linearize the obtained sub-trees into two ways:
    - A pre-order visit registering non-terminals.
    - A pre-order visit registering terminals.

    For example, step ② in Figure 3 shows three sub-trees that will be linearized, and step ③ in the figure shows three lists each made of two linearizations.
(4) Last, We tokenize terminals and remove integer literals.

Tree unfolding. We refer to a tree unfolding as a linearization of a tree through an operation

$$f : (N \cup T)^+ \to (N \cup X)^*,$$

where f is a placeholder representing the tree linearization operation. It can be anything, such as a pre/post-order visit or a function returning the root node. Figure 2 shows some possible behaviors for f. Tree unfoldings can be used to build a hybrid tree representation where some of its sub-trees are folded into a more informative node by applying f and lowering the tree height. Based on the findings of Shi et al. [67], short trees can be more easily exploitable by NNs because information propagates on shorter paths reducing the number of classification errors.

Contextual unfolding. **Contextual unfoldings (CU)** are just tree unfoldings designed to deal with the AST. In particular, CUs are obtained from the sub-trees of each statement AST. We have designed two different types of CU: one for non-terminal nodes (e.g., `IfStmt`, `ForStmt`) and one for terminal nodes (i.e., identifiers, literals, and keywords), respectively div_X and div_N (step ③ in Figure 3). The two CUs provide different information to the NN—the former captures code structure and the latter its topic [56]. Let us define two helper functions to calculate the corresponding CUs:

$$\texttt{div}_X : (X \cup N)^+ \to X^*, \qquad \texttt{div}_N : (X \cup N)^+ \to N^*.$$

Given a node sequence, $\texttt{div}_X$ and $\texttt{div}_N$ return only terminal or non-terminal nodes, respectively. They are inductively defined over the length of their inputs.

Base: Let $|x| = 1$.

$$\texttt{div}_X(x) = \begin{cases} x & \text{if } x \in X \\ \varepsilon & \text{otherwise} \end{cases} \qquad \texttt{div}_N(x) = \begin{cases} x & \text{if } x \in N \\ \varepsilon & \text{otherwise} \end{cases}$$

Step: Let $|x| = n + 1$, where $x = y \cdot z$ with $|y| = 1$ and $|z| = n$.

$$\texttt{div}_X(x) = \begin{cases} y \cdot \texttt{div}_X(z) & \text{if } y \in X \\ \texttt{div}_X(z) & \text{otherwise} \end{cases} \qquad \texttt{div}_N(x) = \begin{cases} y \cdot \texttt{div}_N(z) & \text{if } y \in N \\ \texttt{div}_N(z) & \text{otherwise} \end{cases}$$

We also define a third helper function (f_v) to linearize a given statement tree via a pre-order visit that skips the sub-trees of other statements. It is defined as follows:

$$f_v : (N \cup T)^+ \rightarrow (N \cup X)^+ \qquad f_v(x) = \begin{cases} \psi(x) & \text{if } x \in T \\ x \cdot \texttt{concat}\{f_v(c) | \forall c \in \delta(x) \setminus N_S\} & \text{otherwise.} \end{cases}$$

Given a statement node $x \in N_S$ as input, f_v consumes x and recursively calls itself on the children of x that are not statement nodes by applying $\delta(x) \setminus N_S$. In step ② of Figure 3, f_v is called twice on the root node of every statement sub-tree (i.e., `WhileStmt`, `IfStmt` and `ExprStmt`).
Finally, we define the two linearization functions.

$$f_{cu}^{(N)} : (N \cup T)^+ \rightarrow N^* \qquad x \mapsto \texttt{div}_N(f_v(x)) \qquad f_{cu}^{(X)} : (N \cup T)^+ \rightarrow X^* \qquad x \mapsto \texttt{div}_X(f_v(x))$$

Again, in step ② of Figure 3, $f_{cu}^{(N)}$ and $f_{cu}^{(X)}$ are called once on the root node of every statement sub-tree. The extraction function (f_e) exploits the helper functions to calculate the token sequences to pass to `fold2vec`. It is defined as follows.

$$f_e : (N \cup T)^+ \rightarrow (N^* \times X^*)^+$$

$$f_e(x) = \begin{cases} \varepsilon & \text{if } x \in T \\ \texttt{concat}\{f_e(c) | \forall c \in \delta(x)\} & \text{if } x \in N_E \\ [f_{cu}^{(X)}(x), f_{cu}^{(N)}(x)] \cdot \texttt{concat}\{f_e(c) | \forall c \in \delta(x)\} & \text{otherwise} \end{cases}$$

f_e starts visiting the AST from the root node (in Figure 3, that is the `WhileStmt`); the visit is in pre-order. It uses both $f_{cu}^{(X)}$ and $f_{cu}^{(N)}$ functions to produce the relative CUs at each statement node. The computed CUs are concatenated together in the output.

Tokenization. We additionally split all terminal nodes into sub-tokens (e.g., `"toString"` is split into `"to"` and `"string"`). The split is done to keep the size of the embeddings table $\mathbf{E}$ reasonable (see Section 2). $\mathbf{E} \in \mathbb{R}^{t \times d}$, where t is the number of unique words that can be stored and d is the size dedicated to representing each word. Each unique word is always mapped to a single entry in $\mathbf{E}$. If we create a table $\mathbf{E}$ for all possible terminal nodes, we would end up with an enormous table $\mathbf{E}$. Instead, by considering the tokenization, we reduce the dimension of $\mathbf{E}$ because the sub-tokens can be repeated and shorter. We used `ronin` [35] to split the tokens, but other approaches could be used with comparable results (e.g., [15, 31]). Moreover, we remove integer literals from the output of $f_{cu}^{(X)}$ because their contribution to the classification is negligible and to further reduce the size of the $\mathbf{E}$ table. For simplicity, we also omit the processing of the method declaration, but information on parameters and return type is also included in the output.

An example. Figure 3 summarizes the process. A Java method code is shown at the top. Its AST is in the middle, and the extracted features are at the bottom. Colors highlight representations of different statements. For example, orange highlights the while statement, its AST, and its CUs. The AST is built by a Java parser. Here, f_v is used six times: twice with the root node of the orange sub-tree, twice with the root node of the blue sub-tree, and twice with the root node of the red sub-tree. Results of f_v are combined further on by using f_e.

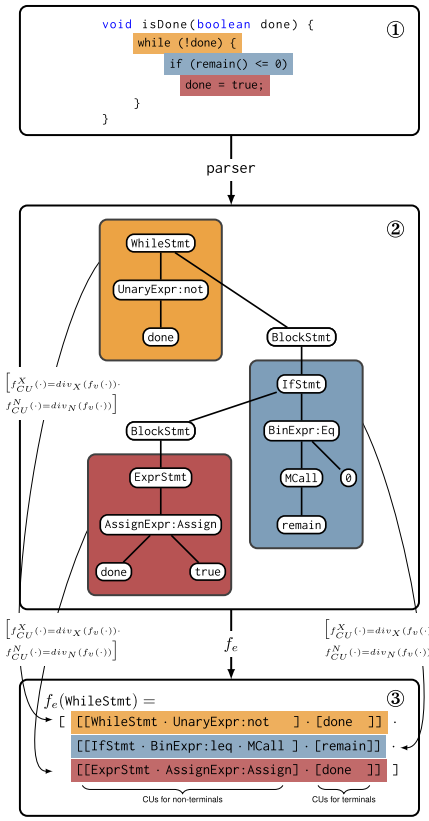


Fig. 3. The `isDone` code snippet (top) with its AST (middle) and extracted CUs (bottom).

3.2 Neural Model for Code Labeling

Overview. Figure 4 shows the adopted architecture for the NN. A statement (step ① in Figure 4) is processed as explained in Section 3.1 (step ② in Figure 4). In the case of multiple statements, the process shown in Figure 4 from ① to ❶ is repeated for each statement. Step ③ of Figure 4 shows the two sequences of tokens resulting from the feature extraction process.

Embeddings. Both terminal and non-terminal tokens are mapped into embeddings obtained from the vector table **E** of 10,000 entries of 100 floats (steps ④ and ⑧ in Figure 4). Table **E** is trained with back-propagation along with the rest of the network.

Terminals and non-terminals. Terminal and non-terminal tokens go through different paths. For terminal tokens, we use the multi-head GA layer (step ⑤ in Figure 4). Each terminal token receives a weight obtained through attention. All weights for each CU sum to 1. The higher the weight for a token, the more relevant is the token considered by the network. A summarized vector is obtained through a weighted sum. Finally, we apply the normalization layer from Ba et al. [9] (step ⑥ in Figure 4). Non-terminal tokens are processed through a RNN, 1StackBiLSTM (step ⑨ in Figure 4). To adopt GA, as used for the terminal tokens, would be undesirable in this case, because non-terminal tokens, unlike terminal ones, are more dependent on their position inside the statement. Using GA would lead to the loss of this positional information. Forward and backward states of the

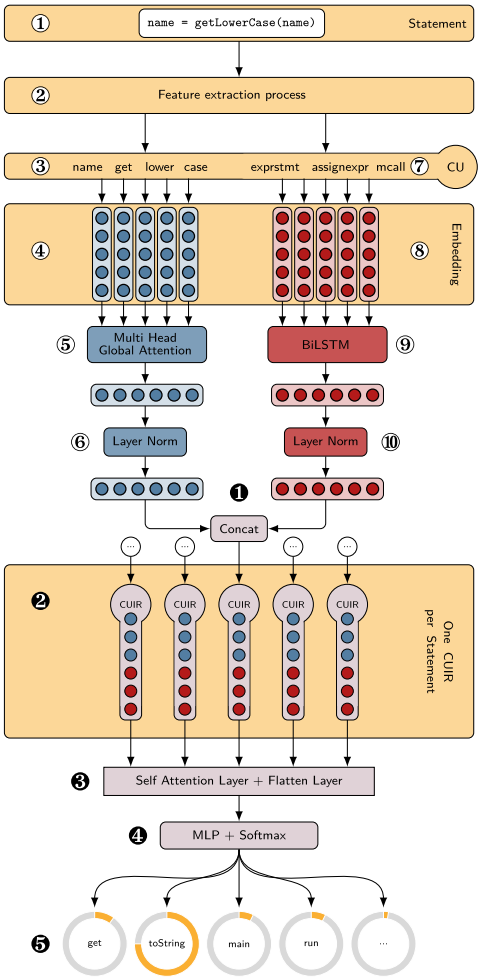


Fig. 4. NN architecture.

