Machine Learning Based Recommendation of Method Names: How Far Are We

Lin Jiang
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
jianglin17@bit.edu.cn

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

He Jiang
School of Software Technology, Dalian University of Technology
Dalian, China
jianghe@dlut.edu.cn

***Abstract*—High quality method names are critical for the readability and maintainability of programs. However, constructing concise and consistent method names is often challenging, especially for inexperienced developers. To this end, advanced machine learning techniques have been recently leveraged to recommend method names automatically for given method bodies/implementation. Recent large-scale evaluations also suggest that such approaches are accurate. However, little is known about where and why such approaches work or don't work. To figure out the state of the art as well as the rationale for the success/failure, in this paper we conduct an empirical study on the state-of-the-art approach *code2vec*. We assess *code2vec* on a new dataset with more realistic settings. Our evaluation results suggest that although switching to new dataset does not significantly influence the performance, more realistic settings do significantly reduce the performance of *code2vec*. Further analysis on the successfully recommended method names also reveals the following findings: 1) around half (48.3%) of the accepted recommendations are made on getter/setter methods; 2) a large portion (19.2%) of the successfully recommended method names could be copied from the given bodies. To further validate its usefulness, we ask developers to manually score the difficulty in naming methods they developed. *Code2vec* is then applied to such manually scored methods to evaluate how often it works in need. Our evaluation results suggest that *code2vec* rarely works when it is really needed. Finally, to intuitively reveal the state of the art and to investigate the possibility of designing simple and straightforward alternative approaches, we propose a heuristics based approach to recommending method names. Evaluation results on large-scale dataset suggest that this simple heuristics-based approach significantly outperforms the state-of-the-art machine learning based approach, improving precision and recall by 65.25% and 22.45%, respectively. The comparison suggests that machine learning based recommendation of method names may still have a long way to go.**

***Index Terms*—Code Recommendation, Machine Learning**

I. Introduction

Identifiers are widely employed to identify unique software entities. According to a recent study [1], identifiers account for approximately 70% of the source code in terms of characters. A well-constructed identifier not only follows language-specific naming conventions, but also conveys intention/responsibility of its associated software entity [2]. Consequently, high quality identifiers have significant influence on the readability of source code [3], [4].

*Corresponding author

Method names are a special kind of identifiers, employed to identify methods. Methods are the smallest named units of aggregated behaviors, and serve as a cornerstone of abstraction [5]. The readability of such methods is critical in understanding the functionality of programs and the interaction among different units (methods). However, constructing high quality method names is often challenging, especially for inexperienced developers [1].

To facilitate the construction of method names, a few automatic approaches have been proposed recently to recommend method names according to given method bodies/implementation [6]–[9]. All such approaches leverage advanced machine learning techniques, and thus in this paper we call them ML-based approaches. The rationale of such approaches is that method names are associated with certain features of methods, and the association could be learned automatically by advanced machine learning techniques. Method features frequently employed by such approaches include source code metrics (e.g., cyclomatic complexity) [6], the sequence of identifiers within the method body [7], and paths connecting nodes in the abstract syntax tree (AST) of method body [8], [9]. Machine learning techniques that are frequently employed by such approaches include log-bilinear neural network [6], convolutional neural network [7], conditional random fields [8], and attentional neural network [9]. Existing approaches exhibit high accuracy, and the state-of-the-art approach *code2vec* achieves high precision of 63.1% and high recall of 54.4% [9].

Although such approaches are overall accurate in recommending method names, little is known about where and why they work or don't work. We also know little about the usefulness of such approaches. To this end, in this paper, we perform a comprehensive assessment and in-depth analysis on the state-of-the-art approach (i.e., *code2vec* [9]) with more realistic settings (i.e., cross-project validation and excluding overriding methods) on a new dataset. Evaluation results suggest that although switching datasets does not significantly influence the performance of *code2vec*, more realistic settings do lead to significant reduction in performance. Further analysis on the cases where *code2vec* succeeds suggests that:

- First, around half (48.3%) of the accepted method names are made for getter/setter methods.
- Second, a large portion (19.2%) of successfully recommended method names could be copied from identifiers within given bodies. That is, the recommended method name for the given body has been typed in the body before recommendation is requested.

These findings suggest that *code2vec* succeeds rarely when it is strongly needed. To further validate its usefulness, we ask developers to manually score the difficulty in constructing names for 600 Java methods, and then request *code2vec* to make recommendations for them. Results confirm the conclusion that *code2vec* rarely works when it is really needed.

To intuitively illustrate the state of the art, and to investigate the possibility of designing simple but effective approaches, we propose a Heuristic based approach to recommending Method names (called *HeMa*) for given method bodies. It is essentially a sequence of simple heuristics, which makes it easy to follow. Evaluation results suggest that it significantly outperforms the state-of-the-art ML-based *code2vec*.

The paper makes the following contributions:

- First, a comprehensive assessment and in-depth analysis of the state-of-the-art approach in ML-based method name recommendation. The analysis not only reveals the state of the art, but also discovers where and why the approach works or doesn't work.
- Second, a simple and straightforward approach in recommending method names. It significantly outperforms the state-of-the-art ML-based complicated approach in the evaluation, which suggests that ML-based approaches may still have a long way to go.

The rest of this paper is structured as follows. Section II presents related work. Section III introduces empirical setting, and Section IV presents results and analysis. Section V proposes a heuristics based approach. Section VI discusses related issues, and Section VII presents conclusions and future work.

II. Related Work

A. Recommendation for Method Name

The quality of identifiers proved to have a significant impact on the readability and maintainability of software source code [10], [11] Method names, as a special kind of identifiers, are especially important because they serve as cornerstone of abstraction for aggregated behaviors [5]. Consequently, a series of approaches have been proposed to improve the quality of method names.

Høst and Østvold [5] develop a technique for automatically inferring naming rules of methods based on the return type, control flow, and parameters. A rule violation indicates a conflict between the name and the associated implementation of a method. To resolve the conflict, they filter relevant phrases for rule violations by sorting the candidate list according to the rank of the semantic profile in candidate phrase corpus as well as the semantic distance from the inappropriate phrase to the candidate phrase.

Allamanis et al. [6] tackle the problem of method naming by introducing a log-bilinear neural language model, which includes feature functions that capture long-distance context in source code, and a subtoken model that can predict neologisms, i.e. names that do not appear in the training set. The model embeds each token into a high dimensional continuous space and suggest the name that is most similar in this space to those in the method body. Allamanis et al. [7] later take the method naming as a problem of extreme summarization of source code where the method name is viewed as the summary of a method body. To generate the summary, they introduce an attentional neural network that employs convolution on the input tokens in the body. The network can learn high-level patterns in source code that uses both the structure of method body and the identifiers to detect and explain complex constructs.

