# Suggesting Natural Method Names to Check Name Consistencies

Son Nguyen
Univ. of Texas-Dallas, USA
sonnguyen@utdallas.edu

Hung Phan
Iowa State Univ., USA
hungphd@iastate.edu

Trinh Le
U. of Eng. & Tech., Vietnam
trinhlk@vnu.edu.vn

Tien N. Nguyen
Univ. of Texas-Dallas, USA
tien.n.nguyen@utdallas.edu

## ABSTRACT

Misleading names of the methods in a project or the APIs in a software library confuse developers about program functionality and API usages, leading to API misuses and defects. In this paper, we introduce MNire, a machine learning approach to check the consistency between the name of a given method and its implementation. MNire first generates a candidate name and compares the current name against it. If the two names are sufficiently similar, we consider the method as consistent. To generate the method name, we draw our ideas and intuition from an empirical study on the nature of method names in a large dataset. Our key finding is that high proportions of the tokens of method names can be found in the three contexts of a given method including its body, the interface (the method's parameter types and return type), and the enclosing class' name. Even when such tokens are not there, MNire uses the contexts to predict the tokens due to the high likelihoods of their co-occurrences. Our unique idea is to treat the name generation as an abstract summarization on the tokens collected from the names of the program entities in the three above contexts.

We conducted several experiments to evaluate MNire in method name consistency checking and in method name recommending on large datasets with +14M methods. In detecting inconsistency method names, MNire improves the state-of-the-art approach by 10.4% and 11% relatively in recall and precision, respectively. In method name recommendation, MNire improves relatively over the state-of-the-art technique, *code2vec*, in both recall (18.2% higher) and precision (11.1% higher). To assess MNire's usefulness, we used it to detect inconsistent methods and suggest new names in several active, GitHub projects. We made 50 pull requests (PRs) and received 42 responses. Among them, five PRs were merged into the main branch, and 13 were approved for later merging. In total, in 31/42 cases, the developer teams agree that our suggested names are more meaningful than the current names, showing MNire's usefulness.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools.**

## KEYWORDS

Naturalness of Source Code; Program Entity Name Suggestion; Deep Learning


*ICSE '20, May 23–29, 2020, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380926

**ACM Reference Format:**
Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380926

## 1 INTRODUCTION

Easy-to-understand code must have meaningful and succinct identifiers and names for the program entities so that engineers can quickly grasp the key functionality of the code [6]. Importantly, misleading names for program entities in a regular program or the APIs in a software library confuse engineers on API usages, leading to misuses and defects [19]. That could negatively affect other projects that rely on them [6]. Thus, companies have been emphasizing on naming conventions and coding standards [5].

Recognizing the importance of meaningful names, researchers have introduced automated tools to verify the consistency between the methods' names and their bodies, and then to suggest succinct names for inconsistent methods [33], or recommend a meaningful name for checking such consistency [6, 12]. Liu *et al.* [33] follows an **information retrieval** (IR) direction with the key idea that two methods with similar bodies should have similar names. However, in our study, we found that in several cases, two methods with the same bodies are given different names or two of them with the same names have different bodies, because they are in different contexts or for different tasks/purposes (Section 6). Importantly, with the IR direction, their tool searches for the names of the methods with similar bodies to suggest for an inconsistent method. Thus, it cannot suggest a new name that it has not seen before.

Following **machine learning** direction, *code2vec* [12] suggests a name for a given method with the key idea that two methods with similar AST structures should have similar names. However two methods implemented with different AST structures (*e.g.,* `for` versus `while`) can perform the same task, thus, can be given the same name. Importantly, *code2vec* is not capable of generating a new name for a method. Instead, it computes the probability of a given name for a given method body and interfaces. In contrast, Allamanis *et al.* [6] can suggest a new name to a method by projecting all the names in the method body and the tokens of the method name into the same vector space using a neural network model. From the vector space, their model selects nearby tokens to compose a new method name. However, the names of program entities differ by nature from the tokens of method names since the entities' names carry complete meaning, while the individual tokens of a method name do not. Thus, they should not be in the same space.

In this paper, we introduce MNire, a machine learning approach to check the consistency between the name of a given method *m* and its implementation. Based on the contexts of the body, the interfaces, and the enclosing class' name, MNire first generates a name and

compares $m$'s current name against it. If they are sufficiently similar, $m$ is considered as consistent, otherwise it is inconsistent.

To infer the method name, we use the principle of naturalness of software [27] that is applied on the tokens of entities' names. That is, the tokens composing the name of a method and the names of entities within its contexts are not chosen randomly. The entities' names appear regularly and naturally, and contain certain tokens due to the intent of developers in implementing the functionality of the method. Meanwhile, the functionality is captured abstractly via a succinct method name. With that principle in mind, we first conducted an empirical study to learn the nature of method names and especially the relation between the tokens composing the names of the entities in the contexts and the tokens of the method names. Our results give the empirical foundation on whether the contexts have impacts on predicting method names.

Our study was performed on a large dataset used in the prior work [8] consisting of more than 17M methods in +14K highly-rated Github projects (Section 3). We found that 62.9% of the method names are unique. In contrast, 78.1% of *the tokens* as part of the method names can be found in the previously seen method names. Thus, a method name suggestion model should work at the level of tokens of method names, rather than at entire method names. Moreover, high proportions of the tokens of method names can be found in the contexts of bodies, interfaces, and enclosing classes of the methods. When encountering all the tokens of the names of the entities used in a method's body, in 35.9% of the cases, we could see a token in the method's name. Even if the tokens are not found in the contexts, one could use the contexts to predict the tokens in the method names due to the high probabilities of the co-occurrences of those tokens. The reasons are that while the implementation represents the method body, the interface reflects its input/output. Hence, the method could be named to reflect somewhat its input/output. The enclosing class provides the general context of task/purpose in which the method is realized. In brief, our results provide an empirical evidence to confirm the principle of naturalness of software [27] that also holds for the tokens composing the names of program entities. That is, the tokens are repetitive and the basis of such repetitiveness/regularity of those tokens can be captured by a statistical model that is trained on a large code corpus.

Based on the results of our study, we developed a method to suggest a method name. *We consider the method name as the abstractive description on the method's functionality.* The problem of generating a method name is treated as *the abstractive text summarization.* Each sentence is the sequential representation of the tokens in a context. The method's name is broken into a sequence of tokens, which is generated as a summary of the input sentences. To create an abstractive summary for a method, we choose Encoder-Decoder [17]. The model statistically produces the encode of the input to summarize the essence of the sentences. The model is used to capture the contextual sentences and to rephrase them in a short sequence with possibly different tokens, which form the suggested name.

We conducted several experiments to evaluate MNire in method name consistency checking and in method name recommending on two large datasets that have been used in the previous works with 2M and 14M methods, respectively [12, 33]. To avoid bias, we chose these datasets that are different from the dataset used in our empirical study from which we draw our solution. To detect inconsistency cases, MNire outperformed the state-of-the-art approach in Liu *et al.* [33] by 10.4% and 11% relatively in recall and precision, respectively. We found that there exist several cases that Liu *et al.* [33]'s solution of "Similar method bodies lead to similar names" does not work. In those cases, the tokens in other contexts help MNire distinguish those methods and make correct detections. For method name suggestion, MNire improves relatively over *code2vec* [12] in both recall (18.2% higher) and precision (11.1% higher).

Our result also shows that using the representations for source code from lexical tokens, to ASTs (*e.g.,* as in *code2vec*), to Program Dependence Graphs (PDGs), the model has lower accuracies than MNire. This suggests that to recommend a method name, which is the abstract of entire method, using tokens of the names in the contexts as in MNire yields better performance than using ASTs or PDGs, which represent code structure and dependencies.