1StackBiLSTM are then summed up together. The resulting vector is the non-terminal intermediate representation after the application of a second normalization layer [9].

CU intermediate representation. Figure 4 (step ❷) emphasizes one of the **contextual unfolding intermediate representations (CUIRs)** obtainable by concatenating the terminal and non-terminal intermediate representations together (step ❶ in Figure 4). Terminals bring information from the domain in which the statement operates. For example, finding identifiers like *log* or *logging* tells us that the statement operates in the logging domain. If more than one domain appears in a single statement, then different attention heads will capture each domain. In contrast, non-terminals bring a different type of information. They provide information about the program control flow and how the terminals are used in the statement. An SA layer is applied once the CUIRs for all statements are computed (step ❸ in Figure 4). The CUIRs are put in relation to each other through the SA layer. An attention score is computed for each CUIR with respect to all of the other CUIRs. The normalized attention scores become the weights used to calculate a weighted sum of the CUIRs. The result is that each CUIR is mixed with the other CUIRs considered relevant. Then,

Table 1. Hyper-Parameter Values

Hyper-parameter	Value	Initial Configuration	Option Class
No. stmts.	15	20	5, 10, 15, 20, 25
No. token per stmt.	25	25	15, 20, 25
Batch size	1,000	10	10, 100, 1,000
Loss	Cross entropy	Cross entropy	–
Optimizer	Adam [44]	Adam [44]	–
Learning rate	0.001	0.001	–
Embed. size	100	100	100
Embed. vocabs	10^4	10^4	10^4, 10^5
CUIR size	$15 \cdot 300$	$15 \cdot 300$	–
MIR size	300	300	100, 150, 300
MLP activations	Relu	Relu	Tanh, relu
BiLSTM activations	Tanh	Tanh	–
LSTM type	1StackBiLSTM	LSTM	BiLSTM, LSTM, 2StackBiLSTM
No. global heads	1	2	1, 2, 3
GA act.	Tanh	Tanh, relu	Tanh
SA act.	Tanh	Tanh, relu	Tanh
Reg.	Layer norm [9]	Dropout	Layer norm [9], dropout

a vector of *2×embedding_size×|stmts|* features (from terminals and non-terminals) is obtained by flattening the SA output.

Final projections. A final MLP layer is applied to the resulting vector (step ❹ in Figure 4); this reduces the hidden representation to a vector of 300 entries, named $\vec{v}$. One final linear transformation is applied to bring $\vec{v}$ into the output space with 261,245 labels (step ❺ in Figure 4), and 261,245 is the same number of labels used by `code2vec` in the work of Alon et al. [7], which we use as a baseline. If the output is made up of $s_1, s_2, \ldots, s_{261,245}$, each s_i is the score given to the i-th label. The higher is s_i, the higher is the relevance of the i-th label. Initial labels are set to the method names with higher frequency in the training split of the `java-large` dataset [7], as discussed in Section 4. The last `softmax` layer normalizes the output so that the scores can be interpreted as probabilities. Our output is a list of labels sorted by probabilities.

Scaling with statements. This architecture was designed to exploit the effectiveness of transformer architecture [72] while reducing its memory requirement. Recall (from Section 2) that the SA mechanism has a memory requirement that grows quadratically in the number of tokens. In practice, this means that methods with more than 512 tokens are difficult to process with the SA mechanism. To avoid this issue, before applying SA, we encode each statement with a different and more memory-efficient technique. By doing so, we move the input of the SA layer from the bare code token to the statements encodings. Thus, the quadratic cost of the SA layer scales with the number of statements and not with the number of tokens.

3.3 Tuning Fold2vec

Table 1 summarizes the hyper-parameters of `fold2vec` and their values. Such hyper-parameters have been tuned over the validation split of the `java-large` dataset during a tuning phase. The tuning consists of manually and independently tweaking each hyper-parameter starting from the initial configuration shown in column 3 of Table 1. We kept all of the hyper-parameters that produce the best results in their option class (column 4 in Table 1, where a dash (–) indicates that

the parameter has not been tuned). The tuned hyper-parameters include layer size, regularization layer, number of global heads, batch size, activation functions, and input size. An exhaustive search of the best hyper-parameters is not feasible due to training times (we based our estimation on the results obtained after one epoch that required about 7 hours). Therefore, there is still space for improvement. We tested the model once we found the combination in Table 1. In addition, note that we had previously used convolutional NNs in an early stage of our work. However, we dropped them in favor of LSTMs, which led to better scores. Moreover, convolutional NNs were also used in the work of Allamanis et al. [3] with lower scores compared to more recent works [5].

4 EVALUATION

Introduction. To evaluate the proposed representation with the proposed architecture, we consider three different classification tasks: *code summarization*, *statements separation*, and *code search*. The first two tasks are evaluated using datasets derived from a collection of GitHub projects. As we will show, models trained for these tasks can be combined to build tools that could improve code readability. The third one uses a dataset collected by Husain et al. [36]. We evaluate our proposal against several baselines. Among these, several are completely replicated in our study to achieve a fair comparison. The only difference between the replicated baselines and the original ones is the used deep learning framework (originally TensorFlow,[1] and here, PyTorch[2]).

Hardware setup. All deep learning models are trained, evaluated, and tested on a single T4 NVIDIA GPU. This ensures complete and free reproducibility using services such as Google Colab[3] or Kaggle Kernels.[4] We provide a Google Colab notebook with the necessary steps to reproduce our study.[5]

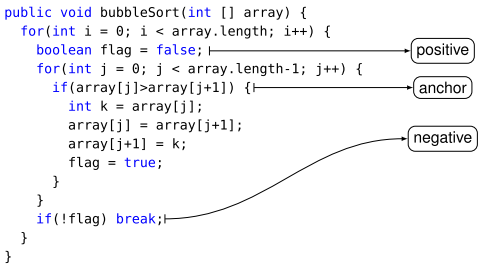
Baselines. To achieve a fair comparison, we focus on baselines that fall in these categories:

- *Non-autoregressive*: We consider only models that are non-autoregressive. Although autoregressive models may work better for some tasks, they are not always applicable. For example, binary classification tasks cannot be solved in an autoregressive manner.
- *End-to-end*: We consider only models that are not pre-trained. Although these models can achieve better results, they are also trained on additional data and for more time. To have a fair comparison, we believe that all models should be trained on the same dataset and for the same number of epochs.
- *Trained on* `java-large`: All models should be trained and tested on the same dataset.

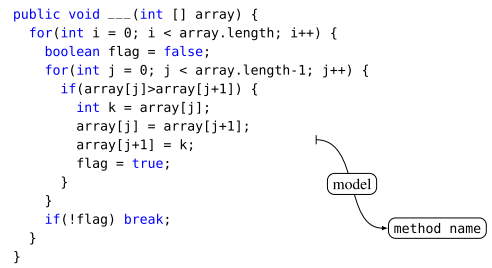
4.1 Code Summarization

Introduction. Summarizing code in meaningful short sentences is an extremely hard task. It can be a challenging and time-consuming task even for experienced programmers. Developing techniques that deal with this task can become useful tools in the developer's hands. The method name is the first thing a developer looks at to understand what the method does. Method names provide a relevant and brief description of code snippets. Moreover, each method has its own method name. Therefore, it is extremely easy to build massive datasets that incorporate several examples from various domains.

[1] https://www.tensorflow.org/.
[2] https://pytorch.org/.
[3] https://colab.research.google.com.
[4] https://www.kaggle.com.
[5] https://colab.research.google.com/drive/1y383wyfNemYO7QYlmp7Nh7L_IHSMPvo4?usp=sharing.



(a) A bubble sort method is shown. The first condition (`if(array[j]>array[j+1])`) is chosen as anchor statement. The second condition (`if(!flag)`) is chosen as negative statement wrt. the anchor. The variable declaration of `flag` (`boolean flag = false;`) is chosen as a positive statement wrt. the anchor.



(b) A bubble sort method is shown. Its name is obscured. Given only the information available from this snippet the model is asked to recover the original method name.

Fig. 5. A bubble sort method annotated for different tasks: statement separation (a) and code summarization (b).

Task. The model is presented with a method stripped of its name. The model should produce a name as close as possible to the original one. Figure 5(b) shows an example: a bubble sort method stripped of its name. The model is asked to retrieve the name just by looking at the method body.

Metrics. We adopt the same metrics used by Alon et al. [6, 7]: *precision*, *recall*, and *F1-score* (the higher the values of precision, recall, and F1-score, the better the model behaves) described in terms of **true positives (TPs)**, **false positives (FPs)**, and **false negatives (FNs)**:

- TP is the number of predicted sub-tokens that are also in the original name.
- FP is the number of predicted sub-tokens that are not in the original name.
- FN is the number of original sub-tokens that are not in the predicted name.