Alon et al. [8] propose a general path-based approach for representation of methods. The main idea is to represent a method using the bag of paths between leaves in its AST and identifiers associated with corresponding leaves. This allows machine learning models to leverage the structured nature of source code rather than treating it as a flat sequence of tokens.

code2vec published by Alon et al. [9] further represents a method body into a distributed vector by aggregating the bag of AST paths with attentional network [12]. The attention mechanism computes a weighted attention value for each of the AST paths in a method and aggregate them into a single vector by weighted summation to represent the method body. The vectors of methods that share similar AST structures are close to each other in the continuous distribution space and thus capture semantic similarity between methods as well as between method names. Through training with abundant AST structures in large-scale open-source projects, *code2vec* can retrieve highly similar (i.e. close in the vector space) method bodies with the given one and recommends to reuse names of such methods. Alon et al. apply *code2vec* to method name recommendation, and evaluation results suggest *code2vec* achieves the state of the art on this task.

Another deep learning based method name recommendation approach is proposed by Liu et al. [13]. They embed method names and method bodies into numerical vectors with paragraph vector and word2vec/CNNs, respectively. For a given method m, they retrieve the set of method names (noted NS) that are close to m in the name vector space, and the set of methods names (noted BS) whose bodies are close to m in the body vector space. If $BS \bigcap NS = \emptyset$, method m should be renamed, and the proposed approach suggests alternative consistent names. We do not evaluate this approach because it is not published yet at the time of completing this paper. In future, we would like to empirically investigate it and compare it again the proposed approach.

B. Recommendation for Other Identifiers

There are also a lot of automated approaches proposed to improve the quality of identifiers, e.g., class, field, variable and parameter names. Caprile and Tonella [14] first propose an approach to standardize program identifiers. They first generate a dictionary that transforms composing words in identifiers into their associated standard terms. Second, they infer a standard syntax from examples representative of the different forms that can be associated to the main grammatical functions for arranging those standardized terms into a sequence.

Deissenboeck and Pizka [1] suggest that well-formed identifier names should be both concise and consistent (i.e., a single identifier represents a single concept throughout the program). They propose a formal model which is based on bijective mappings between concepts of methods and names to provide a solid foundation for the definition of concise and consistent naming. In a follow up experiment, Lawrie et al. [15] surveyed the identifiers found in 48 MSLOC of C, C++, Fortran and Java source code to determine the extent of violations of concise and consistent naming. The syntactic methodology employed by Lawrie et al. identifies potential violations of concise and consistent naming which include some conventional patterns of naming found in Java inheritance trees.

Thies and Roth [16] identify and rename inconsistent identifiers through static code analysis. The rationale of their approach is that variables on different sides of the same assignment had better be named consistently. They consider two types of assignments: assignments between variables, and assignments between variables (on the left side) and method invocations (on the right side). They represent variables and method invocations involved in assignments as nodes, and connect nodes (with edges) that are involved in the same assignments. Based on the resulting graph, they identify inconsistent identifiers with heuristics.

Allamanis et al. [17] propose a framework called NATURALIZE that learns the coding conventions used in a code base and suggest natural identifier names to improve stylistic consistency. NATURALIZE works by identifying names or formatting choices that are surprising according to a probability distribution learned with n-gram model over source code tokens. When surprised, NATURALIZE determines whether it is sufficiently confident to suggest a renaming. If yes, it unifies the surprising name with one that is preferred in similar contexts elsewhere in its training set. LEAR, proposed by Lin et al. [18], is a variation of NATURALIZE. It differs from NATURALIZE in the following aspects. First, LEAR focuses on such tokens only that contain lexical information. Second, LEAR checks the validity of possible renaming identifiers whereas NATURALIZE does not.

Raychev et al. [19] build a scalable prediction engine called JSNICE for predicting properties (including both type and name) of identifiers in the context of JavaScript. They first convert source code into a representation called dependency network that captures relationships between program elements, whose properties are to be predicted, with elements, whose properties are known. Once the network is obtained, they perform structured prediction based on a learned conditional random field model.

Pradel and Sen [20] formulate the problem of name-based bug detection as a binary classification problem. They develop a framework to generate training data and to train a binary classifier. They create training data by simple program transformation that yields likely buggy programs. The binary classifier is then trained on such automatically generated training data (negative samples) as well as buggy free programs (positive samples).

C. Machine Learning based Code Recommendation

Recommender systems have a wide range of applications in software engineering to improve productivity and reliability [21], [22]. Many of these systems employ machine learning techniques to assist developers in writing or maintaining software source code. The most popular recommender system used in integrated development environments (IDEs) is code recommendation system [23]. Hindle et al. [24] are the first to employ n-gram model in recommending the next code token for developers by statistically learning the repetitiveness of source code in token level. Following their work, a series of n-gram based approaches are proposed to recommend the next token by exploiting large-scale dataset [25], augmenting with semantic information [26], employing formal properties of code [27] and adding cache mechanism [28], [29]. Apart from token-level models, graphic probability models are also successfully employed to recommend the next API method call [30], [31] Such models statistically learn the probability distribution of API usage graphs [32] extracted from source code snippets, and then recommend the next API by computing the appearance probability of each API against a given usage graph. Another line of machine learning based code recommendation approaches leverage extensively used deep neural networks. Raychev et al. [27] first employ RNN model [33] for code recommendation and White et al. [34] also demonstrate its high effectiveness in recommending sequential source code. Li et al. [35] augment RNN model with attention mechanism [12] and pointer copy component [36] to recommend the next AST node in both type and name of identifiers.

III. Experimental Setup

This section specifies the setup of the experiment, i.e., approaches selected for the evaluation, research questions expected to answer, and metrics employed to quantitatively assess the performance of the evaluated approaches.

A. Evaluated Approach

To evaluate the state of the art in ML-based recommendation of method names, we select *code2vec* [9] for the evaluation. *code2vec* is selected because of the following reasons. First, it represents the state of the art in this field. As introduced in Section II, *code2vec* was proposed recently on POPL 2019, and proved significantly more accurate than alternative approaches [9]. Second, the source code of its implementation is publicly available, which significantly facilitates the evaluation. It also facilitates other researchers to replicate the experiment. We make the replication package of the evaluation publicly available on GitHub [37] to facilitate third-party replication and further investigation.

B. Research Questions

The experiment investigates the following research questions:

- **RQ1**: How well does *code2vec* work on datasets other than the one employed by the original evaluation conducted by the authors of *code2vec*?
- **RQ2**: How well does *code2vec* work with more realistic settings?
- **RQ3**: Can *code2vec* generate method names correctly when the given method bodies do not contain method name tokens?
- **RQ4**: Where and why does *code2vec* work?
- **RQ5**: Where and why does *code2vec* fail?
- **RQ6**: Is *code2vec* useful for developers? How often does *code2vec* recommend correctly for methods that are challenging to name manually?