There are 43.1% of the cases suggested by MNire that exactly match with the correct method names in the oracle, and 5.1% of those cases (*i.e.,* 2.2% of total cases) do not appear in training data. This shows that MNire is able to learn to suggest the method names, rather than retrieving what have been stored and seen in the training corpus. Finally, there are *13.1% of the cases in which the names are not previously seen in the training data.* The precision and recall of this set of generated names are 59.8% and 58.3% respectively. To assess MNire's usefulness, we made 50 pull requests on suggesting a new name for the inconsistent methods detected by MNire, and received 42 responses. Among them, 5 PRs were actually merged into the main branch, and 13 were approved for later merging. In total, there are 31 cases where the developer teams agree that our suggested names are more meaningful than the current names.

In summary, this paper makes the following contributions:

**A. Empirical Study:** Our results confirm and provide empirical evidence for the principle of naturalness of software [27] on the regularity at the token level of program entities' names.

**B. Representation and Tool:** a novel approach/tool to recommend method names and to detect method name inconsistencies. The method name generation is treated as an abstractive summarization of the tokens of the entities' names in the contexts. The intuition is that the method names depend on the names of the program entities to serve the purpose of the method.

**C. Empirical Results:** Our extensive empirical evaluation shows

1) that MNire is useful in detecting inconsistencies in method names and in suggesting meaningful method names for real-world projects. We show that it outperforms the state-of-the-art approaches in both inconsistency detection and method name suggestion.

2) as a surprising finding that for method name suggestion, relying on the regularity of the tokens of the entities' names in the context yields better results than using the code structures (AST) and dependencies (PDG). For detailed results, see our website [1].

## 2 MOTIVATING EXAMPLES

We first present the examples on the method name inconsistency problem, and then discuss the observations motivating MNire.

### 2.1 Method Name Inconsistency

Let us present two typical scenarios of this inconsistency problem. The first scenario is that the inconsistency occurs at the first place, when a misleading or confusing name is given for a method. In

```java
public class ProfilerTimerFilter {
    /**
    * The maximum time the method represented by IoEventType ...
    */
    //Correct method name: getMaximumTime
    public long getMaxValue(IoEventType type) {
        if (!timerManager.containsKey(type))
            throw new IllegalArgumentException("...");
        return timerManager.get(type).getMaximum();
    }
}
```

**Figure 1: A confusing method name in Apache MINA project**

Figure 1, an API method in Apache MINA project [4], which was implemented to get the maximum time, is inappropriately named at commit `49d56328`. The initial name for the method is `getMaxValue`, which does not reflect well the functionality of the method. Hence, the development team decided to change the method name to a more precise method name, `getMaximumTime`; and this change was explained at commit `aadd1fbc` with the message *"Renamed public methods in ProfilertimerFilter to clarify its meaning"*.

In another scenario, the inconsistency between the method name and the method functionality occurs during *software evolution*. That is, code changes during development make the method name no longer consistent with the new implementation. For example, Figure 2 shows the first version of the method named `getHostName` in Apache Cassandra project [2], in which the name reflected well the method's purpose. After a few months of the development, the team changed their design to use IP address in the entire project as explained at commit `f06a9da6`: *"switch to IP everywhere"*. Consequently, the body of this method was changed to return an IP address, instead of a host name as in its previous version (Figure 2). This change leads to that `getHostName` was no longer appropriate for its new implementation shown in Figure 3. Finally, to fix this inconsistency, this method was renamed to have a more appropriate name, `getHostAddress`, as this was explained in the log of the commit `0bf69f8c`: *"rename* `getHostName` -> `getHostAddress` *since it should always be an IP now"*.

The method names such as `getMaxValue` and `getHostName`, that poorly reflect the program behavior, can confuse programmers on the APIs' functionality. Programmers might incorrectly use these APIs. In fact, an empirical study showed that poor method names can affect other projects [6] and even cause software defects [19]. Thus, it is important to check name inconsistency for the methods.

## 2.2 Observations and Approach Motivation

In the examples, the new method names, `getMaximumTime` and `getHostAddress`, descriptively summarize the purpose of the methods shown in Figures 1 and 3, respectively. That is, *the good name of a method can be considered as the abstract of the meaning/purpose of the method*. If a model aims to derive a good name for the method, that good name can be used to check the current method's name to decide if the inconsistency between the method's implementation and its name occurs or not. Moreover, generating the good name for a method is beneficial not only for suggesting the alternative for the inconsistent name, but also for recommending the appropriate name for the method at the first time when the method was written.

```java
public static String getHostName(){
    if (DatabaseDescriptor.getListenAddress() != null)
        return DatabaseDescriptor.getListenAddress();
    return getLocalAddress().getCanonicalHostName();
}
```

**Figure 2: Method `getHostName` in Apache Cassandra project before commit `f06a9da6`**

```java
// Appropriate method name: getHostAddress
public static String getHostName() {
    InetAddress inetAddr = getLocalAddress();
    if (DatabaseDescriptor.getListenAddress() != null) {
        inetAddr = InetAddress.getByName(
            DatabaseDescriptor.getListenAddress());
    }
    return inetAddr.getHostAddress();
}
```

**Figure 3: After commit `f06a9da6`, the name `getHostName` is no longer appropriate for its new implementation**

To generate a good method name, one can rely on multiple factors. Let us illustrate these factors via the following observations:

**O1.** In a method, the program entities including the variables/-fields/methods that are used/accessed/invoked to implement the method body are often not named randomly. These names usually carry certain meaning that reflects the roles of the program entities which collectively perform the task to accomplish the purpose of the containing method. Therefore, *the method's name, that abstracts the method's purpose, and the names of the program entities used to implement the method's body, have a relation with regard to the description of the method's functionality*. Such relation has two folds. First, the method's name and the name of a variable, field, or method call in the body could share parts relevant to the method's functionality. In the scenario 1, the parts of the good name `getMaximumTime` can be found in the variables' names or method calls in the body, *e.g.*, `getMaximum`, `timerManager`. Second, the tokens composing the good method name and those of the program entities in the body often co-occur (Section 3). This suggests that encountering the tokens of the entities in the method body (referred to as *implementation context*) can provide an indication to predict the tokens of the good method name.

**O2.** The types of the parameters and the return type of a method are also parts of the method declaration. Technically, they describe the method's input and output, and have significant impact on its usage with other entities. Let us call this *the interface context*. Thus, *the interface context can affect the name of the method, and they can be used for the name suggestion*. For example, in the class `FileUtils` of Apache Commons IO library [3], the method `copyURLToFile` takes two parameters `source` and `destination` of the types of `URL` and `File` respectively, and is used to copy bytes from `source` to `destination`. Whereas the method that also copies bytes to a `File` but from `InputStream` is given a different name, `copyInputStreamToFile`. As another example, in the class `FileUtils`, while `readFileToString` returns a `String`, `readFileToByteArray` returns `byte[]`.

**O3.** In object-oriented programming, a method $m$ defines a behavior/action of an object $o$ belonging to a class $C$. This implies that

**Table 1: The Occcurrences of Method Names and Tokens**

| | Method name | Token |
|---|---|---|
| Mean #occurrence | 4.8 | 400.3 |
| Median #occurrence | 1 | 3 |
| #occurrence = 1 | 62.9% | 21.9% |
| #occurrence > 1 | 37.1% | 78.1% |

the object *o* could perform the action described by the name *m* of the method. Therefore, the name *C* of the class for *o* could be considered as the subject that can take the action described by the name *m*. In the scenario 1, an object of the class `ProfilerTimerFilter` can invoke the action of the method `getMaximumTime`. Thus, the class name `ProfilerTimerFilter` is the subject of the action described by the term `getMaximumTime`. As a result, the class (called *enclosing context*) can be used to help infer the name of the contained method.