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \qquad F1 = \frac{2 * precision * recall}{precision + recall}$$

Dataset. The dataset (`java-large`[6]) used to train, evaluate, and test every considered model is the same used in the experiment reported in the work of Alon et al. [7]. This permits a fair comparison with `code2vec`, currently considered the state of the art for non-autoregressive models. Java-large[6] is a publicly available dataset composed of 16M samples. It is made up of 9,550 Java projects collected since 2007 among those top-starred on GitHub. The projects in the dataset have been used for training (9,000 projects), validating (300 projects), and testing (250 projects) the NN. All baselines use these exact splits both in the literature and in our study. Following the preprocessing used by Alon et al. [7], the training split of `java-large` is filtered to contain only the top 261,245 common method names. However, the test and validation split keeps all of their samples. This process generated the following dataset splits: 10,993,069 samples for training, 321,718 samples for validation, and 416,986 samples for testing. Additionally, a dataset with the same properties was generated with `code2vec` style leaf-to-leaf paths and `code2seq` style leaf-to-leaf paths.

Models. We report the results for six different models, which are briefly described as follows:

(1) With the name `Paths+CRFs`, we denote a conditional random field model trained in the work of Alon et al. [6]. Again, we only report results obtained in the previous study.

[6] https://s3.amazonaws.com/code2seq/datasets/java-large.tar.gz.

Table 2. Results for the Task Code Summarization

Model	Parameter (M)	TP	FP	FN	Precision	Recall	F1-score
Paths+CRFs [6]	–	–	–	–	32.56	20.37	25.06
HeMa [38]	–	270,057	901,178	358,232	23.06	42.98	30.02
code2vec	179	430,399	412,910	740,380	51.04	36.76	42.74
code'2vec	81	484,331	328,407	685,989	59.59	41.38	48.84
fold2vec	83	508,526	302,249	661,720	62.72	43.45	51.34
HeMa×fold2vec	83	518,177	315,187	652,365	62.18	44.27	51.72

For parameters, FP, and FN, the lower the better. For precision, recall, F1-score, and TP, the higher the better. All non-replicated results are accompanied with the respective citations. Last, "M" stands for millions.

(2) With the name fold2vec, we denote the proposed architecture with the proposed code representation.
(3) With the name code2vec, we denote a replication of the original experiment proposed by Alon et al. [5].
(4) With the name code'2vec, we denote an improved version of code2vec that employs the improvement proposed by Alon et al. [5] re-purposed for the non-autoregressive setting.
(5) With the name HeMa, we denote the model developed by Jiang et al. [38]. HeMa uses a set of handcrafted heuristics to produce a method name given its body. HeMa can classify only a few method categories (getters, setters, delegations, and methods that match a pre-defined set). On these methods, HeMa works exceptionally well. Although HeMa is non-autoregressive and is non-pre-trained, it is also not trained on java-large. As a matter of fact, HeMa is not trained at all. Therefore, following the conditions mentioned previously, HeMa cannot be compared to other baselines. However, we believe that this baseline can lead to an interesting model, HeMa×fold2vec.
(6) With the name HeMa×fold2vec, we denote a model produced by combining HeMa and fold2vec. This model is trained only on those samples in which HeMa fails to give a prediction. HeMa×fold2vec returns the HeMa prediction when available otherwise returns the fold2vec prediction.

Although models 1 and 5 are not directly replicated in this study, all of the other techniques were replicated to achieve maximum compatibility between training settings. In particular, models 2, 3, 4, and 6 are all trained to minimize the cross-entropy loss [20]:

$$\mathcal{L}(l, y) = -l_y + \log\left(\sum_i \exp(l_i)\right),$$

where l is the logits vector (the model output vector with size equal to the number of classes) and y is the correct class. In addition, all models are trained using the Adam optimizer [44] with a batch size of 100 and accumulated gradients for 10 steps. All chosen models predict in a non-autoregressive style, meaning that each prediction is based only on the input and not on previous, possibly iterated, predictions.

Results. Table 2 summarizes the results. code2vec achieves similar results to those reported by Alon et al. [7]. In particular, our replication differs in −0.89%, 0.38%, and 0.57% with respect to precision, recall, and F1-score when compared to the result reported in the work of Alon et al. [7]. These results independently confirm those reported by Alon et al. [7]. Additionally, the improvement of code'2vec over code2vec is noticeable. In fact, code'2vec uses far less trainable parameters (M less) and improves 8.55%, 4.62%, and 6.1% with respect to precision, recall, and F1-score when compared to its predecessor (code2vec). Although HeMa uses handcrafted heuristics and Paths+CRFs

use traditional machine learning techniques, both do not match the results obtained by the deep learning approaches. This fact shows how hard the task is of giving relevant method names to code snippets. The new representation alongside the proposed model is shown to be effective. We report an improvement of 11.68%, 6.69%, and 8.6% with respect to precision, recall, and F1-score against the previous best model (`code2vec`). The model that achieves the highest scores is `HeMa×fold2vec`. It surpasses its counterpart (`fold2vec`) in −0.54%, 0.82%, and 0.38% with respect to precision, recall, and F1-score. This fact shows that deep learning techniques with traditional approaches may have a positive effect.

4.2 Statements Separation

Introduction. The *code topical locality principle* [56] states that human-developed code is spatially organized. Close statements have a high probability to contribute to the same topic. We will train a model that tries to recognize if two statements are close. Assuming the truthfulness of the code topical principle, this model should be able to predict, with high probability, if two statements contribute to the same topic. Although it is extremely hard to measure how good a model is in distinguishing topically related statements, it is extremely easy to measure how good a model in distinguishing close to far apart statements.

Task. A model is presented with two statements. The model should predict whether the two statements were taken close (positive sample) or far apart (negative sample) in the source code. We impose two statements to be close if there is at most one statement between them. Figure 5(a) shows a bubble sort method. A statement named `anchor` is highlighted. Respectively to the anchor, both a positive and a negative statement are highlighted.

Metrics. We adopt common metrics used in literature for binary classification tasks. In particular, we show precision, recall, F1-score, and accuracy, which are described in terms of TPs, **true negatives (TNs)**, FPs, and FNs.

- TP is the number of nearby statements classified as nearby.
- TN is the number of far apart statements classified as far apart.
- FP is the number of nearby statements classified as far apart.
- FN is the number of far apart statements classified as nearby.

Although precision, recall, and F1-score are computed as shown previously, accuracy is computed as follows.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Dataset. We again used `java-large` to generate the dataset to train, validate, and test our models. Let us consider only the training split of `java-large`. For each statement, we sampled one positive sample and one negative sample. Positive samples are statements taken from a window of five statements around the anchor. Negative samples are statements that are taken either outside of this window or randomly from another method. This process generated the following splits from `java-large`: 75,910,619 samples for training, 2,816,578 samples for validating, and 4,768,550 samples for testing. A dataset with the same properties was generated with `code2vec`-style leaf-to-leaf paths and `code2seq`-style leaf-to-leaf paths.

Models. We designed and trained four separate models:

- `stmt-fold2vec`. `stmt-fold2vec` reuses most of the architecture of `fold2vec` to encode one statement at a time.

Table 3. Results for the Task Statement Separation

Model	Parameters (M)	TP	TN	FP	FN	Precision	Recall	F1-score	Accuracy
stmt-fold2vec	14	1,522,725	1,701,919	682,356	861,550	69.06	63.87	66.36	67.62
stmt-code2vec	107	1,460,976	1,617,295	766,980	923,299	65.57	61.28	63.35	64.55
stmt-code'2vec	19	1,546,965	1,623,206	761,069	837,310	67.03	64.88	65.94	66.48
stmt-BiLSTM	12	1,541,846	1,655,046	729,229	842,429	67.89	64.67	66.24	67.04

For parameters, FP, and FN, the lower the better. For precision, recall, F1-score, accuracy, TP, and TN, the higher the better.

- `stmt-code2vec`. `stmt-code2vec` applies the same architecture and representation for methods used by `code2vec` (leaf-to-leaf paths) to statements.
- `stmt-code'2vec`. `stmt-code'2vec` introduces the improvements introduced by `code2seq` in the context of `stmt-code2vec`.
- `stmt-BiLSTM`. `stmt-BiLSTM` uses a BiLSTM to encode statements. BiLSTMs are a standard architecture used to process sequences.

All models are trained on the mentioned training split to minimize the triple loss [10]:

$$\mathcal{L}(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + m, 0\},$$

where $d(x_i, y_i) = ||x_i, y_i||_2$ and $m = 1$. a represents an anchor statement, whereas p and n are a close statement (positive statement) and a far apart one (negative statement), respectively. Moreover, all models are trained with the same batch size of 100 samples and accumulated gradients for 10 steps. All models use the Adam optimizer [44] and take as input one statement and encode it into a vector of 100 floats.

Results. Table 3 summarizes the results. `stmt-fold2vec` is the model that achieves the best results, with an increment of 3.07% over `stmt-code2vec` and 1.14% over `stmt-code'2vec` in terms of accuracy. `fold2vec` can distinguish the positive samples from the negative ones the 67.62% of times. Nonetheless, all models achieve comparable results for this task.

4.3 Intent Complexity

Introduction. Over the years, the software engineering community has proposed a variety of software metrics to measure several aspects of code quality. In this section, we propose a software metric to measure the quality of source code with respect to the **single responsibility principle (SRP)** [49]. Our metric should be low for an SRP compliant method and should be higher otherwise.