Research question RQ1 validates whether *code2vec* keeps highly accurate when evaluation data is replaced. *code2vec* was originally evaluated on a big dataset (called *original dataset* for short) [9]. Besides that, we employ another publicly available dataset [38] (called *new dataset* for short) that consists of 1,000 top-starred Java projects from GitHub. Answering this question may reveal the generality of *code2vec*.

Research question RQ2 validates *code2vec*'s sensibility to the empirical settings. The original evaluation conducted in [9] is file based, i.e., Java files from subject applications are shuffled, and randomly divided into three groups (training, validation and testing sets). As a result, while recommending method names for a project, *code2vec* may exploit a large number of files from the same project. We call such validation setting *file-based validation*. Although *file-based validation* is reasonable, it is quite often that developers create new projects from scratch, and thus they don't have enough data from the current project to train the method name recommendation models. *Project-based validation* fits this scenario. It also releases developers from on-site training, which requests deep understanding of the underneath learning models. In this setting (*project-based validation*), we pre-train *code2vec* off-line with a corpus of open-source subject applications and no source code from the testing project is exploited for training. The key advance of *project-based validation* is that the time and resource consuming model training is conducted once and for all before such models are actually exploited by developers for method name recommendation. The third validation setting is *project-based non-overriding validation*. The only difference between *project-based non-overriding validation* and *project-based validation* is that overriding methods are excluded by the former pattern but included by the latter. Although the original evaluation in [9] excludes constructors because it is unlikely that developers do not know how to name constructors, they do not exclude overriding methods that are quite similar to constructors. Overriding methods share the same names with methods they override, and thus it is unlikely that developers need help in naming such methods.

Research question RQ3 investigates the performance of *code2vec* when the given method bodies do not contain the method name tokens. Answering research question RQ3 may reveal to what extent *code2vec* can *coin* (instead of *copy*) method names correctly.

Research questions RQ4 and RQ5 investigate the strengths and weaknesses of *code2vec*. It is likely that *code2vec* works well on one category of methods, but works badly on another category. Answering these two research questions may reveal where *code2vec* succeeds/fails and the rationale for the success/failure.

Research question RQ6 concerns the usefulness of *code2vec*. The usefulness depends on both accuracy of the recommendation and the difficulty of method name construction. If *code2vec* frequently fails when requested to recommend difficult method names (i.e. when really needed), it would be useless for developers. To this end, we manually score the difficulty in method name construction, and evaluate the usefulness of the approach based on the scores. Answering research question RQ6 helps to reveal how often *code2vec* works when it is really needed.

C. Metrics

Commonly employed metrics for method name recommendation include $Precision@k$, $Recall@k$, and $F1@k$ [6]–[8]. $Precision@k$ presents the precision of top k recommendation, and it is computed as follows:

$$Precision@k = \frac{N_{accepted}@k}{N_{recommended}}, \tag{1}$$

where $N_{accepted}@k$ is the number of cases where the evaluated approach succeeds, and $N_{recommended}$ is the number of cases the evaluated approach tries. Notably, the approach succeeds if and only if one of items within the top k recommendation list is accepted. In our evaluation, a recommended method name is accepted if and only if it is identical to the one manually constructed by developers. However, it is possible that the recommended name is acceptable although it is different from the original one, especially when the original one is an improper name. Consequently, the performance we report is exactly the lower-bound, i.e., the actual performance could be slightly higher from what we report in the paper.

Similarly, $Recall@k$ and $F1@k$ are computed as follows:

$$Recall@k = \frac{N_{accepted}@k}{N_{tested}} \tag{2}$$

$$F1@k = 2 \times \frac{Precision@k \times Recall@k}{Precision@k + Recall@k} \tag{3}$$

where N_{tested} is the number of tested methods, i.e. the size of testing set. For approaches, e.g., *code2vec*, that always make recommendation regardless of the input, N_{tested} is equal to $N_{recommended}$. Consequently, their precision and recall are always equal, i.e., $Precision@k = Recall@k$.

TABLE I
EVALUATION RESULTS ON DIFFERENT DATASETS

Datasets	Number of methods	Precision / Recall			MRR
		Rank 1	Rank 5	Rank 10	
Original Dataset	13,376,807	49.89%	56.99%	58.41%	52.96%
New Dataset	3,826,986	43.87%	49.90%	51.33%	46.51%

In addition, we employ the Mean Reciprocal Rank to assess the performance of method name recommendation approaches. It is computed as follows:

$$MRR = \frac{1}{N_{tested}} \sum_{i=1}^{N_{tested}} \frac{1}{rank_i} \quad (4)$$

where $rank_i$ is the rank of correct name within the recommendation list for the i-th testing method.

IV. RESULTS AND ANALYSIS

A. RQ1: Comparable Performance on Different Datasets

To address research question RQ1, we evaluate *code2vec* on a new dataset as well as its original dataset employed in paper [9]. To be aligned to the evaluation in paper [9], we randomly select 50,000 Java files from each of the datasets as testing set, and another 50,000 as validation set. Others are employed as training set.

Evaluation results are presented in Table I. The first column presents datasets employed for the evaluation. The second column presents the size of involved datasets. The third to fifth columns show the performance (precision/recall) of *code2vec* to quantitatively present how often correct method names appear in the top-k recommendation list. The last column presents the Mean Reciprocal Rank of *code2vec*. Notably, for *code2vec*, its recall is always equal to the precision. The reason has been presented in Section III-C.

From the table, we make the following observations:

- First, *code2vec* is accurate in recommending method names. The top 1 recommendation is often (at a chance of 49.89% on original dataset and 43.87% on new dataset) correct. It has a great chance (58.41% on original dataset and 51.33% on new dataset) to present the correct method names on its top 10 recommendation list. High MRR (52.96% and 46.51%) also suggests that the correct names are often ranked on the top.
- Second, switching datasets does not result in significant reduction in performance. Although the precision/recall is slightly reduced (e.g., from 49.89% to 43.87% on rank 1), the major reason for the reduction is the size of new dataset: the number of methods in *new dataset* is only 28.6% of that in *original dataset*.

We conclude from the preceding analysis that *code2vec* overall is accurate, and switching to a new large-scale dataset dose not result in significant reduction in performance of *code2vec*.