## 3 EMPIRICAL STUDY

Based on our observations, we conducted an empirical study to answer the following questions on the nature of method names:
**RQ1**: What are the characteristics of method names regarding their uniqueness and sizes?
**RQ2**: How is the relation between method names and the implementation, interface, and enclosing class contexts in a program?

The answers to the questions provide the empirical foundation on whether the tokens/names in contexts could have predictive impacts on the occurrences of the tokens of the method names.

**Data collection and processing**. We used the dataset of 14,317 top-ranked, high-quality, long-history Java projects on GitHub, that was used in a prior work [8]. In this dataset, all duplicated Java files and migrated projects and the forks of the same projects are filtered out. This dataset includes 2,127,355 files and 17,012,754 methods. It has the latest, stable versions of the projects, ensuring the method names at the stable and good standing. For each method in the dataset, we collected the method's name, the parameters' names and types, the return type, the class name, and the names of the variables, fields, and method calls within the body of the method. We tokenized each of those names using Camelcase and underscore naming conventions, and the tokens are normalized to lowercase.

### 3.1 Uniqueness and Sizes of Method Names

In our dataset, there are 3,402,550 unique method names and 120,303 *unique tokens* in method names. On average, there are 2.64 tokens per method name, and the median is 3 tokens. The longest one contains 83 tokens. Meanwhile, the number of tokens in a method body is 17.3 times greater than that of a method name. There are 95% method bodies whose numbers of tokens are 3.0 times greater than the method names. Most of method names (with few tokens) are multiple times shorter than the corresponding method bodies.

As seen in Table 1, for method names, nearly 2 out of 3 names (62.9%) are unique. *This leads to that a high number of method names (≈ 2/3 of the names) cannot be identified by searching in the set of cases that are previously encountered.* The mean value 4.8 of the occurrences of method names is due to a large number of occurrences of common names, *e.g.*, `toString`, `equals`, and `hashCode`.

In contrast, for the tokens as parts of method names, a *token is usually used repeatedly multiple names.* 3 out of 4 tokens (78.1%)

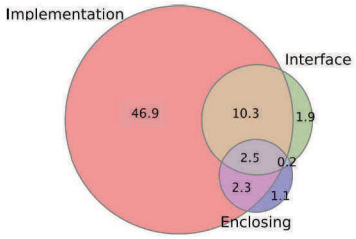


**Figure 4: % of tokens in method names found in contexts**

used to comprise a method name are likely to be previously seen as parts of other method names. Thus, we can conclude that *a method name is often comprised of the tokens that have been previously seen.*

*These above results support us to use a generative summarization approach to learn the tokens composing method names from the tokens that are previously encountered in other method names.*

### 3.2 Method Names and Contexts

*3.2.1* **Common tokens shared between a method name and the contexts**. For a method, we first computed **the percentage of the tokens of the method name that also appear in its contexts**. In Figure 4, on average, for a method, 65.0% of the tokens in its name are found in the names of the program entities in the contexts (the median is 66.7%). Note that the mean and median number of tokens in a method name are 2.7 and 3, respectively. *Thus, on average, about 2 out of 3 tokens of a method name can be found in the three contexts.* As seen in Figure 4, the highest percentage of the tokens in a method name is found in the body (62.0%), while the next ones are in *interface context* (14.9%) and *enclosing context* (6.1%).

The reasons for such trend among three contexts are as follows. First, the *implementation context* usually contains the largest number of tokens, and the number of tokens in a class name is usually less than that of *interface context* (including the names, and the return type and parameters' types). Second, since for a method, its name describes the functionality, while the *implementation context* describes how the functionality is realized, the *implementation context* has the closest relation with the method name. Meanwhile, the *interface context* describes how the method interacts with other entities, thus having a stronger relation with the method name compared to the *enclosing context*, which describes the general context for the method and others in the same class. We will quantitatively analyze their impacts on name suggestion accuracy in Section 6.

We also aim to explore the pervasiveness of the sharing tokens between method names and the contexts. Specifically, **we calculated the percentages of the methods whose names share certain proportions of tokens with the corresponding contexts**.

We found that 84.6% of the methods are given the method names in which at least 33.3% of the tokens (1 out of 3 tokens) are found in the contexts. In 79.8% of the methods, at least half of the tokens in method names are in the contexts. Especially, 36.7% of the methods is comprised of the tokens, such that all of the tokens in method names are in the contexts. Thus, *there are high percentages of the methods whose names share with those of the entities in the context.*

*3.2.2* **The conditional occurrences of tokens in the method names on the contexts**. We investigated the conditional occurrences of the tokens in method names on those of program entities

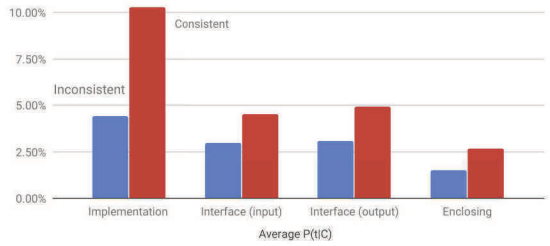


**Figure 5: The average conditional occurrence of tokens in method names on the contexts, $P(t|C)$**

in the contexts. For a method, we computed the conditional occurrence as the conditional probability that the token $t$ is used in the method name given the tokens from the names in the contexts. For a context, the conditional probability of $t$ given context $C$ is computed as: $P(t|C) \approx \frac{Cooccur(t,C)}{Occur(C)}$ where $Cooccur(t, C)$ is the number of methods whose names contain $t$, and their contexts are identical to $C$. $Occur(C)$ is the number of methods whose contexts are identical to $C$. The higher $P(t|C)$, the stronger power the context $C$ provides to predict the token $t$.

We found that on average, the occurrence of a token in method name conditionally on the implementation context is 35.9%. That is, when encountering all the tokens in a method's body, in 35.9% of the cases, we could see a token in the method's name. Meanwhile, the input interface (parameters), output interface (return type), and enclosing contexts provide certain indication to predict the tokens in method names, in which the conditional occurrences of a token in method name on each of those contexts are 18.8%, 17.1%, and 8.3%, respectively. Especially, there is a considerable number of tokens *that are always found in the method names when certain contexts are encountered* (*i.e.*, the case when $P(t|C) = 1$). Specifically, the percentages for the implementation, input interface, output interface, and enclosing contexts are 5.9%, 2.2%, 1.5%, and 0.63%, respectively. For example, the *enclosing context* of `smtp mail sender` always contains the methods whose names contain the token `send`. *Among the cases of $P(t|C) = 1$, the percentage of the cases in which the token $t$ does not appear in $C$ where $C$ is implementation context is 43.6%.* The respective percentages of such cases for *input*, *output*, and *enclosing* contexts are 89.7%, 76.6% and 82.6%. Thus, **even the tokens are not in the contexts, the contexts can be used to predict the tokens in method names** due to high conditional occurrences.