Task. We will evaluate the proposed metric with respect to a simple binary classification task. A model is presented with a method, and it is asked to classify the method as positive or negative, where a positive method means an SRP-compliant method, whereas a negative method means a method with multiple responsibilities (which violates the SRP).

Metrics. We will evaluate the proposed models with respect to the following simple accuracy metric.

$$\frac{\#\text{correct prediction}}{\#\text{predictions}}$$

Dataset. To properly evaluate the extracted intents, we adopted an approach similar to Gu et al. [27]. We collected a simple dataset made of 100 manually annotated methods. These methods were collected and annotated using the following systematic approach (illustrated later in Figure 8):

Table 4. Ability of Several Source Code Metrics to Detect Tangled Concerns

Metric	Description	Accuracy
Di	Data Complexity	50%
Si	Structural Complexity	50%
Ci	System Complexity	50%
VG	no. Execution paths	55%
NVAR	no. Control variables	57%
NCOMP	no. Comparisons	60%
MCLC	no. Comparisons + no.control variables	62%
TLOC	no. Lines	71%
IC(ours)	Intent Complexity	80%

(1) We collected GitHub projects that use using the Spring framework Aspect-Oriented Programming.[7]
(2) We randomly sampled pointcuts (which describe (join) points in the program execution, like method calls/executions).
(3) Given a pointcut, we sampled the relative advice (which describes the effect at the described join point).
(4) Given a pointcut, we also sampled the method whose call/execution is captured by the pointcut (i.e., the method that is decorated (woven) with the advice code).
(5) The sampled method, by itself, represents a positive sample (shown later in Figure 8).
(6) The sampled method merged with the advice represents a negative sample (shown later in Figure 8).

At this point, we have 100 annotated samples (50 positives, 50 negatives). Positive samples are usually brief methods that accomplish a single responsibility (SRP compliant). Negative samples are usually longer and accomplish multiple responsibilities (non-SRP compliant).

Models. All models will be defined based on software engineering metrics. From each metric, we build a simple threshold classifier. The considered metrics are listed in Table 4. In addition to the established metrics, we propose the **intent complexity (IC)** metric. IC is based on the previously discussed task, statements separation. Recall that `stmt-fold2vec` encodes statements into $\mathbb{R}^{100}$ vectors (called *encodings*). We use a hierarchical clustering algorithm to group together statements based on this distance. We refer to these groups as intents. For example, Figure 6 shows a Java method with its detected intents highlighted and annotated with `fold2vec`. To obtain a score out of this set of intents, we measure the maximum Hausdorff distance between intents:

$$IC(I) = \max_{A,B \in I} \{d_H(A, B)\},$$

where I is a set of intents and d_H is the Hausdorff distance. Intuitively, the IC score is higher when a method covers many and broad topics. Meanwhile, it is low when a method covers few and similar topics.

Results. The average IC score on positive samples is 4.04. Meanwhile, the average IC score on negative samples is 6.87. Moreover, by tuning a simple threshold, one can separate positive from negative samples with an accuracy of 80%. In practice, this means that the IC score can be used to detect SRP violations. When compared to other classical software engineering metrics (see Table 4),

[7] https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop-api.

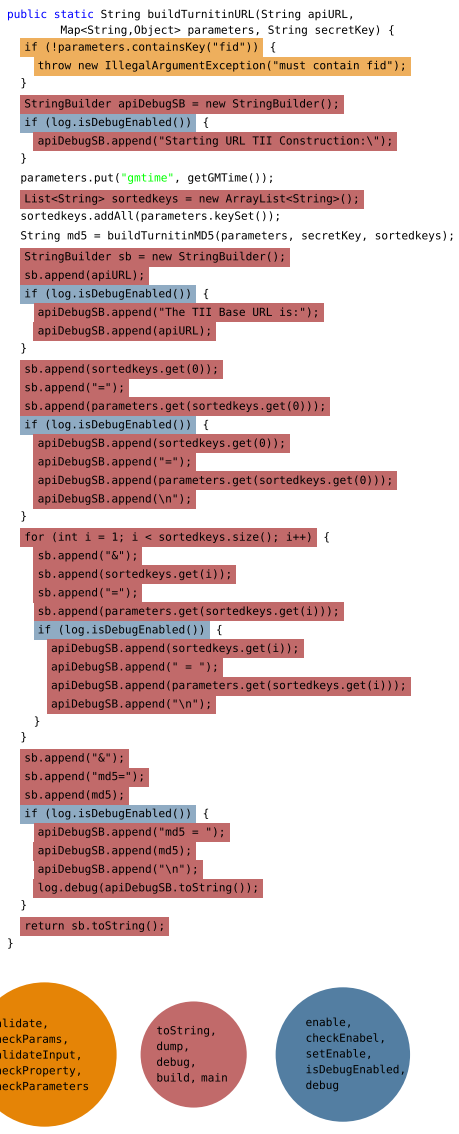


Fig. 6. On the top, statements are clustered according to the metric induced by `stmt-fold2vec`. On the bottom, clusters are summarized using `fold2vec` in the colored bubbles.

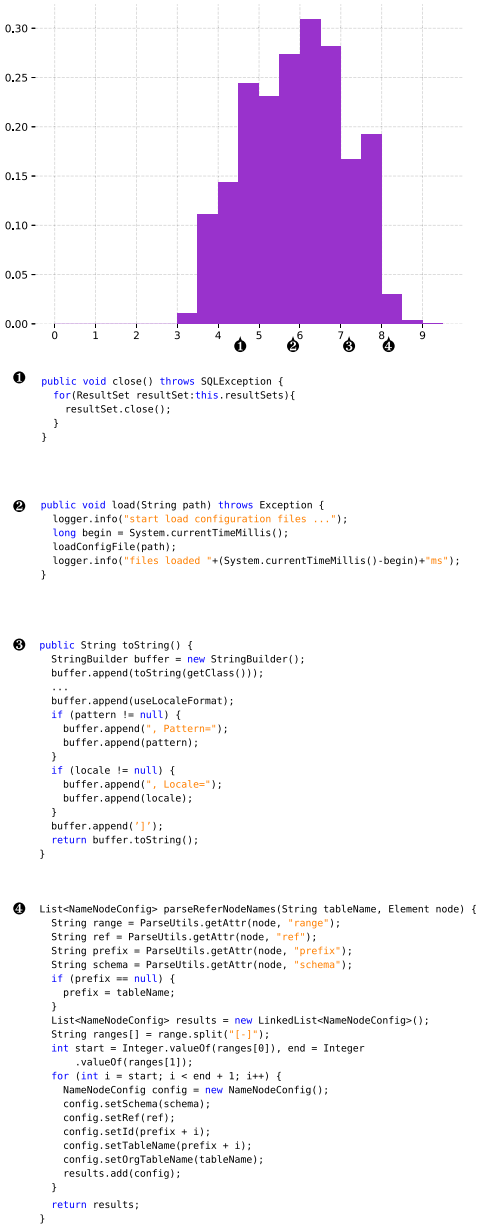


Fig. 7. On the top, the distribution of intent complexity scores (of the method with more than one intent) inside the test split of `java-large`. On the bottom, some examples organized according to their intent complexity score.

the IC score is the most accurate in detecting methods with multiple responsibilities. Additionally, Figure 7 shows how the IC score is distributed among methods found in the test split of `java-large`. Alongside the histogram, four methods are shown with an increasing IC score. The more the IC score increases, the more methods become longer and convoluted.

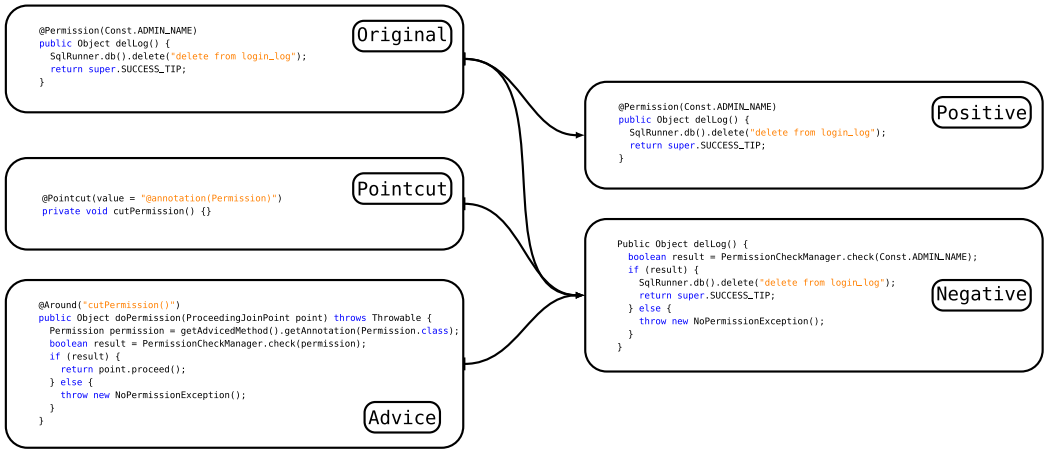


Fig. 8. On the left, a method, its pointcut, and its advice are shown. On the right, the original method by itself composes a positive example (i.e., a single, brief, and coherent concern). The original method combined with the advice composes a negative example (i.e., a method with mixed concerns).

4.4 Code Search

Introduction. The activity of programming often relies on exploiting information already available in other code bases [27]. Therefore, search engines capable of effectively filtering relevant information can have a tremendous impact on the development process. To fairly evaluate the proposed model with the state of the art, we followed the procedure described in the work of Husain et al. [36].