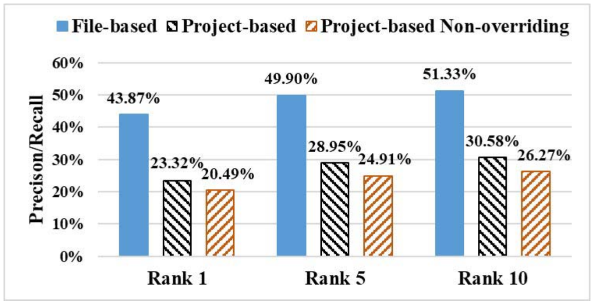

Fig. 1. Evaluation Results with Different Settings

B. RQ2: Realistic Settings Result in Reduced Performance

To address research question RQ2, we evaluate *code2vec* on our *new dataset* with different settings, i.e., *file-based validation*, *project-based validation*, and *project-based non-overriding validation*. *File-based validation* partitions dataset into training, validation and test sets at file level whereas the other two validations work at project level. The only difference between *project-based validation* and *project-based non-overriding validation* is that overriding methods are excluded by the latter but included by the former.

Evaluation results with different settings are presented in Fig. 1. From the figure, we make the following observations:

- First, switching evaluation setting from *file-based validation* to *project-based validation* decreases the performance significantly. The precision/recall is significantly reduced by 46.84%=(43.87%-23.32%)/43.87% at rank 1, 41.98%=(49.90%-28.95%)/49.90% at rank 5, and 40.42%=(51.33%-30.58%)/51.33% at rank 10.
- Second, excluding overriding methods further decreases the performance of *code2vec*. Notably, overriding methods are popular and account for 30%=1,147,980/3,826,986 of methods in the dataset. Excluding such methods reduces precision/recall by 12.14%=(23.32%-20.49%)/23.32% at rank 1, 13.96%=(28.95%-24.91%)/28.95% at rank 5, and 14.09%=(30.58%-26.27%)/30.58% at rank 10.
- Third, overall, *code2vec* works well with different settings. Its precision/recall keeps greater than 20% regardless of the change in settings, suggesting that on more than one fifth of cases *code2vec* succeeds in inferring the exact method name. The high MRR (46.51%, 25.76%, and 22.43% in *file-based validation*, *project-based validation*, and *project-based non-overriding validation*, respectively) also suggests that *code2vec* is quite promising.

TABLE II
EVALUATION RESULTS ON UNSEEN/SEEN TOKENS

Token Types	Total (N_t)	Correct (N_c)	N_c/N_t
Unseen Tokens	385,225	85,034	22.07%
Seen Tokens	581,237	152,826	26.29%

We have not yet fully understand why switching from *file-based validation* to *project-based validation* results in so much reduction (more than 40%) in performance. A possible reason for the significant reduction is that *file-based validation* leverages the nature of source code: local repetitiveness [28], [29]. It proved that source code has the characteristic of local repetitiveness within projects, and machine learning models may learn special patterns that are specific to a given project if they are trained with part of the project and then applied to the other part of the project [28], [29]. *File-based validation* exploits the major part of a subject application while another small part of the same application is under test. As a result, *code2vec* can take full advantage of local repetitiveness. However, *project-based validation* partitions data at the granularity of projects, i.e., a single project as a whole is taken as either testing data or training data (but never the both at the same time). Consequently, it cannot take advantage of local repetitiveness as *file-based validation* does.

We conclude from the preceding analysis that the performance of *code2vec* decreases significantly on more realistic settings. However, its overall performance is still promising.

C. RQ3: Coining Method Names

To address research question RQ3, we evaluate the performance of *code2vec* in generating seen and unseen method name tokens, respectively. For each of method names in *new dataset*, we split it into tokens according to the Camel-Case naming convention. *A token t from method name mn is a seen token if t appears (case insensitive) in the body named by mn. Otherwise, it is an unseen token.* We assess how often *code2vec* can successfully recommend seen/unseen method name tokens during *project-based non-overriding validation* conducted in Section IV-B.

Evaluation results are presented in Table II. The first column presents the types of method name tokens: unseen tokens and seen tokens. The second column presents total number of seen/unseen method name tokens in the testing set. The third column presents the number of correctly recommended seen/unseen method name tokens. A token t from method name mn is correctly recommended if and only if the name recommended for the method body of mn contains token t. The forth column presents the chance that seen/unseen method name tokens are recommended correctly. The chance is computed by dividing the number of correctly recommended seen/unseen method name tokens by the total number.

From the table, we make the following observations:

- First, a large percentage of method name tokens (39.86%=385,225/(385,225+581,237)) are unseen, i.e., they do not appear in the input (method bodies). Consequently, ML-based approaches that strongly rely on *copy* mechanism to select tokens from input have significant limitation on their maximal potential.
- Second, *code2vec* works quite well in recommending unseen method name tokens. It successfully generates 22.07% of unseen tokens. Its performance in generating unseen method name tokens is even comparable to that (26.29%) on seen tokens.

code2vec can generate unseen method name tokens because it does not select (copy) tokens from its input, i.e., method body. In contrast, it extracts features of methods (i.e., paths connecting nodes in the AST of method bodies), and associates such features with method names. As a result, if the given method body (mb_1) shares the same or highly similar features with another method body (mb_2) in the training set, *code2vec* may recommend the method name ($Name_{mb_2}$) of mb_2 for mb_1 even if the tokens of $Name_{mb_2}$ do not appear in mb_1.

A typical example is presented in Listing 1. *code2vec* successfully coins the method name for the testing method on Line 1 though the token *contains* doesn't appear in the body. Through retrieving training methods named as *contains* (e.g. the method on Line 9), we find that their syntax structures exploited by *code2vec* are always similar. Thus *code2vec* can successfully recognize the specified method body associated with *contains* and suggest to reuse the training method name.

```
1  public static boolean contains(String str,
2    String[] array) {
3    for(String s : array)
4      if(str.equals(s))
5        return true;
6    return false;
7  }
8
9  public static boolean contains(int[] values,
10   int candidate) {
11   for(int i = 0; i < values.length; i++)
12     if(values[i] == candidate)
13       return true;
14   return false;
15 }
```

Listing 1. An Example of Coined Method Name

From the analysis in the preceding paragraphs, we conclude that *code2vec* works well in generating unseen method name tokens.

D. RQ4: Where and Why code2vec Works

To investigate where and why *code2vec* works, we manually analyze cases where it works during *project-based non-overriding validation*. Notably, *code2vec* succeeds on a large number (61,906) of methods, and thus it is challenging, if not impossible, to manually analyze all of these methods. To this end, we randomly sample one thousand of these methods for manual analysis. Based on the manual analysis, we make the following observations:

- First, a large portion (48.3%=483/1,000) of accepted names are recommended for getter/setter methods.