*3.2.3* **Conditional occurrences of tokens in inconsistent and alternative good names on the contexts**. We also study the capability of the contexts in making distinction between a consistent name and an inconsistent name of a method. Specifically, we used a dataset (see Section 5) from Liu *et al.* [33], which contains a set of methods whose names and their bodies are inconsistent, as well as the corresponding good names that were used by real-world developers to replace the inconsistent names of those methods. Figure 5 shows the average conditional probabilities on the occurrences of the inconsistent and alternative good names of the tokens in the method names given each of the contexts. As seen, for all contexts, the average $P(t|C)$ of the tokens in inconsistent method names is relatively much lower than that in the alternative consistent names. *Thus, each context can provide the indication of the occurrences of the tokens in good names more than those in inconsistent names.*

Our results provide empirical evidence to confirm the *principle of naturalness of software [27]* that also holds for *the tokens* composing the names of the methods and the names of program entities in the contexts. The tokens composing the names of the entities in the contexts appear together regularly and naturally due to the intention of developers in realizing the method's functionality. That functionality is captured by an abstract, succinct method name. Thus, the appearances of the tokens in the entities in the contexts can have impact on those of the tokens in the method name. For example, in the method `getHostAddress`, which aims to retrieve the host address, the tokens of the program entities (`inet`, `Addr`, `Local`, `Address`, `Listen`, *etc.*) are relevant to achieve that task. Moreover, our results also confirm the benefits of using code contexts for name prediction as in the prior work for code-to-texts [28].

**Conclusion**. We conclude the following results in our study:

(1) 62.9% of the full method names are unique. For a given method, one **cannot rely solely on searching for a good name** in the data of the previously seen method names.
(2) 78.1% of the tokens in method names can be found in the other previously seen method names.
(3) There are **high proportions of the tokens** (average of 65.0%) of method names which are shared with the three contexts. There are **high percentages of the methods** (79.8%) whose tokens in names share (+50%) with the tokens of the entities' names in the contexts.
(4) When encountering all the tokens of the name of the program entities used in the body of a method, in 35.9% of the cases, we could see a token in the method's name.
(5) Even the tokens are not found in the contexts, one could **use the contexts to predict the tokens in the method names** due to those high conditional occurrences.
(6) Each context provides the indication of occurrences of the tokens in the good names more than in inconsistent names.

## 4 MNIRE: CONSISTENCY CHECKING MODEL

In this work, we propose MNire, a machine learning approach to generate the candidate good name for a given method, and use it to compare with the current method name to check its consistency.

### 4.1 Key Ideas

**Naturalness of Names at Token Level.** To infer a good candidate name, we use the principle of naturalness of software [27] on the tokens of the program entities' names. Our empirical study confirms the principle at the token level. We also draw our ideas from those results. That is, to learn the basis of regularity of the tokens in the names, we rely on statistical learning from the contexts with a large code corpus: the tokens of program entity names regularly co-occurring have higher impact in deciding the tokens of the method name than the less regular ones. Thus, observing the tokens of program entities in the three contexts from a large corpus, MNire can leverage that to derive the most likely tokens in the method names. For inference, MNire focuses on 1) *implementation context* (*how the method is implemented*), including the names of the variables, fields, and methods that are used, accessed, or invoked in the method's body, 2) *interface context* (*the method's input/output*), including the parameters' types and the return type of the method, 3) *enclosing context* (*the enclosing class*), including the class name's tokens.

**Abstractive Summarization.** In MNire, *we consider the method name as the abstract on the method's functionality, which is expressed via the three above contexts.* We treat the problem of generating a method name as the abstractive summarization on the sentences extracted from the tokens of the program entities in the three contexts. As input, for each context, a sentence is formed by the textual tokens of the entities in the context. The method's name is broken into a sequence of tokens, which will be generated as a summary of the sentences for the contexts. The rationale for the choice of abstractive summarization (the summary is an abstract of the texts) over extractive summarization (the summary is the few selective sentences from the texts) has two folds. First, the number of tokens of a method name is much smaller than those of the sentences representing the contexts (Section 3.1). Second, the method name can be a new sequence of tokens with new tokens.

To produce an abstract for a method, we use a machine learning model, called Encoder-Decoder [17]. The model aims to capture the sentences and then rephrases them in a short sequence with possibly different tokens, which form the suggested method name.

## 4.2 Context Extraction

To build the contextual sentences of a method, we first extract the *implementation (IMP)*, *interface (INF)* and *enclosing (ENC)* contexts. Figure 6 shows the contexts extracted from the method `getMaximumTime`. *IMP* is the set of *tokens* of the variables/fields and methods that are used in the method body. Technically, in the AST of the method body, they are *AST simple names*, which are the identifiers being declared/referenced, other than keywords or literals. The examples are `timer manager`, `contains key`, or `type`. Meanwhile, the *INF* context of the method includes two parts: 1) the set *Input* of the tokens from the parameters' types, *e.g.*, {`io event type`}, and 2) the set *Output* of the tokens from the return type, *e.g.*, `long`. Finally, the set *ENC* contains the tokens from the enclosing class name, *e.g.*, `profiler timer filter`. A sequence of tokens for a context is called a *(contextual) sentence.*

All the contextual sentences are then concatenated to form the sequential representation of the three contexts, that are separated by the periods (“.”). In the *INF* sentence, the *Input* and *Output* parts are separated by commas (“,”). For example, the contextual sentence of method `getMaximumTime` is "`profiler timer filter . io event type , long . timer manager ... get maximum`". In the contextual sentences, for *IMP* and *INF*, the tokenized names/types are arranged in the appearance order in the code. We performed experiments in several random orders of names/types for *IMP* and *INF*, and found that the order of names/types does not affect the results.

## 4.3 Abstractive Summarization Model

Figure 7 describes the architecture of the abstractive summarization model, *seq2seq* [17] that we used in MNire. This model is based on an encoder-decoder architecture with attention mechanism [14, 35]. Generally, the model aims to first capture contextual sentences and then rephrase it in short, using possibly different tokens to construct the names for the corresponding methods.

In this model, the encoder takes as input the contextual sentences, which is embedded as the vector $x = (x_1, x_2, ..., x_m)$, and encodes the sentences into a hidden representation $h = (h_1, h_2, ..., h_m)$. The decoder is responsible for predicting the probability of a method

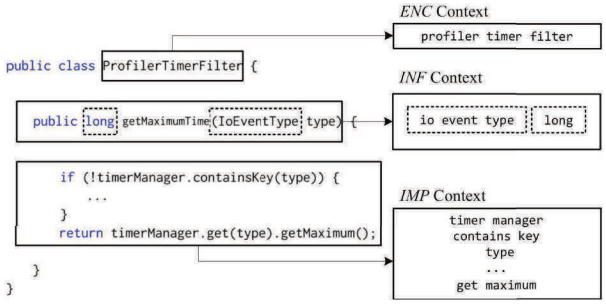


**Figure 6: Context Extraction**

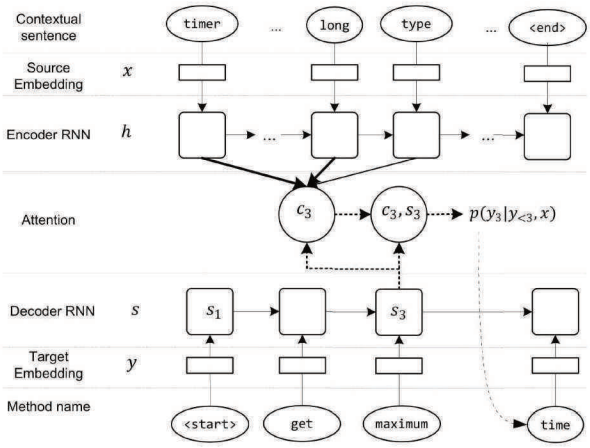


**Figure 7: Abstractive Summarization Model**

name that is expressed as the vector $y = (y_1, y_2, ..., y_k)$ based on the vector $h$. The probability of each token $y_i$ in the method name is predicted based on the recurrent state of the decoder RNN $s_i$, the previous predicted token $y_{<i}$, and a context vector $c_i$:

$$p(y_i|y_{<i}, x) = softmax(W[s_i; c_i] + b)$$

where $c_i$, called the *attention vector*, is calculated based on $s_i$, and the encoder hidden states, $h$: $c_i = \sum_j a_{ij} \times h_j$, and $a_{ij} = \frac{att(s_i, h_j)}{\sum_j att(s_i, h_j)}$ $att(s_i, h_j)$ is an attention function that computes an unnormalized alignment score between the encoder state $h_j$ and the decoder state $s_i$ [17]. Generally, the context vector $c_i$ helps the decoder decide which parts of the contextual sentence to focus on at each generation step to generate $y_i$. For example, the prediction of the third token in the method name depends on $s_3$, the previous generated tokens for the method name: `get` and `maximum`, and the attention vector $c_3$, which is produced based on $s_3$ and $h$. $c_3$ assists the decoder to find the relevant parts in the contextual sentence. In this case, the relevant parts in the contextual sentence are `timer` in *ENC*, `long` in *INF* rather than `type` in *IMP*. Finally, the most likely token in the third position of the method name is `time`. Details on the model *seq2seq* can be found in another document [17].

## 4.4 Method Name Consistency Checking

To check the consistency for a method with the name $c$, we computed the similarity $Sim(p, c)$ between the name $p$ produced by

**Table 2: MCC Corpus for Method Name Consistency Check**

| | Test data | Train data |
|---|---|---|
| #Methods | 2,700 | 1,962,872 |
| #Files | – | 250,972 |
| #Projects | – | 430 |
| #Unique method names | – | 540237 |
| #Occurence>1 | – | 33.5% |

MNire in Section 4.3 and the current name $c$. $Sim(p, c) \in [0, 1]$, is defined as the portion of the tokens that are shared between $p$ and $c$:

$$Sim(p, c) = \frac{numOfSharedTokens(p, c)}{(numOfTokens(p) + numOfTokens(c))/2}$$

where $numOfSharedTokens(p, c)$ is the number of the shared tokens of $p$ and $c$, $numOfTokens(p)$ and $numOfTokens(c)$ are the numbers of tokens of $p$ and $c$. The consistency of the method $m$ is decided using a varied threshold $T$. In particular, if $Sim(p, c) \leq T$, MNire classifies $c$ as *inconsistent*, otherwise $c$ is classified as *consistent* with the method's implementation. Currently, MNire uses lexical similarities. However, semantic and syntactic similarities can be detected by more sophisticated representation techniques such as *word embedding* [31] on the method names.

## 5 EMPIRICAL METHODOLOGY

We evaluated MNire in its main goals of method name consistency checking (*MCC*) and method name recommending (*MNR*). For evaluation, we seek to answer the following questions:
**RQ3**: *Accuracy and Comparison.* How accurate is MNire in method name consistency checking and recommending? and how is it compared with the state-of-the-art approaches for method name consistency checking in [33] and method name recommending in [12]?
**RQ4**: *Context Analysis.* How do the three contexts contribute to MNire's accuracy in MCC and MNR in different settings?
**RQ5**: *Sensitivity Analysis.* How do various factors affect MNire's performance, such as representations, data's sizes, thresholds?
**RQ6**: *Time Complexity.* What is MNire's training/testing time?
**RQ7**: *Usefulness.* How useful is MNire in MCC and MNR?

### 5.1 Datasets

**1. Corpus for Method Name Consistency Checking (*MCC* Corpus.)** For comparison, we used the same corpus as in the state-of-the-art technique for *MCC* in Liu *et al.* [33]. The training dataset (Table 2) from that corpus was collected from the highly-rated, open-source projects from four communities, namely Apache, Spring, Hibernate, and Google. It contains the latest versions of 430 Java projects with at least 100 commits. In total, it has 1,960,872 methods, which were considered by Liu *et al.* [33] as good/consistent names because they selected the methods whose names have been stable for a long time. The testing dataset consists of 2,700 methods in which half of them are labelled as consistent and the other half as inconsistent. For the inconsistent ones, the authors chose the methods whose names have been modified/replaced by developers in the projects for the reasons of confusing or inconsistent names. In each dataset, the duplicated Java files and migrated projects and the forks of the same projects are filtered out.

**Table 3: MNR Corpus for Method Name Recommending**

| | Test data | Train data | Total |
|---|---|---|---|
| **[*Dataset 1*] Comparison Experiment with *code2vec*** | | | |
| #Files | 61,641 | 1,746,272 | 1,807,913 |
| #Methods | 458,800 | 14,000,028 | 14,458,828 |
| **[*Dataset 2*] Experiments for RQ4, RQ5, RQ6, RQ7** | | | |
| #Project | 450 | 9,772 | 10,222 |
| #File | 51,631 | 1,756,282 | 1,807,913 |
| #Methods | 466,800 | 13,992,028 | 14,458,828 |
| **[*Dataset 3*] Live Study on Real Developers** | | | |
| #Project | 100 | | 100 |
| #File | 18,970 | | 18,970 |
| #Methods | 139,827 | | 139,827 |

**2. Corpus for Method Name Recommendation (*MNR* Corpus.)** We compared MNire with *code2vec* [12]. The authors provided the tool, the list of projects and the procedure to collect the training and testing data, without the dataset itself. We followed the same setting and procedure to build the corpus. We collected 10K top-ranked, public Java projects on GitHub. The *MNR* corpus contains about 14.5M methods and 1.8M unique files (Table 3).

In the comparative study, we used the same setting as in *code2vec*. We divided the *MNR* corpus into the training and test data in a ratio that is comparable with the numbers in *code2vec*'s experiments, accordingly to the number of files. Specifically, all the files in all the projects are shuffled and split into 1.7M training and 61K testing files, *i.e.,* 14M training and 458K testing methods.

In the experiments for RQ4, RQ5 and RQ6, we split the corpus based on the number of projects, instead of files. The project-based setting reflects better the real-world usage of MNire where it is trained on the set of existing projects and used to check for a new project. Thus, in *MNR* corpus, we split into training and testing projects such that the ratios of the numbers of training and testing files and methods are comparable to the ratios in the file-based setting in *code2vec* [12]. Finally, we randomly shuffled and split all the projects in the *MNR* corpus into 9,772 training and 450 testing projects (Table 3). The dataset for live study will be explained later.

For a non-bias evaluation, these datasets for empirical evaluation are different from the dataset in our empirical study (Section 3).

### 5.2 Evaluation Setup, Procedure, and Metrics

**Comparative Study.** For each application of *MCC* and *MNR*, we trained each model under study with the respective training dataset and then tested it with the testing dataset accordingly.
**Context Analysis.** For each application, to study the impacts of different contexts, we created different variants of MNire with different combinations of contexts, and measured the performance.
**Sensitivity Analysis.** For each application, we studied the impacts of the following factors: representation, similarity threshold, context and data sizes. We varied them and measured performance.
**Metrics.** For *MCC*, we compared the predicted cases against the ground truth on consistent and inconsistent method names provided as part of *MCC* corpus [33]. For *MNR*, we compared predicted names against the good method names in the *MNR* oracle, which was built as in *code2vec* [12]. To measure the performance in *MCC*, we used the same metrics as in Liu *et al.* [33] including

*Precision*, *Recall*, *F-score*, and *Accuracy* for both inconsistency (*IC*) and consistency (*C*) classes. For *IC* class, $Precision = \frac{|TP|}{|TP|+|FP|}$, and $Recall = \frac{|TP|}{|TP|+|FN|}$. For *C* class, $Precision = \frac{|TN|}{|TN|+|FN|}$, and $Recall = \frac{|TN|}{|TN|+|FP|}$, in which *TP* is true positive (*IC* is classified as *IC*), *FN* is false negative (*IC* is classified as *C*), *TN* is true negative (*C* is classified as *C*), and *FP* is false positive (*C* is classified as *IC*). For both *IC* and *C*, *F-score* is defined as $\frac{2\times Precision\times Recall}{Precision+Recall}$. Accuracy is defined: $Accuracy = \frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|} = |TP| + |TN|$.