Task. A model takes as input a natural language query q and several code methods K=$\{k_1, k_2, \ldots, k_n\}$. The model should pair q with the most relevant method in K. In practice, the natural language query is represented by a documentation string. Instead, the most relevant method (with respect to q) is represented by the method that q is documenting.

Metrics. We evaluate the considered model on two metrics:

- *Top-1 accuracy (top-1)*: This represents the number of queries matched with their correct answers over the total number of queries.
- *Mean reciprocal rank*: **Mean reciprocal rank (MRR)** represents the mean of the reciprocal rank. The reciprocal rank of an answer is the reciprocal of the rank of the correct answer with respect to the query:

$$MRR = \frac{1}{Q}\sum_{i=1}^{|Q|}\frac{1}{rank_i},$$

  where Q is a set of queries and $rank_i$ is the rank of the correct answer with respect to the i-th query.

Dataset. We use the Java portion of the CodeSearchNet dataset [36]. The training split contains 454,450 samples. The validation split contains 15,327 samples. The test set contains 26,908 samples. Each sample is composed of the code and its documentation string.

Models. We trained and evaluated four models: `search-code2vec`, `search-code'2vec`, `search-fold2vec`, and `search-SA`. As such, `search-code2vec`, `search-code'2vec`, and

Table 5. Top-1 Accuracy and MRR Scores Are Shown for Each Model

Name	Parameters (M)	Top-1	MRR
`search-code2vec`	228	24.47	34.05
`search-code'2vec`	12	39.75	46.81
`search-SA`	13	46.78	56.02
`search-fold2vec`	13	48.8	56.05

For both top-1 and MRR, the higher the better.

`search-fold2vec` are based upon `code2vec`, `code2seq`, and `fold2vec`, respectively. Instead, `search-SA` is a replica (written using PyTorch instead of TensorFlow) of the best-performing model proposed in the work of Husain et al. [36]. All models are made of two parts: an encoder for query, E_q, and an encoder for method, E_c. All models are trained using the same loss function:

$$\mathcal{L}(Q) = -\frac{1}{N}\sum_i \log \frac{\exp(E_c(c_i)^T E_q(d_i))}{\sum_j \exp(E_c(c_j)^T E_q(d_i))},$$

where q_i represents the i-th query, c_i represents the correct answer, Q is the set of samples, and c_j, with $j \neq i$, represents incorrect answers (also called *distractors*). The result of minimizing this loss is the maximization of the inner product between q_i and c_i while minimizing the inner product between q_i and c_j. Now, we proceed by describing the evaluated models.

All models use the same query encoder E_q. E_q is made of three consecutive layers of SA. Meanwhile, E_c, the code encoder, is specialized for each model:

(1) `search-code2vec` uses the same encoding procedure used by `code2vec`.
(2) `search-code'2vec` uses the same encoding procedure used by `code2seq`.
(3) `search-fold2vec` encodes the source code as it is done in `fold2vec`.
(4) `search-SA` is the best-performing model used in the work of Husain et al. [36], as E_c, similarly to E_q, uses three consecutive layers of SA.

Result. Table 5 shows the scores obtained by each considered model. Models are ordered by their performance. Both `search-code2vec` and `search-code'2vec` fall behind on several points when compared to `search-SA` and `fold2vec`. Meanwhile, in terms of MRR, both `search-fold2vec` and `search-SA` behave almost identically. Instead, in terms of top-1 accuracy, `search-fold2vec` achieves the best score.

5 ABLATION STUDY

Introduction. In this section, to answer RQ_3, we are going to evaluate how the model `fold2vec` behaves with and without some of its parts. Moreover, we are going to study the dependence of `fold2vec` from code identifiers and how identifiers relate to the method name.

Identifier dependence. First, let us study the contribution of the two types of information (terminals and non-terminals) fed to `fold2vec`. Let us call `term2vec` the `fold2vec` variant stripped of the architecture path from ⑦ to ⑩ of Figure 4 and `non-term2vec` the `fold2vec` variant stripped of the architecture path from ③ to ⑥ of Figure 4. Table 6 summarizes the result. `term2vec` achieves a similar result to the complete architecture, whereas `non-term2vec` falls behind by several percentage points (−32.18%, −23.52%, and −27.22% with respect to precision, recall, and F1-score). This fact should not come as a surprise. Developers put a lot of effort into using relevant identifiers, and often one can guess the method behaviors just by looking at them. Moreover, without identifiers, all method calls look alike. Therefore, the information on what a sub-call is doing is lost. If we test

Table 6. Effect of Positional information within Statements on `fold2vec` Variants

Model	Precision (%)	Recall (%)	F1-score(%)
`term2vec`	59.88	40.84	48.56
`term2vec+PE`	59.97	40.46	48.32
`non-term2vec`	30.54	19.93	24.12
`non-term2vec+PE`	30.33	19.93	24.05
`fold2vec` (std.)	62.72	43.45	51.34
`fold2vec+PE`	62.54	42.9	50.89

`non-term2vec` on methods that do not perform other method calls (which are 120,551 samples), we can even raise the scores to 37.49%, 27.93%, and 32.01% with respect to precision, recall, and F1-score. Alongside method calls, information about types and reused variables are also lost, which explains the ongoing low result.

Method name token dependence. The previous paragraph showed and explained the dependence of `fold2vec` from code identifiers. However, not all identifiers are equal. In particular, `fold2vec` is dependent on a specific kind of identifier, those that appear in method names. Let us consider only methods that have among their code identifier tokens that appear in the method name itself. By doing so, we can build a subset of the test split composed of 155,030 samples. `fold2vec` evaluated on the set scores 67.04%, 48.32%, and 56.16% with respect to precision, recall, and F1-score, which represent an improvement of around 5 percentage points.

Positional embeddings. Let us consider the effect of **positional embeddings (PEs)** within statements (see Table 6). Notably, most of the time, PEs do not yield improvements and when they do, the improvements are marginal. This fact suggests that positional information is not a necessary component for the CUIRs. For `non-term2vec`, this should not come as a surprise since CUIRs are generated with a BiLSTM; however, the same appears to be true for `term2vec` and `fold2vec`.

LSTM variants. Next, consider Table 7. We used increasingly bigger LSTM layers processing non-terminals. Namely, we trained `fold2vec` using an LSTM, a BiLSTM, and a 2LayerBiLSTM for non-terminals. The best scores are achieved using a single layer of BiLSTM (51.34% with respect to F1-score). The 2LayerBiLSTM leads to almost identical results (51.31% with respect to F1-score). This study suggests that adding more BiLSTM layers does not lead to better results.

Terminals and non-terminal processing variants. To validate the different choices of using different architectures for terminals and non-terminals, consider Table 8. We evaluated three additional variants of `fold2vec`: `fold2vec+GA|GA` uses GA instead of the BiLSTM, `fold2vec+BiLSTM|GA` has the architecture for terminals swapped with the architecture for non-terminals, and `fold2vec+BiLSTM|BiLSTM` uses a BiLSTM instead of GA. The most noticeable result is that swapping architecture has a detrimental effect. When exclusively using BiLSTMs and GAs, a mild version of the same effect seems to appear. These facts seem to suggest that terminals and non-terminals can benefit from different architectural paths of execution, each leveraging specific properties of the CUs.

Not all methods are equal. Not all methods are equally easy to classify. In fact, some methods are much easier than others. Among these methods, we have getters and setters. Removing these methods from the test set has negative effect on the results: −11.35%, −9.65%, and, −10.57% with respect to precision, recall, and F1-score. This is mainly caused by two factors: the intrinsically

Table 7. Effect of LSTM Variants

Model	Precision (%)	Recall (%)	F1-score (%)
`fold2vec+LSTM`	62.17	43.03	50.86
`fold2vec+BiLSTM` (std.)	62.72	43.45	51.34
`fold2vec+2LayerBiLSTM`	62.59	43.47	51.31

Table 8. Effect of Different Models for the Two Representations, Where X|Y means X for terminals and Y for non-terminals

Model	Precision (%)	Recall (%)	F1-score (%)
`fold2vec+GA|GA`	62.35	42.9	50.83
`fold2vec+GA|BiLSTM` (std.)	62.72	43.45	51.34
`fold2vec+BiLSTM|GA`	61.48	42.62	50.34
`fold2vec+BiLSTM|BiLSTM`	62.25	43.21	51.01

hardness of getters and setters (which can be considered low), and the abundance of these methods in the dataset (which is high for `java-large`).

6 DISCUSSION

In this section, we discuss strengths and shortcomings of the considered approaches and the considered tasks.

6.1 Code Summarization

`HeMa` shortcomings. `HeMa` can give accurate method predictions for only three categories of methods: getters, setters, and delegations. These few categories cover a fairly large portion of the test set (32.89%). If one considers only these methods, `HeMa` achieves quite outstanding results: 70.26%, 77.49%, and 73.7% with respect to precision, recall, and F1-score. However, when one considers the full test set, `HeMa` performs rather poorly: 23.06%, 42.98%, and 30.02% with respect to precision, recall, and F1-score. This fact is also highlighted by Figure 9 snippets ❶ and ❷. `HeMa` is the only model that is unable to properly classify the shown factorial methods. Moreover, all methods predicted by `HeMa` are rather simple, and other techniques achieve high results. For example, `fold2vec` achieves 78.96%, 60.9%, and 68.76% when considering only `HeMa` predictable methods. Moreover, `HeMa` is also sensible to code modifications. Consider Figure 9 snippets ❸ and ❹; enclosing the return expression in parentheses causes a prediction to fail. Of course, this is just a limit of the handcrafted features of `HeMa` that cannot possibly cover all possible cases even for the limited scope of getters and setters.