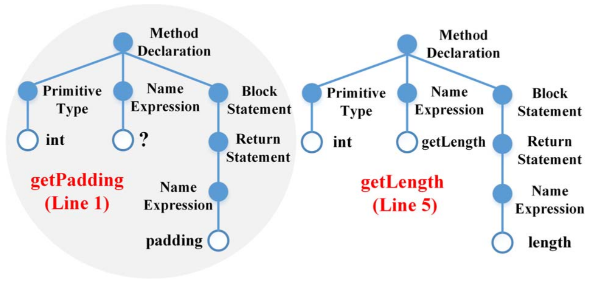

Fig. 2. Common AST Pattern of Getter Methods

- Second, it is quite often (at a chance of 19.2%) that the correct method name could be copied from the corresponding method body. In the rest of this paper, we call such methods name-contained methods.

To investigate why *code2vec* works well on getter/setter methods, we manually analyze features of such methods. Two typical getter methods are presented in Listing 2. Their ASTs are presented in Fig. 2. From this figure, we observe that the two trees are highly similar, and thus their features (i.e., paths that connect each pair of terminal nodes in the AST) are also highly similar. According to our manual analysis, most of getter methods have highly similar ASTs. Consequently, *code2vec* can distinguish getter/setter methods because of the features, and generate method names according to the terminal node on the bottom right corner of the AST.

```
1  public int getPadding() {
2    return padding;
3  }
4
5  public int getLength() {
6    return length;
7  }
```

Listing 2. Getter Methods

```
1  public void reset() {
2    //delegation
3    reset(operation);
4  }
5
6  public void reset(String operation) {
7    //delegation and server
8    reset(operation, true);
9  }
10
11 public void reset(String newOperation,
12   boolean logOperationOnChange) {
13   //server
14   if (time != null)
15     if(operation.equals(newOperation) ||
16       logOperationOnChange)
17       logOperation(operation, elapsedTime());
18   time = new Date().getTime();
19   if(!operation.equals(newOperation)) {
20     operation = newOperation;
21     log.info(String.format(
22     "Operation␣'%s'␣started.", newOperation));
23   }
24 }
```

Listing 3. Delegations

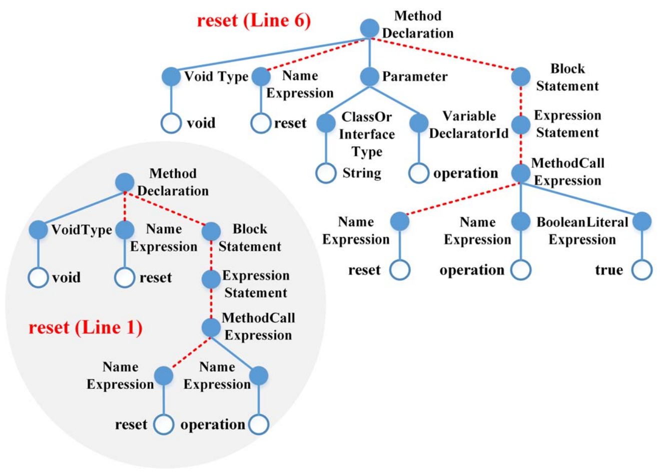

Fig. 3. Common Path of Delegations

Name-contained methods are created for various reasons, e.g. delegations. A *delegation* is such a method that does nothing except passing messages between its invoker and another method (called server). A typical example is presented in Listing 3. Three overloading methods share the same name *reset*, which is required by overloading mechanism. Overloading method with fewer parameters (e.g., the one on Line 1) often serves as a delegation to more complex one (e.g., the one on Line 6) with default arguments. As a result, the name '*reset*' of invoked method appears within the method body of the delegation (on Line 3), and it *happens* that this name is identical to that of the delegation. The same is true for another delegation (on Line 6) whose server is the method defined on Line 11. According to our manual analysis, delegations account for a large percentage (45.83%) of the name-contained methods.

To investigate why *code2vec* works well on delegations, we analyze the AST of delegations. ASTs of the delegations in Listing 3 are presented in Fig. 3. From this figure, we observe that such delegations share a common path (shown in dashed line). The path travels from the method name (terminal node *reset*) to the root of the tree (*MethodDeclaration*), passes the root of method body (*BlockStatement*) and the only statement (*ExpressionStatement* and *MethodCallExpression*), and finally reaches the name of invoked method (another terminal node *reset*). *code2vec* can learn to decide whether a given method body is a delegation by looking for such a path (ignoring the two terminals). If a delegation, *code2vec* suggests to reuse the name of invoked method as the name of the delegation.

We conclude from the preceding analysis that *code2vec* works well on getter/setter methods and delegations. The rationale for the success is that such methods have common structures in their ASTs.

E. RQ5: Where and Why code2vec Fails

To investigate where and why *code2vec* fails, we conduct another manual analysis that is highly similar to that in

Section IV-D. The only difference is that here we analyze one thousand failed cases during *project-based non-overriding validation* whereas Section IV-D analyzes successful cases.

Based on the manual analysis, we figure out the following major reasons for the failure. First, *out-of-vocabulary* method names are the primary for the failure. *A method name is out-of-vocabulary if and only if none of the methods in training set is named with it.* Notably, *code2vec* builds a vocabulary of method names by collecting all unique method names in the training set, and *selects* a name from this vocabulary as the recommendation for a given method body. Consequently, if the optimal method name is not included in the vocabulary, *code2vec* has no chance to generate it. This kind of methods are common, accounting for 52%=157,128/302,200 of methods in the testing set. As a result, *code2vec* fails frequently for this reason. Employing open vocabulary or coining method name by composing tokens may help to further improve the performance.

The second reason for the failure is that the exploited method features are often insufficient to infer method names. Listing 4 presents a typical example. Method bodies of *deepCopy* and *build* are almost identical except for the data type (*TIncrement* vs. *ReportAttributes*). Consequently, their AST paths exploited by *code2vec* as method features are highly similar as well. As a result, *code2vec* recommends the same name for them. Another example is presented in Listing 5, where two methods (*err* and *rawError*) from different applications have the same method body. However, they are named differently by their authors. All of these examples suggest that the exploited feature, sometimes even the whole input (method body), is often insufficient to infer method names. To further improve the performance, additional information, e.g., their enclosing classes, should be exploited as well.

```
public TIncrement deepCopy() {
  return new TIncrement(this);
}

public ReportAttributes build() {
  return new ReportAttributes(this);
}
```

Listing 4. Similar Methods Named Differently

```
public static void err(String msg) {
  System.err.println(msg);
}

public static void rawError(String msg) {
  System.err.println(msg);
}
```