For method name recommending (*MNR*), we used the same metrics as in *code2vec* [12] and [6], which measure *Precision*, *Recall*, and *F-score* over case-insensitive tokens. Specifically, for the pair of an expected method name $e$ and its recommended name $r$, the precision, $pre(e, r)$, and recall, $rec(e, r)$ are computed as: $pre(e, r) = \frac{|token(r)\cap token(e)|}{|token(r)|}$, and $rec(e, r) = \frac{|token(r)\cap token(e)|}{|token(e)|}$; $token(n)$ returns the tokens in the name $n$. *Precision*, *recall*, and *F-score* of *the set* of the suggested names are defined as the average ones of all the cases. We also counted exact-matched and case-sensitive ones.

# 6 EMPIRICAL RESULTS

## 6.1 Accuracy Comparison (RQ3)

***6.1.1* Accuracy on Method Consistency Checking (MCC) (Table 4). For the *IC* classification**, MNire's recall and precision are 10.4% and 10.8% relatively higher than Liu *et al.* [33]. Recall in this case refers to detecting inconsistent methods, and precision means precise in detecting the inconsistent name for a given method.

We also found that the key reason for such improvement is the use of program entities' names in MNire. In Liu *et al.* [33], the principle of their method is "methods implementing similar behaviors in their bodies are likely to be given similar names, and vice versa". Given a method $m$ whose body is $b$ and name is $n$, during identifying the set $M_b$ of similar method bodies for $b$, to calculate the similarity of two methods, their tool renames all the local variables of the same type `T` to a single name, *TVar* [33]. This increases the similarity of methods that actually implement different tasks. As a consequence, $M_b$ *(the set of methods with similar bodies with m)* is incorrectly expanded, and for an inconsistent method, $M_b$ might overlap with $M_n$ *(the set of methods with similar names)*. Their tool incorrectly considers this case as consistent since $M_b \cap M_n \neq \emptyset$. This leads to that the *inconsistent methods* might not be classified as *inconsistent* (lower recall), and the predicted *inconsistent methods* might be incorrect (lower precision). MNire does not have this issue because it uses the concrete tokens of the program entities' names, rather than their types.

**For the *C* classification**, MNire detects better consistent names than Liu *et al.* [33] with relatively higher recall (16.6%) and precision (9%). We found that there exist several cases that do not follow the main principle of their technique. There are *consistent* methods which are similarly implemented, however are named differently. For example, in Apache Axiom and Apache Tika, there are two methods having the same body: `return stream;`, however, they are given 2 different names `getInputStream` and `getOutputStream`. Both are consistent. Liu *et al.* [33] considers one of them as inconsistent because one method belongs to the set $M_b$ of the other, but not the set $M_n$. Thus, the recall is lower. In this case, MNire uses as a feature the returned type (either `InputStream` or `OutputStream`) in the interface context to generate the good names and consider both cases as consistent. Moreover, we found other cases in which two methods are named the same, but implemented in different ways for different tasks. For example, in Apache ServiceMix, in two classes `StartCommand` and `StopCommand`, there are two methods with the same name `handle(.)` with different bodies `artifact.start()` and `artifact.stop()`. Their tool considers one of them inconsistent since one method belongs to $M_n$ of the other but not the set $M_b$. MNire uses the class names `StartCommand` and `StopCommand` to generate the good name `handle`, and considers both as consistent.

**Table 4: Consistency Checking Comparison Results (in %)**

| | | *Liu et al.* [33] | MNire |
|---|---|---|---|
| IC | Precision | 56.8 | 62.7 |
| | Recall | 84.5 | 93.6 |
| | F-score | 67.9 | 75.1 |
| C | Precision | 51.4 | 56.0 |
| | Recall | 72.2 | 84.2 |
| | F-score | 60.0 | 67.3 |
| Accuracy | | 60.9 | 68.9 |

**Table 5: Name Recommending Comparison Results (in %)**

| | *code2vec* [12] | MNire |
|---|---|---|
| Precision | 63.1 | 70.1 |
| Recall | 54.4 | 64.3 |
| F-score | 58.4 | 67.1 |
| Exact Match | – | 43.1 |

***6.1.2* Accuracy on Method Name Recommendation (*MNR*) (Table 5).** As seen, our tool MNire achieves relatively higher than *code2vec* [12] in both recall (18.2%) and precision (11.1%). First, with higher recall, MNire generates a method name covering more correct tokens than the name created by *code2vec*. Second, with higher precision, the proportion of the correct tokens in the name generated by MNire is also higher.

Analyzing the results, we found a reason for *code2vec* to have a lower recall. With encoding code structures, it requires two methods to have similar structures to have similar names. Thus, two methods that are realized in different structures are not likely to be similarly named by *code2vec*. In fact, source code with different structures can have similar names. In this case, *code2vec* recommends two different names. This leads to *lower recall* of *code2vec* than MNire.

We also investigated the reason for MNire's higher precision over *code2vec* [12]. There are cases in which two methods are realized in the same structure, but are named differently since they are in different classes for different purposes (*e.g.*, different types of files). *code2vec* suggested the same name since two methods have the same body. However, MNire takes richer contexts, *e.g.*, the enclosing class context, whose name has a strong correlation with the method names (Section 3). Thus, it improves precision over *code2vec*. Interestingly, **43.1%** of the cases suggested by MNire are exactly matched with the correct method names in the oracle.

***6.1.3* Generative Capability to Infer Previously Un-seen Method Names.** We studied MNire's performance on the suggested method names that *were not* in the training data. There are +60K (13.1%) out of +460K generated cases that were *not* seen during

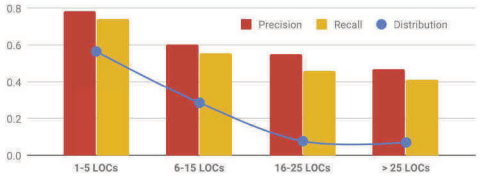


**Figure 8: Accuracy by different Methods' Sizes in Test Set**

**Table 6: Impact of Contexts on *MCC* Results (in %)**

| | | IMP | IMP+INF | IMP+ENC | IMP + INF + ENC =MNire |
|---|---|---|---|---|---|
| IC | Precision | 60.2 | 61.7 | 61.0 | 62.7 |
| | Recall | 90.0 | 92.1 | 91.3 | 93.6 |
| | F-score | 72.1 | 73.9 | 73.1 | 75.1 |
| C | Precision | 53.2 | 55.1 | 54.1 | 56.0 |
| | Recall | 79.3 | 82.3 | 80.6 | 84.2 |
| | F-score | 63.7 | 66.0 | 64.7 | 67.3 |
| Accuracy | | 62.1 | 65.2 | 64.2 | 68.9 |

**Table 7: Impact of Contexts on *MNR* Results (in %)**

| | IMP | IMP+INF | IMP+ENC | IMP+INF+ENC = MNire |
|---|---|---|---|---|
| Precision | 49.7 | 63.2 | 54.4 | 66.4 |
| Recall | 43.3 | 57.8 | 48.9 | 61.1 |
| F-score | 46.3 | 60.4 | 51.5 | 63.6 |
| Exact Match | 20.2 | 34.7 | 25.7 | 43.1 |

training. The precision and recall of this set of generated ones are 59.8% and 58.3%, respectively. Especially, in 16.8% of these generated cases (i.e., 2.2% total cases), the generated names exactly match with the expected ones in the oracle. Thus, MNire predicts well even the un-seen method names. This shows that it learns to suggest the method names, rather than retrieving what have been stored in the training corpus.