`HeMa` strengths. Although `HeMa` has an extremely narrow scope, its predictions are accurate and cover a fairly large portion of the training set (58.15%). Combining `HeMa` with a deep learning technique like `fold2vec` means cutting down the training set of 58.15%, which greatly reduces training times without compromising the results.

`fold2vec` shortcomings. As was highlighted by the ablation study (Section 5), `fold2vec` is extremely dependent on code identifiers. In fact, removing identifiers reduce the scores of 32.18%, 23.52%, and 27.22% with respect to precision, recall, and F1-score. Consider snippets ❶ and ❷ in Figure 9, which shows two factorial methods. The method on the right has the identifier "`n`" changed to "`image`." Although this modification keeps both methods equivalent, it is enough to compromise the classification.

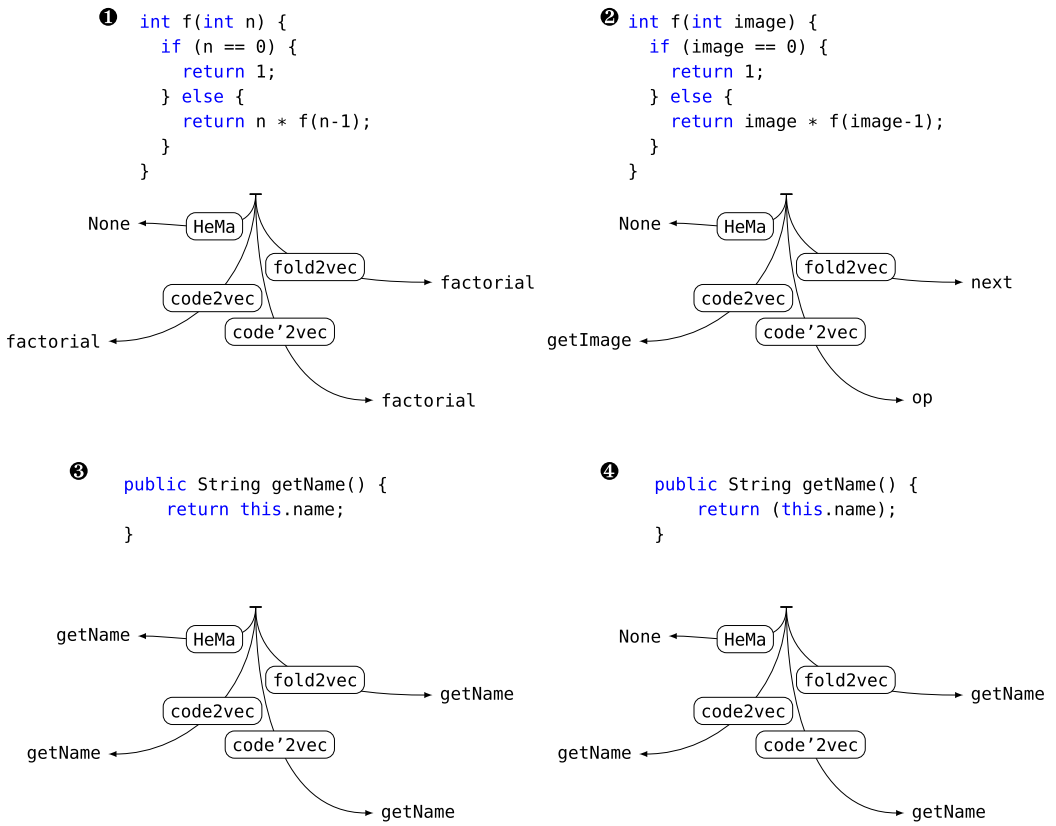


Fig. 9. Four methods with relative predictions (`HeMa`, `fold2vec`, `code2vec`, and `code'2vec`) are displayed. ❶ and ❷ display a factorial method with different variable identifiers. ❸ and ❹ display a getter with different yet equivalent return expressions.

`fold2vec` strengths. As mentioned in Section 2, the SA layer has a memory footprint that grows quadratically with the sequence length. Although processing each token in the method through SA is probably beneficial, it is also extremely costly in terms of memory. `fold2vec` can cut down this cost by applying the SA only to the statement-level vectors. When compared to other transformer architectures such as `CodeBERT` [25], `fold2vec` requires much less computational power. `fold2vec` requirements are small enough that the whole experiment can be reproduced by using free services as Google Colab and Kaggle Kernels.

`code2vec` shortcomings. `code2vec` stores into embedding matrices not only terminals and non-terminals tokens but also sequences of non-terminals (the mentioned leaf-to-leaf paths). In fact, leaf-to-leaf paths are stored and trained as embedding vectors. This greatly affects the memory requirements of the architecture. Consider that `code2vec` has 116 M trainable parameters, whereas its counterpart `code2seq` has only 37 M trainable parameters. Like `fold2vec`, `code2vec` also depends heavily on identifiers. Again, this dependence results in misclassifications like the one shown in snippets ❶ and ❷ in Figure 9.

`code2vec` strengths. `code2vec` is an extremely shallow yet effective network, thus it can efficiently make predictions. `code2vec` is the model that achieves faster throughput (on a i7-10700K

CPU @ 3.80 GHz) of 1641.7 methods per second. Even when compared to its counterpart, `code2seq`, `code2vec` is faster. In fact, `code2seq` achieves only 135.3 methods per second. Like `fold2vec`, `code2vec` can be trained using a single P100 NVIDIA GPU with 16 GB of memory. Thus, it can be trained without any cost using the mentioned services (Google Colab and Kaggle Kernels).

`code'2vec` shortcomings. `code'2vec` is by far the slowest model. It takes about 6 hours to complete a training epoch. Meanwhile, `fold2vec` and `code2vec` complete an epoch in about 2 hours. This reflects also only on the prediction throughput of 186.2 methods per second. Meanwhile, `fold2vec` achieves 1544.4 and `code2vec` achieves 1641.7 methods per second.

`code'2vec` strengths. Although `code'2vec` is quite slower, it addresses and solves several of the problems mentioned for `code2vec`. First, instead of storing the most common leaf-to-leaf, they are processed token by token (using a bidirectional LSTM), reducing the need for the embedding matrix. As a result, `code'2vec` uses far less trainable parameters (only 81 M). Nonetheless, although this approach is more effective and more memory efficient, it is also slower.

6.2 Statement Separation

`stmt-fold2vec` shortcomings. As was previously highlighted, `fold2vec` is sensible to identifier changes. Although these changes may leave the semantics unaltered, they heavily affect the model predictions. `stmt-fold2vec` inherits the same issue.

`stmt-fold2vec` strengths. `fold2vec` representation is designed to separately process statements into neural hidden representations. `stmt-fold2vec` exploits the same representation to generate semantically relevant statement-level encodings and achieves the best results.

`stmt-code2vec` shortcomings. Like the other deep learning techniques, `stmt-code2vec` manifests dependence on code identifiers. Moreover, the leaf-to-leaf representation used by `code2vec` cannot be applied directly to statements without adjustments. For example, consider snippet ❶ in Figure 10. `code2vec` leaf-to-leaf paths would normally require two different leaves to be computed. However, a statement like "**`return`** `name;`" has only one leaf. Moreover, statements without leaf identifiers can occur as well. One example is the plain "**`return;`**" statement as shown in snippet ❷ in Figure 10. Ultimately, `code2vec` representation needs to be adapted to deal with corner cases that were not possible when considering only methods.

`stmt-code2vec` strengths. Once again, `stmt-code2vec` tops in terms of prediction throughput. However, our approach (`stmt-fold2vec`) can predict 11380.8 statement per section. `stmt-code2vec` practically doubles this number, achieving 22282.9 statements per section.

`stmt-code'2vec` shortcomings. As mentioned previously, `stmt-code'2vec` is quite slow in both training and evaluating. For example, completing a training epoch requires around 17 hours. Meanwhile, `stmt-code2vec` requires around 7 hours.

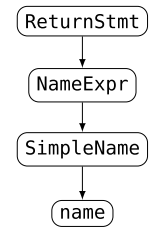
`stmt-code'2vec` strengths. Again, `stmt-code'2vec` retains the strengths discussed for the code summarization task: it improves in terms of scores and in terms of memory efficiency.

6.3 Code Search

`search-SA` strengths. When compared to the other architectures, this has several advantages. SA has become a popular component in many architectures. Thus, it is already available in most of

❶

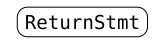
```
return name;
```



code2vec-styled : `name:SimpleName↑NameExpr↑SimpleName↑ReturnStmt↓NameExpr↓SimpleName:name`
code2seq-styled : `name:SimpleName,NameExpr,SimpleName,ReturnStmt,NameExpr,SimpleName:name`

❷

```
return;
```



code2vec-styled : `ReturnStmt:ReturnStmt:ReturnStmt`
code2seq-styled : `ReturnStmt:ReturnStmt:ReturnStmt`

Fig. 10. ❶: On the left, the statement **`return`** `name;` is shown. On the right, its AST is shown. On the bottom, our interpretation of leaf-to-leaf paths for this case is shown. ❷: On the left, the statement **`return;`** is shown. On the right, its AST is shown. On the bottom, our interpretation of leaf-to-leaf paths for this case is shown.

the popular deep learning libraries (e.g., TensorFlow,[8] PyTorch[9]). Additionally, `search-SA` treats source code as a simple text string, meaning that (1) it does not require the input to be parsable, and (2) it can be easily applied to many programming languages.