Listing 5. Identical Methods Named Differently

The third reason for the failure is improper method names in the testing set. During the evaluation, all recommended names are compared against original names associated with testing methods, i.e., the *code2vec* fails if the recommended name is different from the original one. However, it is likely that the recommended name could be even better than the original one. In such cases, *code2vec* fails, and we call such cases false negatives. For example, *code2vec* generates name '*initRecyclerView*' for the method on Line 1 in Listing 6. The recommended name is better than the original one '*initializeRecyclerview*' because the abbreviation 'init' significantly shortens the identifier whereas keeps the readability. Notably, 'init' has been widely employed to represent 'initialize' (e.g. in a large number of JDK methods) and thus it is not difficult for experienced developers to guess the meaning. The recommended new name also successfully corrects the spelling of 'Recyclerview' to make it consistent with Camel Case naming convention. Another example is presented on Line 11, for which *code2vec* generates name '*setPassword*'. It is better than the original name '*password*' because the latter fails to reveal the intent of method, and violates naming conventions: method names should be a verb or a verb-noun phrase [39], [40].

```
1  private void initializeRecyclerview() {
2    RecyclerView.LayoutManager layoutManager =
3    new GridLayoutManager(getActivity(), 2);
4    recyclerView.setLayoutManager(layoutManager);
5    recyclerView.setHasFixedSize(true);
6    adapter = new RecyclerViewAdapter(
7    getActivity().getApplicationContext());
8    recyclerView.setAdapter(adapter);
9  }
10
11 public KeycloakBuilder password(
12   String password) {
13   this.password = password;
14   return this;
15 }
```

Listing 6. Improper Method Names in Testing Set

F. RQ6: Limited Usefulness for Developers

To answer research question RQ6, we investigate how often *code2vec* works when it is strongly needed. The investigation is conducted as follows:

- First, we invite six developers involved in a commercial project for the evaluation. The commercial project has been released recently by a giant of IT industry to conduct large-scale software refactorings.
- Second, for each of the participants, we randomly select one hundred methods developed by himself/herself.
- Third, we request all participants to score the difficulty in naming sampled methods. Notably, participants do not score methods developed by other developers, and thus each of them score exactly one hundred methods. The scores rank between one and five (i.e. 5-point scale [41]–[44]) where one represents least difficulty and five represents highest difficulty.
- Fourth, we apply *code2vec* of *project-based non-overrding validation* to the scored methods, and validate the recommendations against manually constructed names.

Evaluation results are presented in Table III. The first column presents the difficulty scores in naming methods. The second column presents the number of methods with the specific difficulty. The third column presents the precision/recall of *code2vec* at rank 1 on the given methods.

TABLE III
EVALUATION RESULTS ON MANUALLY SCORED METHODS

Scores	Number of methods	Precision / Recall
1 (Very Easy)	287	34.84%
2 (Easy)	159	6.92%
3 (Normal)	91	5.49%
4 (Difficult)	50	2.00%
5 (Extremely Difficult)	13	0.00%
Total	**600**	**19.50%**

From the table, we make the following observations:

- First, a significant percent of methods are difficult to name even for authors of such methods. We note that out of 600 methods, 13 are extremely difficult to name, and 50 are difficult to name. In total, 10.5% = (13+50)/600 of the methods are difficult to name for experienced developers.
- Second, the overall precision/recall (19.50%) of *code2vec* is comparable to that on large-scale dataset employed in Section IV-B where its precision at rank 1 is 20.49%. It suggests that conclusions drawn on open-source applications may hold on commercial closed-source applications as well.
- Third, *code2vec* works well on very-easy-to-name methods, resulting in high precision of 34.84%.
- Finally, *code2vec* rarely succeeds on difficult-to-name methods where recommendation on method names is strongly needed. It fails on all of 13 extremely-difficult-to-name methods, and 49 out of 50 difficult-to-name methods. Overall, its precision on such methods is significantly reduced to 1.6%. We also notice that its precision decreases significantly with the increase of difficulty scores.

From the analysis in the preceding paragraph, we conclude that *code2vec* rarely works when it is strongly needed.

G. Threats to Validity

A threat to the external validity is that we employ only one new dataset to validate the impact of switching datasets on the performance of *code2vec*. It is likely that our conclusion may not hold if other datasets are involved. To reduce the threat, we perform our assessment and analysis on a large-scale dataset composed of 1,000 projects. It is challenging to collect additional comparable dataset for the evaluation. Another threat to external validity is that only 600 manually scored methods are involved in our experiment to assess the usefulness of *code2vec*. It is challenging to recruit more expert developers from the industry for the evaluation. Another threat to validity concerning the manual scoring is that the scoring is rather subjective. Consequently, conclusions drawn on such participants may not hold for other developers.

A threat to construct validity is that we assess the correctness of generated names based on their equivalence to the manually constructed ones. However, as introduced in Section IV-E, it is likely that the recommended method names could be better than manually constructed ones, which makes the assessment inaccurate. To reduce the threat, we employ only top-starred projects where most of the method names are likely well-constructed.

V. HEURISTICS BASED ALTERNATIVE APPROACH

code2vec is a complicated ML-based approach that is difficult to interpret. It also requires training on a large corpus of high quality source code that is both time consuming and resource consuming. To investigate the possibility of designing a simple and straightforward approach for method name recommendation, in this section, we propose a heuristics based approach *HeMa* that outperforms *code2vec*. The implementation (source code) of *HeMa* is publicly available on GitHub [37].

A. Overview

HeMa is essentially a sequence of heuristics. For a given method body mb, *HeMa* works as follows:

1) First, based on a sequence of heuristics, *HeMa* decides whether mb is a getter/setter method. If yes, *HeMa* generates method name for it automatically based on another sequence of heuristics.
2) Second, based on a sequence of heuristics, *HeMa* decides whether mb is a delegation. If yes, *HeMa* generates method name for it automatically based on another sequence of heuristics.
3) Third, if the preceding heuristics fail, *HeMa* employs a sequence of heuristics to retrieve methods in a large corpus that share the same return type and parameters (both parameter names and parameter types but regardless of parameter orders) with mb. From the resulting set of methods (notated as S_m), *HeMa* picks up the most popular method name and suggests it to mb. If $S_m = \emptyset$, *HeMa* refuses to make any recommendation.

Details of the key steps are presented in the following sections.

B. Distinguishing Getter/Setter Methods

Getter and setter methods often follow common patterns. One of the most well-known patterns for getter methods is "return ${field};" whereas "${field} = ${param};" is for setter methods. Following these patterns, we define a sequence of heuristics to distinguish getter and setter methods from others.

For a given method body mb, *HeMa* distinguishes whether mb is a getter method as follows:

- First, if the given method body returns nothing, i.e. the return type is *void*, the given method is not a getter;
- Second, if the given method body contains more than one *ReturnStatements*, *HeMa* will not recognize it as a getter method;
- Third, if the value returned by the only *ReturnStatement* is a field (notated as ${field}) declared within the enclosing class, it is a getter method.