*6.1.4* **Accuracy by the Sizes of Methods in Test Set.** We also studied MNire's accuracy on the methods with different sizes. In Figure 8, as expected, MNire works well on the methods with the regular sizes (1–25 LOCs). Even on the longer methods (+25 LOCs), MNire's accuracy decreased gracefully with the precision and recall of 47.0% and 41.4% respectively. This shows that it is harder to capture the functionality of a longer method. Especially, in 7,826 cases out of about 31K long methods (+25 LOCs), the suggested names exactly match with the correct names (see our website [1]).

## 6.2 Context Analysis Results (RQ4)

As seen in Tables 6 and 7, additionally using *interface context (INF)* with *implementation context (IMP)* provides accuracy improvements in both applications. Specifically, for *MNR*, precision and recall significantly increases from 49.7% to 63.2% and from 43.3% to 57.8%, *i.e.*, relatively increases 27% and 33.5%, respectively. This means that when we used both bodies and interfaces, the proportions of the correct tokens in a generated name was improved 27%, while 33.5% more correct tokens are found in comparison with the case of using only IMP. Meanwhile, for *MCC*, there are slightly improvements in the precision and recall for both classes, *IC* and *C*, resulting the improvements in F-score of 2–3% for both *IC* and *C*. Note that the results in Tables 6–7 are in incomparable metrics and obtained when running the models in different datasets (Section 5.1).

**Table 8: Impact of Representation on *MCC* Results (in %)**

| | | Lexeme | AST | Graph | MNire |
|---|---|---|---|---|---|
| IC | Precision | 59.0 | 57.2 | 55.3 | 62.7 |
| | Recall | 88.3 | 85.6 | 80.3 | 93.6 |
| | F-score | 70.7 | 68.6 | 65.5 | 75.1 |
| C | Precision | 47.1 | 46.2 | 45.8 | 56.0 |
| | Recall | 78.2 | 73.5 | 72.1 | 84.2 |
| | F-score | 58.8 | 56.8 | 56.0 | 67.3 |
| Accuracy | | 52.0 | 51.1 | 50.5 | 68.9 |

**Table 9: Impact of Representation on *MNR* Results (in %)**

| | Lexeme | AST | Graph | MNire |
|---|---|---|---|---|
| Precision | 29.5 | 23.1 | 16.2 | 50.6 |
| Recall | 25.1 | 29.2 | 30.3 | 45.1 |
| F-score | 27.1 | 25.9 | 21.1 | 47.7 |
| Exact Match | 9.1 | 8.1 | 4.7 | 22.1 |

Let us explain an example of the impact of *INF*. The method `setRowsPerPage` was identified as having an unclear name by its developers. Using only *IMP*, MNire suggested the name `getItemsPerPage`. It correctly predicted 3 tokens, however, it failed to predict token `set`. Meanwhile, using *IMP+INF*, it can suggest the correct name `setItemsPerPage`. The reason is that it can associate the return type of `void` and parameter's type of `int`, with the token `set`.

Adding enclosing class context *ENC* to *IMP*, precision and recall of name suggestion improve 9.5% and 12.9% relatively. Meanwhile, for consistency checking, precision and recall slightly increase for both *IC* and *C* classes (+1.5% relatively). For example, if two methods with the same body, *IMP* model could incorrectly produce the same name. However, relying on the class names for different tasks, *IMP+ENC* can correctly suggest the names for both methods. As a result, both methods are correctly considered as consistent.

Compared to *IMP+INF*, the improvement of *IMP+ENC* over *IMP* is lower. The reason is that *INF* has a better predictive relation with method name in term of the proportion of shared tokens and the occurrences of tokens in method name conditionally depending on a context. Note that, the numbers of tokens in *INF* and *ENC* are few and much smaller than that in *IMP* (*INF* and *ENC* do not contain many entities). Thus, the improvements over *IMP* are relatively small (Table 6). With all contexts, MNire achieves highest accuracy. The impacts of contexts' sizes and quality are in Section 6.3.

## 6.3 Sensitivity Results (RQ5)

**1. Accuracy with Different Representations.** MNire parses the code and built different representations before using *seq2seq* model.

1) *Lexeme*: all the tokens in the body are collected.

2) *AST*: the method's body is parsed to build the AST; the input to the seq2seq model is the sequence of the tokens in the AST with the use of the delimiters to encode the tree structures.

3) *Graph*: the method's body is parsed to build the PDG. All the PDGs in the entire corpus are converted to vectors using a tool named *graph2vec* [38]. All the vectors are then fed to *seq2seq*.

As seen in Tables 8 and 9, the more sophisticated representations are used, the lower accuracies in both *MCC* and *MNR*. Comparing *AST* model with *Lexeme* model, for *MNR*, precision decreases 21.7% relatively, whereas recall increases only 13.6%. This is reasonable considering the similarity conditions of two models. If two methods have the same sequence of lexical tokens, they will have the same

**Table 10: Impact of Contexts' Sizes on *MNR* Results (in %)**

| | 1–10 tokens | 10–20 tokens | 20–30 tokens | +30 tokens |
|---|---|---|---|---|
| F-score | 35.9 | 41.1 | 43.2 | 51.0 |

**Table 11: Impact of Tokens' Lengths in Contexts on *MNR***

| | 0–80% | 80–90% | 90–95% | +95% |
|---|---|---|---|---|
| F-score | 37.0 | 39.4 | 42.5 | 48.5 |

**Table 12: Impact of Training Data's Size on *MNR* results (%)**

| # Projects | 1.0K | 2.5K | 5.0K | 7.5K | 9,772 |
|---|---|---|---|---|---|
| Precision | 41.1 | 56.2 | 63.1 | 64.9 | 66.4 |
| Recall | 47.8 | 53.7 | 57.6 | 59.5 | 61.1 |
| F-score | 44.2 | 54.9 | 60.2 | 62.0 | 63.6 |
| Exact Match | 19.9 | 29.4 | 34.8 | 36.9 | 38.2 |

AST, thus will be given the same name by both models. However, if two methods have the same AST, they might not necessarily have the same sequence of lexical tokens. Thus, the *Lexeme* model has stricter similarity condition, leading to its higher precision but lower recall. However, the F-score of the *AST* model is still lower than the *Lexeme* model's. Similarly, the *Graph* model has lower precision and higher recall than the *Tree* model because *Tree* model has a stricter similarity condition than *Graph* model. F-score of the *Tree* model is still higher than that of the *Graph* model.

The fact that MNire achieves highest accuracies shows that *the representations of the method bodies such as text, tree, graph, are important for code structures, however for method naming, the naturalness factor, i.e., the program entities' names are more deciding factors.*

**2. Impact of Contexts' Sizes on Accuracy.** We aimed to measure the impact of the size of contexts on accuracy. For MNR, we divided the training data into four buckets of the methods with different sizes of their contexts: the buckets with 1–10, 10–20, 20–30, and +30 tokens in the contexts. Each bucket contains 3M methods and are used as training. As seen in Table 10, with longer contexts (having more tokens), MNire performs better as it has seen more possible tokens. Thus, using training data with longer contexts is better.