`search-SA` shortcomings. Although `search-SA` performs very well, it may still be beneficial to have information based on the AST. Additionally, given the quadratic memory cost of SA layers, `search-SA` has higher memory requirements than its counterparts. Moreover, `search-SA` is one of the slowest architectures at inference time (112 samples/second). Meanwhile, `search-code2vec` and `search-fold2vec` achieve 217 and 186 samples/second, respectively. Finally, as mentioned for `fold2vec`, `search-SA` is still heavily dependent on source code identifiers.

Other models. Finally, it is worth noting that the strengths and weaknesses of the other models (`fold2vec`, `code2vec`, and `code'2vec`) previously noted appear for the code search task as well.

6.4 Intent Complexity

We compared the IC score with respect to a specific aspect of software quality (i.e., the SRP). However, the quality of source code should be evaluated considering several aspects, such as the mentioned SRP, quality of identifiers, and complexity of the method, and many other aspects. Although our evaluation tells us that the IC score can be used to measure the adherence of a method to the SRP, it should be combined with other metrics to measure the overall quality of the code.

[8] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Attention.
[9] https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html.

6.5 Research Questions

In Section 1, we proposed three research questions, which we are ready to answer:

***RQ$_1$*:** Can NNs for code comprehension benefit from a statement-based code representation?
We evaluated the proposed representation against three baselines (`code2vec`, `code'2vec`, and `search-SA`) using different source code representations with three different tasks (code summarization, statement separation, and code search). In all considered cases, the statement-based representation showed improvements. We can conclude that source code statements can be compressed in a concise hidden representation that can be effectively used by the deeper layers of a NN. This has another main benefit: the SA layer scales quadratically in the number of statements and not in the number of source code tokens, which greatly reduces the memory footprint of the layer without compromising the results.

***RQ$_2$*:** Can such a code representation enable a NN to detect code misconduct?
We showed that a statement-based code representation can be used to train NNs to cluster statements into coherent snippets. We show that the induced distance between snippets can detect tangled responsibilities with a success rate of 80% using a simple threshold.

***RQ$_3$*:** Which neural component with which neural representation behaves the best?
We evaluated `fold2vec` under different conditions. Most notably, identifiers represent the information that is needed the most. In addition, the most important identifiers are those that mimic the method name. However, other parts seem to have little influence. For example, the presence of PE or the number of LSTM layers does not heavily impact the scores.

6.6 Threats to Validity

In this section, we present the threats to validity that may have affected the evaluation.

External validity. All networks were evaluated only on one dataset processed from `java-large`. This involves a risk relative to a measurement bias since our model could behave particularly well only on this dataset. However, this risk is mitigated by the fact that `java-large` is a huge dataset made up of numerous Java projects from several different domains.

Internal validity. The initial dataset is composed of single `.java` files. We transformed these files into the used dataset by using a parser and a tokenizer. We ended up with differences in the used dataset by adopting a parser and a tokenizer different from those used by the other models. These differences can negatively affect the estimation of the measurements. To mitigate this issue, we replicated the work of Alon et al. [7] using our framework. Nonetheless, by replicating `code2vec`, we may have inadvertently introduced errors or misimplemented some parts. However, the replicated models achieved results similar to the original one. Thus, we are confident of the validity of the comparison.

Construct validity. Due to the difference in representation (between `code2vec` and `fold2vec`), there are also differences in the network architecture. It is possible that the better results are due more to the architectural differences than to the difference in representation. To mitigate the problem, we restricted the comparison to models with strong similarities: only non-autoregressive models and only end-to-end models.

7 RELATED WORKS

In this section, we examine approaches that share the same goals, techniques, or application domain as `fold2vec`.

Code summarization. In the non-autoregressive family of models, we can enumerate `code2vec` [7] and `Paths+CRFs` [6], which we have already discussed extensively. Autoregressive models produce a sequence of predictions where each prediction is based on the previous one. Even if these are promising approaches, they better fit when the classification can be split into multiple steps. `code2seq` [5] and `ConvAttention` [3] fall into this category. The former is, once again, based on the leaf-to-leaf path representation, whereas the latter is based on source code tokenization. Both use NNs to achieve their purpose: attention based and convolution based, respectively. Although we focused on deep learning methods, there are other techniques using more traditional approaches. For example, Hellendoorn and Devanbu [30] use an n-gram-based language model. Raychev et al. [64] use support vector machines constrained to predict only few method names. Wang et al. [74] train an autoregressive model using reinforcement learning using two different datasets for Python and Java code.

Intent identification. Since intents and concerns can be overlapping concepts, in this section we present some work on concern identification. Kästner et al. [42] proposed a semi-automatic tool for extracting features from code. It relies on a configuration given from a domain expert to correctly recognize the features. In the same research area, Valente et al. [71] proposed an approach to semi-automatically annotate optional features. UML model variants have been used to automatically identify model-based product lines in the work of Martinez et al. [50]. Qu and Liu [62], Breu and Krinke [13], and Tonella and Ceccato [70] proposed approaches based on program tracing to automatically mine concerns. Moldovan and Şerban [53] proposed a clustering-based approach. All of these are semi-automatic or manually driven approaches to feature identification, whereas `fold2vec` is automatic; however, apart from this aspect, they share the same goal.

Machine learning on code. Allamanis and Sutton [4] presented an early analysis of source code based on n-gram language models. Their results have been used to build a Java-source code dataset that has been used as the basis to build the `java-large` dataset we use. From the same research group, graph NNs were applied for code generation and representation learning [2, 14]. The task of code generation is tackled in the work of Hu et al. [33] using transformer-generated representation from AST paths. For the same task, a massive transformer language model developed in the work of Chen et al. [17] achieved outstanding results. CodeBERT uses a large transformer language model for the task of mask language modeling [25]. Pre-trained models of the size of CodeBERT usually have high training costs. However, these models can be generalized to several tasks through fine-tuning with lower cost (both in terms of data and hardware). Usually, these large models are resource heavy and are accessible through online APIs. The main benefit of this approach is that it leads to better results. Again, for representation learning, Wang et al. [75, 76] used graph NNs and program traces to achieve the same scope. Instead, Ben-Nun et al. [11] used recurrent architectures trained from features extracted from the LLVM representation of source code. Raychev et al. [65] exploited CRF on a dependency network built from JavaScript code to automatically predict program properties such as type annotation and variable identifiers. Their approach led to `JSNice`, which predicts correct name identifiers in 63% of the cases and correct type annotations in 81% of the cases. Hu et al. [34], Chen and Zhou [18], and Movshovitz-Attias and Cohen [54] developed autoregressive models to automatically generate comments from source code. Iyer et al. [37] used attention networks to generate natural language descriptions from code, and Haiduc et al. [28] shared the same goal. Jiang et al. [40] used NN to automatically generate commit messages from diffs. Piech et al. [61] translated programs into real valued embeddings. Chen et al. [19] used tree decoders and encoders to translate programs from one language to another. Oda et al. [57] tried to translate formal code into pseudo code. Another, interesting research field applies machine learning techniques to spot bugs, as in the work of Dam et al. [22] and

Shi et al. [68]. Related to bug detection, much work focuses on program repair. For example, Jiang et al. [39] used a transformer-based architecture (GPT) to pre-train a large language model for the task. For the same task, a recurrent neural architecture was used by Wang et al. [73]. All of these approaches exploit code representation and machine learning, and represent potential application domains for `fold2vec`. A different type of code representation based on code updates was developed by Hoang et al. [32]. Last, Kang et al. [41], Keim et al. [43], and Rabin et al. [63] assessed generalizability of several deep learning baselines as `code2vec`, `code2seq`, and CodeBERT.

8 CONCLUSION AND FUTURE WORK

This work introduced a source code representation to be used with NNs that differs from traditional approaches for its granularity. On such a representation, we developed the `fold2vec` model and compared it to the state-of-the-art techniques for the task of code search, code summarization, and statements separation. Such a representation could also bring an improvement in other code tasks, such as automatic defect detection and automatic comment generation. We addressed and answered RQ_1 by showing improvements on the task of code summarization, code search, and statements separation. Moreover, by addressing RQ_2, we showed that a NN trained on the statements separation task can be used to measure the code quality with respect to the SRP. To answer RQ_3, we evaluated several variants of `fold2vec`. It appears that the different properties of terminal and non-terminal can potentially benefit from tailored architectures. Both the model and data needed to reproduce our experiment are available at the following URL:

https://cazzola.di.unimi.it/fold2vec.html.

Additionally, we provide a Google Colab notebook to replicate our experiments:

https://colab.research.google.com/drive/1y383wyfNemYO7QYlmp7Nh7L_IHSMPvo4?usp=sharing.

In the future, we will move `fold2vec` to the autoregressive model family and compare it to `code2seq`. We must note that other valid directions are positional-like embedding that encode the parent-child relations between tokens [58]. In addition, Tree-LSTMs [69] could be applied for both inter-statements and intra-statements. Another available approach is to introduce an auto-encoder for statements to be either pre-trained or trained altogether with `fold2vec`.

ACKNOWLEDGMENTS

The authors wish to thank Uri Alon for his precious suggestions and cooperation and Sudipto Ghosh for his thorough proofreading and comments.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 38–49.

[2] Miltiadis Allamanis, Marc Brockschmmmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*. Vancouver, BC, Canada.

[3] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33rd International Conference on Machine Learning (ICML'16)*. PMLR, New York, NY, USA, 2091–2100.