For the potential getter method, *HeMa* composes a method name as "get${field}".

HeMa distinguishes method body mb as a setter method if:

- The given method body has at least one parameter;

TABLE IV
COMPARISON BETWEEN HEMA AND CODE2VEC

Scores	HeMa			code2vec
	Precision	Recall	F1	Precision
1	54.59%	41.46%	47.13%	34.84%
2	25.00%	10.69%	14.98%	6.92%
3	7.69%	3.30%	4.62%	5.49%
4	9.09%	4.00%	5.56%	2.00%
5	0.00%	0.00%	0.00%	0.00%
Total	**39.94%**	**23.50%**	**29.59%**	**19.50%**

- The body is composed of a single assignment;
- The left hand side of the assignment is a field (notated as ${field}) declared within the enclosing class;
- And, the right hand side of the assignment is a parameter of the method (notated as ${param}).

For the potential setter method, *HeMa* composes the method name as "set${field}".

C. Distinguishing Delegations

For a given method body mb, *HeMa* distinguishes it as a delegation if:

- The method body contains a single statement;
- And the statement is a *ReturnStatement* that returns an invocation on another server method (notated as $sMethod$).

For the potential delegation, *HeMa* recommends to reuse the name of invoked method (i.e. $sMethod$).

D. Frequency-based Name Recommendation

If the given body mb is neither geter/setter nor delegation, *HeMa* retrieves a set of methods (S_m) that share the same return type and the same parameter list (including both parameter types and names but regardless of the order of parameters). If S_m is empty, *HeMa* refuses to make any recommendation. Otherwise, it selects the method name with highest frequency in S_m, and recommends to use this name for the given method body.

E. Comparison Against code2vec

To compare *HeMa* against *code2vec*, we evaluate *HeMa* on the *new dataset* employed in Section IV-B with *project-based non-overriding validation*. Evaluation results suggest that *HeMa* significantly outperforms *code2vec*. Its precision, recall, and F1 at rank 1 are 33.86%, 25.09%, and 28.82%, respectively. In contrast, the precision of *code2vec* at rank 1 is 20.49% (notably, its recall and F1 are equal to the precision as explained in Section III-C). Compared to state-of-the-art *code2vec*, *Hema* successfully improves precision, recall, and F1 by 65.25%=(33.86%-20.49%)/20.49%, 22.45%=(25.09%-20.49%)/20.49%, and 40.65%=(28.82%-20.49%)/20.49%, respectively.

We also evaluate *HeMa* with manually scored dataset used in Section IV-F to compare the usefulness of *HeMa* against *code2vec*. Evaluation results (at rank 1) are presented in Table IV. From the table, we observe that *HeMa* outperforms *code2vec* on difficult-to-name methods, improving the precision, recall and F1 from 2.00% to 9.09%, 4.00% and 5.56%, respectively. One of the reasons for the improvement is that the *frequency-based name recommendation* employed by *HeMa* works for many complicated method bodies. This heuristic alone successfully generates method names for 8.99%=27,155/302,200 of the testing methods in *new dataset*.

From the analysis in the preceding paragraphs, we conclude that the heuristics based *HeMa* significantly outperforms the state-of-the-art ML-based *code2vec*.

VI. DISCUSSION

A. Implications

The empirical study in Section IV and the alternative approach proposed in Section V have the following implications:

1) Empirical Settings Are Critical: We switch from casual empirical setting to more realistic ones in Section IV-B, and results suggest that the switching leads to significant reduction in performance. It implicates that empirical settings should be carefully designed, and should be as close as possible to real scenarios in the industry. It is quite often that researchers, especially those from fields (e.g., AI and NLP) other than software engineering, pay little attention to the application scenarios of proposed automatic software engineering tools/approaches. As a result, the empirical settings are significantly different from real scenarios, and thus the evaluation results fail to reveal the reality of such approaches. Notably, recent empirical study conducted by Hellendoorn et al. [45] results in similar implications. They analyze empirical settings employed in the evaluation of code completion tools, and suggest that such settings are essentially different from real ones in practice. They empirically switch to more realistic settings, and results suggest that the switching leads to significant reduction in performance. Our findings are highly consistent with theirs.

2) A Friend in Need Is A Friend Indeed: Evaluation results in Section IV-F suggest that although the overall performance of *code2vec* is promising, it is not much useful. The major reason is that it frequently works when it is not strongly needed but fails when it is in need. The evaluation may suggest that overall performance alone could be insufficient to measure the usefulness of automatic software engineering tools. We should allocate priority to cases where the tools are urgently needed, and assess how often the tools work in such cases.

3) Complex Approaches Are NOT Necessarily Better than Simple and Straightforward Ones: To investigate the possibility of designing a simple and straightforward approaches whose performance is comparable to *code2vec*, we propose a heuristics based approach called *HeMa*. Evaluation results in Section V-E suggest that this simple approach significantly outperforms the state-of-the-art ML-based *code2vec*. The finding is consistent with recent reports by Fu and Menzies [46] and by Liu et al. [44] in that complex approaches are not necessarily better than simple and straightforward ones. The empirical study conducted by Fu and Menzies [46] suggests that simple Differential Evolution (DE) and SVM are comparable to (if not better than) complex convolutional neural network (CNN) in retrieving linkable questions from Stack Overflow. The empirical study conducted by Liu et al. [44] suggests that the classical k-Nearest Neighbor model (KNN) works even better than advanced state-of-the-art deep learning models in generating commit messages. Our evaluation, as well as existing reports [44], [46], suggest that simple and straightforward approaches are worth a shot before complicated and time-consuming techniques are exploited for software engineering tasks.

4) Coining Method Names with Tokens Outside Method Body: As suggested by the evaluation results in Section IV-E, 52% of method names in testing set are not used as method names in the large-scale training set. Consequently, method name recommendation approaches, e.g., *code2vec*, that encode method names as a whole, fail to generate such names unseen in their bodies. A potential way to solve this issue is to encode tokens instead of whole method names, and to coin method names at fine-grained token level. However, we also notice from Table II that only 60.14%=581,237/(385,225+581,237) of the method name tokens could be copied from the input (associated method bodies). Consequently, coining method names with tokens from the method body only could be insufficient. Enlarging the search scope is potentially a good solution: up to 81% of the method name tokens can be found within the enclosing class, and up to 94% can be found within the enclosing package.

B. Limitations

The first limitation of the empirical study is that only one state-of-the-art approach (i.e.,*code2vec*) is involved in the study. Notably, ML-based approaches, especially deep learning based ones [6], [7], are often time and resource consuming. Consequently, involving more baselines in the study could significantly increase the cost in both computing resource and human resource of manual analysis. For example, we evaluated convolutional attention network [7] with the employed datasets, and it failed to generate complete results in several days. We only select *code2vec* for evaluation because it represents the state of the art and proved significantly better than other approaches [9]. However, in future work, involving more baseline approaches in the empirical study may increase the generality of conclusions drawn on the empirical study.

The second limitation is the limited usefulness of alternative approach proposed in Section V. We design this approach to intuitively reveal the state of the art as well as the possibility of designing simple but effective approaches. Evaluation results in Section V-E suggest that it has successfully accomplished its mission. However, it should be noted that most of method names it recommends successfully are associated with simple methods, like getter/setter and delegations. Developers rarely need help in naming such simple methods, which may suggest that usefulness of the approach is limited.

VII. Conclusions and Future Work

Researches have recently achieved significant advances in machine learning techniques. As a result, many of the resulting advanced machine learning techniques are exploited to solve software engineering tasks, e.g., code completion and method name recommendation. Although machine learning based approaches are proved accurate in generating method names, existing evaluations fail to reveal their real performance for various reasons, e.g., arbitrary empirical settings. Such evaluations fail to reveal where and why existing approaches work or do not work. To this end, in this paper we conduct an empirical study on the state-of-the-art method name recommendation approach *code2vec* with more realistic settings. Our evaluation results suggest that *code2vec* deserves significant improvement. To intuitively reveal the state of the art and to investigate the possibility of designing simple and straightforward alternative approaches, we also propose a heuristics based approach to recommending method names according to given method bodies. Our evaluation results suggest that it outperforms *code2vec* significantly, and improves precision and recall by 65.25% and 22.45%, respectively.

Implication of the empirical study is severalfold. First, empirical setting is critical for empirical study, and improper setting can significantly warp conclusions drawn on the empirical study. Second, the usefulness of automatic software engineering tools should not be assessed solely by their performance, e.g., precision and recall. Third, advanced and complex approaches are not necessarily better than simple and straightforward approaches. Finally, machine learning based recommendation of method names may not work as well as expected. With more rigorous settings, the deep learning based *code2vec* fails frequently. However, in this paper we empirically evaluate *code2vec* only, and other machine learning based approaches [6]–[8], [13] may outperform *code2vec* in our settings. In future, it should be interesting to investigate this question.

Future work is needed to investigate the correlation between difficulty in naming method names and source code metrics. In this paper, we measure the difficulty subjectively by asking developers to give a number ranking from 1 to 5 to represent the difficulty. Such subjective ranking could be inaccurate. In future, it would be interesting to investigate more accurate and more objective ways to measure the difficulty in naming methods. In future, it is also interesting to investigate whether clone detection techniques could be exploited to retrieve similar methods in corpus, and whether such techniques could outperform the heuristics presented in this paper (Section V).

Acknowledgment

The authors would like to say thanks to the anonymous reviewers for their valuable suggestions.

The work is partially supported by the National Natural Science Foundation of China (61690205, 61772071, 61722202) and the National Key Research and Development Program of China (2016YFB1000801).

References

[1] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1007/s11219-006-9219-1

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, June 2006, pp. 3–12.

[3] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings 10th International Workshop on Program Comprehension*, June 2002, pp. 271–278.

[4] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension an empirical study," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 55–65. [Online]. Available: https://doi.org/10.1109/ICPC.2017.27

[5] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 294–317. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_14

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

[7] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2091–2100. [Online]. Available: http://proceedings.mlr.press/v48/allamanis16.html

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 404–419. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192412

[9] ——, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

[10] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 217–227.

[11] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196332

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[13] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00019

[14] Caprile and Tonella, "Restructuring program identifier names," in *Proceedings 2000 International Conference on Software Maintenance*, Oct 2000, pp. 97–107.

[15] D. Lawrie, H. Feild, and D. Binkley, "An empirical study of rules for well-formed identifiers: Research articles," *J. Softw. Maint. Evol.*, vol. 19, no. 4, pp. 205–229, Jul. 2007. [Online]. Available: http://dx.doi.org/10.1002/smr.v19:4

[16] A. Thies and C. Roth, "Recommending rename refactorings," in *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808921

[17] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[18] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, "Investigating the use of code analysis and nlp to promote a consistent usage of identifiers," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2017, pp. 81–90.

[19] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 111–124. [Online]. Available: http://doi.acm.org/10.1145/2676726.2677009

[20] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276517

[21] I. Avazpour, T. Pitakrat, L. Grunske, and J. Grundy, "Recommendation systems in software engineering," *Dimensions and metrics for evaluating recommendation systems*, pp. 245–273, 2014.

[22] K. Mens and A. Lozano, "Source code-based recommendation systems," in *Recommendation Systems in Software Engineering*. Springer, 2014, pp. 93–130.

[23] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 124–134.

[24] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 837–847.

[25] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 207–216.

[26] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[27] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[28] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[29] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 763–773. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106290

[30] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[31] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: A statistical approach," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 416–427. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884873

[32] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[33] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[34] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, “Toward deep learning software repositories,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[35] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI’18. AAAI Press, 2018, pp. 4159–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=3304222.3304348

[36] O. Vinyals, M. Fortunato, and N. Jaitly, “Pointer networks,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’15. Cambridge, MA, USA: MIT Press, 2015, pp. 2692–2700. [Online]. Available: http://dl_acm.gg363.site/citation.cfm?id=2969442.2969540

[37] “Replication package,” https://github.com/Method-Name-Recommendation/HeMa, 2019.

[38] “New dataset,” https://s3.amazonaws.com/code2seq/datasets/java-med.tar.gz, accessed May 5th, 2019.

[39] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, “Lexicon bad smells in software,” in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 95–99.

[40] E. W. Host and B. M. Ostvold, “The programmer’s lexicon, volume i: The verbs,” in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, Sep. 2007, pp. 193–202.

[41] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 165–176. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931051

[42] J.-P. Krämer, J. Brandt, and J. Borchers, “Using runtime traces to improve documentation and unit test authoring for dynamic languages,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’16. New York, NY, USA: ACM, 2016, pp. 3232–3237. [Online]. Available: http://doi.acm.org/10.1145/2858036.2858311

[43] E. Wong and and, “Autocomment: Mining question and answer sites for automatic comment generation,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 562–567.

[44] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, “Neural-machine-translation-based commit message generation: How far are we?” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 373–384. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238190

[45] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, “When code completion fails: A case study on real-world completions,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 960–970.

[46] W. Fu and T. Menzies, “Easy over hard: A case study on deep learning,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106256