[4] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*. IEEE, Los Alamitos, CA, 207–216.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. `code2seq`: Generating sequences from structured representations of code. In *Proceedings of the 7th International Conference on Learning Representations (ICLR'19)*. New Orleans, LA, USA.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 404–419.
[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. `code2vec`: Learning distributed representations of code. In *Proceedings of the 46th Annual Symposium on Principles of Programming Languages (POPL'19)*. ACM, New York, NY.
[8] Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. 2017. Neural attribute machines for programming generation. *arXiv e-prints arXiv:1705.09231* (2017).
[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hilton. 2016. Layer normalization. *ArXiv e-prints arXiv:1607.06450* (2016).
[10] Vassileios Balntas, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. 2016. Learning local feature descriptors with triplets and shallow convolutional neural networks. In *Proceedings of the British Machine Vision Conference (BMVC'16)*. Article 119, 11 pages.
[11] Tal Ben-Nun, Alice Shashana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. 3589–3601.
[12] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transaction on Neural Networks* 5, 2 (March 1994), 157–166.
[13] Silvia Breu and Jens Krinke. 2004. Aspect mining using event traces. In *Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE'04)*. IEEE, Los Alamitos, CA, 310–315.
[14] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative code modeling with graphs. In *Proceedings of the 7th International Conference on Learning Representations (ICLR'19, poster session)*. New Orleans, LA, USA.
[15] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. In *ECOOP 2011—Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 6813. Springer, 130–154.
[16] Ahmet Celik, Pai Sreepathi, Sarfraz Khurshid, and Milos Gligoric. 2017. Bounded exhaustive test-input generation on GPUs. In *Proceedings of the 32nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*. ACM, New York, NY.
[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. 2021. Evaluating large language models trained on code. *arXiv e-prints arXiv:2107.03374* (2021).
[18] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. ACM, New York, NY, 826–831.
[19] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS'18)*. 2552–2562.
[20] David R. Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society* 20, 2 (July 1958), 215–232.
[21] Maureen Cudill. 1987. Neural networks primer, part I. *AI Expert* 2, 12 (Dec. 1987), 46–52.
[22] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Tryuen Tran, John Grundy, Aditya Ghose, Kim Taeksu, and Chul-Joo Kim. 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR'19)*. IEEE, Los Alamitos, CA, 46–57.
[23] Alaistair Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Bounded exhaustive test-input generation on GPUs. In *Proceedings of the 32nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*. ACM, New York, NY.
[24] Vijay Prakash Dwivedi and Manish Shrivastava. 2017. Beyond Word2Vec: Embedding words and phrases in same vector space. In *Proceedings of the 14th International Conference on Natural Language Processing (ICON'17)*. 205–211.
[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP'20)*. Association for Computational Linguistics, 1536–1547.
[26] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. 2000. Learning to forget: Continual prediction with LSTM. *Neural Computation* 12, 10 (Oct. 2000), 451–2471.
[27] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 933–944.
[28] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. IEEE, Los Alamitos, CA, 223–226.

[29] Alon Halevy, Peter Norvig, and Fernando Pereira. 2009. The unreasonable effectiveness of data. *IEEE Intelligent Systems* 24 (March/April 2009), 8–12.

[30] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural network the best choice for modeling source code? In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 763–773.

[31] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vujay-Shanker. 2014. An empirical study of identifier splitting techniques. *Empirical Software Engineering* 19, 6 (Dec. 2014), 1754–1780.

[32] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*. ACM, New York, NY, 518–529.

[33] Han Hu, Qiuyuan Chen, and Zhaoyi Liu. 2019. Code generation from supervised code embeddings. In *Proceedings of the 26th International Conference on Neural Information Processing (ICONIP'19)*. Communications in Computer and Information Science, Vol. 1142, Tom Gedeon, Kok Wai Wong, and Minho Lee (Eds.). Springer, Sydney, Australia, 388–396.

[34] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18)*. ACM, New York, NY, 200–210.

[35] Michael Hucka. 2018. Spiral: Splitters for identifiers in source code files. *Journal of Open Source Software* 3, 24 (April 2018), Article 653, 3 pages.

[36] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[37] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16)*. 2073–2083.

[38] Lin Jiang, Huai Liu, and He Jiang. 2019. Machine learning based recommendation of method names: How far are we? In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE'19)*. IEEE, Los Alamitos, CA, 602–614.

[39] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering (ICSE'21)*. IEEE, Los Alamitos, CA, 1161–1173.

[40] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, Los Alamitos, CA, 135–146.

[41] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. 2019. Assessing the generalizability of code2vec token embeddings. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE'19)*. IEEE, Los Alamitos, CA, 1–12.

[42] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering* 40, 1 (Jan. 2014), 67–82.

[43] Jan Keim, Angelika Kaplan, Anne Koziolek, and Mehdi Mirakhorli. 2020. Does BERT understand code?—An exploratory study on the detection of architectural tactics in code. In *Software Architecture*. Lecture Notes in Computer Science, Vol. 12292. Springer, 220–228.

[44] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR'15)*. San Diego, CA, USA.

[45] Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. 2015. Skip-thought vectors. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. MIT Press, Montréal, Canada, 3294–3302.

[46] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2017. Data-driven program completion. *arXiv e-prints arXiv:1705.09042* (2017).

[47] Minh-Thang Luong, Hicu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP'15)*. Association for Computational Linguistics, Lisbon, Portugal, 1412–1421.

[48] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on Machine Learning (ICML'14)*. PMLR, Beijing, China, 649–657.

[49] Robert C. Martin, James W. Newkirk, and Robert S. Koss. 2003. *Agile Software Development: Principles, Patterns and Practices*. Prentice Hall, Upper Saddle River, NJ.

[50] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves le Traon. 2015. Automating the extraction of model-based software product lines from model variants. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, Los Alamitos, CA, 396–406.

[51] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*. Curran Associates Inc., Lake Tahoe, NV, USA, 3111–3119.

[52] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, New York, NY, 997–1016.

[53] Grigoreta Sofia Moldovan and Gabriela Şerban. 2006. Aspect mining using a vector-space model based clustering approach. In *Proceedings of the 2nd Workshop on Linking Aspect Technology and Evolution (LATE'06)*. CWI, Bonn, Germany, 36–40.

[54] Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL'13)*. Association for Computational Linguistics, Sofia, Bulgaria, 35–40.

[55] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermain. 2018. Neural sketch learning for conditional program generation. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*, Vancouver, BC, Canada.

[56] Nan Niu, Juha Savolainen, Tanmay Bhowmik, Anas Mahmoud, and Sandeep Reddivari. 2012. A framework for examining topical locality in object-oriented software. In *Proceedings of the 36th International Conference on Computer Software and Applications (COMPSAC'12)*. IEEE, Los Alamitos, CA, 219–224.

[57] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, Los Alamitos, CA, 574–584.

[58] Yutaro Omote, Akihiro Tamura, and Takashi Ninomiya. 2019. Dependency-based relative positional encoding for transformer NMT. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANPL'19)*. Varna, Bulgaria, 854–861.

[59] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*. Atlanta, GA, USA, III-1310–III-1318.

[60] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP'14)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543.

[61] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. PMLR, Lille, France, 1093–1102.

[62] Liping Qu and Daxin Liu. 2007. Extending dynamic aspect mining using formal concept analysis. In *Proceedings of the 4th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'07)*. IEEE, Los Alamitos, CA, 564–567.

[63] Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (Feb. 2021), 106552.

[64] Rafiqul Islam Rabin, Arjun Mukherjee, Omprakash Gnawali, and Mohammad Amin Alipour. 2020. Towards demystifying dimensions of source code embeddings. In *Proceedings of the 1st International Workshop on Representation Learning for Software Engineering and Program Languages (RL+SE&PL'20)*. ACM, New York, NY, 29–38.

[65] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from 'big code.' In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages (POPL'15)* ACM, New York, NY, 111–124.

[66] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 419–428.

[67] Haoyue Shi, Hao Zhou, Jiaze Chen, and Lei Li. 2018. On tree-based neural sentence modeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP'18)*. Association for Computational Linguistics, Brussels, Belgium, 4631–4641.

[68] Ke Shi, Yang Lu, Jingfei Chang, and Zhen Weu. 2020. PathPair2Vec: An AST path pair-based code representation method for defect prediction. *Journal of Computer Languages* 59 (Aug. 2020), 100979.

[69] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (ACL/IJCNLP'15)*. 1556–1566.

[70] Paolo Tonella and Mariano Ceccato. 2004. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*. IEEE, Los Alamitos, CA, 112–121.

[71] Marco Tullio Valente, Virgilio Borges, and Leonardo Passos. 2012. A semi-automatic approach for extracting software product-lines. *IEEE Transactions on Software Engineering* 38, 4 (July–Aug. 2012), 737–754.

[72] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan M. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates, Inc., Long Beach, CA, USA, 6000–6010.

[73] Ke Wang, Rishabh Sing, and Zhendong Su. 2018. Dynamic neural program embedding for program repair. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*, Vancouver, BC, Canada.

[74] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2017. Linformer: Self-attention with linear complexity. *arXiv e-prints arXiv:2006.04768v3* (2017).

[75] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Yu, Philip Yu, and Guandong Xy. 2022. Reinforcement-learning-guided source code summarization via hierarchical attention. *IEEE Transactions on Software Engineering* 48, 1 (2022), 102–119.

[76] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. In *Proceedings of the 35th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20)*. ACM, New York, NY, 1–27.

Received 4 May 2021; revised 20 January 2022; accepted 26 January 2022