**3. Impact of Lengths of Tokens in Contexts on Accuracy.** While the previous study focused on contexts' sizes, this one focuses on the tokens of the entities' names in the contexts. We divided the methods in the training data into buckets according to the percentage of the tokens in its contexts that have more than one character. The rationale is that the higher this percentage, the more meaningful names in the contexts, *i.e.,* the higher quality of the contexts. Each bucket has 2M methods for training. As seen in Table 11, *accuracy increases much when contexts contain more meaningful tokens/names.* Thus, for training, one should select the methods with more tokens of the names having more than one character.

**4. Impact of Training Data's Size on Accuracy.** We varied the training data size by adding more data to a smaller size. As seen in Table 12, the accuracy for *MNR* increases when we increase the data size. In particular, precision and recall increase relatively more than 50% and 20.5%, respectively, when data is increased from 1.0K to 5.0K projects because more tokens can be found to form good method names. Meanwhile, the increasing trend slows down when data size is increased from 5K to 9.7K projects. The reason is that the added data does not contain much more new names as in the smaller sizes. The same trend is for the impact on *MCC* (not shown).

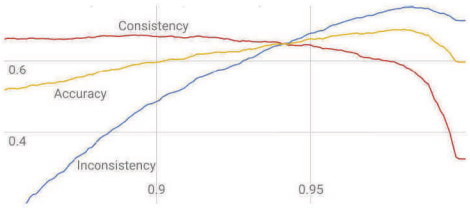


**Figure 9: Impact of Threshold on MCC results**

**5. Impact of Threshold for *MCC*.** We varied similarity threshold $T$ (Section 4.4) from 0.85 to 1.0 (Section 4.4). As seen in Figure 9, both the F-score of *IC* and the accuracy are highest when $T = 0.98$, while when $T = 0.89$, the F-score of $C$ is maximized. The point which balances the F-scores of *IC* and $C$ at 64.7% is $T = 0.94$.

## 6.4 Time Complexity (RQ6)

All experiments were run on a Windows workstation with 16 Intel Xeon 3.7GHz processors, 32GB RAM, and a single Quadro P5000. For *MCC*, MNire took 3 hours for training, and classified with a rate of 530 methods/second. For *MNR*, it took 15 hours for training, compared to 30 hours by *code2vec* [12] with a higher performance GPU (Tesla K80). While MNire collects the tokens of program entities' names, *code2vec* traverses and builds paths along AST nodes to capture code structures, which requires much more computation in training. With a lower computation power machine, our prediction rate is 700 methods/sec compared to 1K methods/sec of *code2vec*. In brief, MNire is also more efficient than *code2vec* [12].

## 6.5 Live Study on Real Developers (RQ7)

To evaluate MNire's usefulness, we conducted a study on active open-source projects in which we submitted pull requests (PRs) of method renaming suggested by MNire and assess PR acceptance rates. To train MNire, we used the 10K top-ranked Java GitHub projects from *code2vec* [12] (dataset 2, Table 3). To use as test dataset (dataset 3, Table 3), we selected 100 active Java open-source GitHub projects that were not in that training set. We ran MNire on the projects to detect inconsistent method names and provide alternative names. Overall, MNire identified 3,682 out of 133,827 methods as inconsistent in the testing dataset. To avoid much work for developers, we randomly selected only 50 cases of inconsistent method names and suggested names. We performed method renaming refactoring in the projects and submitted the changes as pull requests.

As seen in Table 13, 5 cases were merged by the development teams. Additionally, 13 PRs have been validated and approved by the team members. For those cases, the developers acknowledged that the current names are misleading and confusing, and welcomed the suggested names as providing more meaningful names. However, the PR changes have not been merged at the time of this writing since the teams require additional tasks, *e.g.,* reviewing and unit testing before PRs can be merged to the main branch of a project.

In 13 (=5+8) cases, the development team agreed that the method names are not intuitive and our names are more meaningful, however, they cannot approve the renaming at this point due to different

**Table 13: Results on Pull Requests of Real-world Projects**

| Agree | | Agree - but Not Fix | | | | |
|---|---|---|---|---|---|---|
| Merged | Approved | Cannot Fix | Not Fix | Disagree | No answer | Total |
| 5 | 13 | 5 | 8 | 11 | 8 | 50 |

reasons. In 3 cases, the project temporarily do not approve the "non-functional" changes (*e.g.*, refactoring). In 2 cases, the method under study is an overridden one from an external library. In 6 cases, the developers, despite agreeing with the suggested names, stated that they will not change them because the new names do not conform to the convention in the projects. Two cases are the auto-generated names. There are 11 cases where the developers disagree with our suggested names, and in other 8 cases, we did not get responses by the time of this writing. In brief, in 31/42 cases, the developers agree that our suggested names are more meaningful than the current names. This shows that MNire is useful in real-world projects.

**Threats to Validity.** Our data has only Java code. For code2vec [12], we used the same metrics for comparison (*i.e.*, the accuracy metrics for a set of method names are the average of those for individual names). We did not have a statistical test to compare with MNire since they did not provide individual resulting names, and running their tool requires a high-computational power machine.

## 7 RELATED WORK

MNire is related to the work on **suggesting method names** [6, 7, 12, 33]. Liu *et al.* [33] relies on the principle that methods with similar bodies have similar names. Our experiment showed that several cases violate that principle in checking consistencies. When comparing the methods' bodies, they abstract the variables' names and keep their types. In contrast, MNire uses the names for inferring a good method name. Similar to Liu *et al* [33], Jiang *et al.* [30] use IR with the heuristics to search the methods having similar return type and parameters to derive method names. *code2vec* [12] is a vector representation for source code that is applied to method name suggestion. In comparison, *code2vec* [12] is not a generative model as in MNire. It computes the probability for a given name. Moreover, *code2vec* encodes code structures in order to predict a method name, and we have shown that it is too strict and less effective in method name suggestion than using entities' names.

Allamanis *et al.* [7] develop a neural network with attention mechanism that uses convolution on the input code tokens to summarize source code into short, descriptive method name-like summaries. In comparison, while that model considers all the code tokens in the source code, MNire focuses on the tokens of the names of the program entities in not only the code, but also the contexts including the interfaces and the enclosing class. Allamanis *et al* [6]'s model projects all the names in the method bodies and the tokens of the method names into the same space. From the vector space, their model selects nearby tokens to compose a new method name. MNire treats them in two separate spaces. Importantly, we show that MNire outperforms *code2vec* [12], which has been shown to perform better than both of those models [6, 7].

There are several approaches to **recover/predict the names or types of program entities** within the method bodies [39, 41, 44, 46]. While JSNeat [44] searches for names in a large corpus to recover variable names in minified code, JSNice [41] and JSNaughty [46] use condition random field and machine translation. Naturalize [5] learns and enforces a consistent naming conventions.

Several works have proposed neural networks for **code representations** for various SE tasks. The modeling ranges from encoding code sequences, to code tree structures, to graph structures such as data flow graph (DFG), control flow graph (CFG), and program dependence graph (PDG). Code Vectors [26] encoded abstractions of traces from symbolic execution using word2vec [36] for a general purpose. *code2vec* [11, 12] uses CNN to build vocabulary-based embeddings for frequent paths on the ASTs. Tree LSTM [43] trained a tree-structured LSTM model with the ASTs of methods. Deep Learning Similarity [45] used Recursive Autoencoders on Identifiers and ASTs, and graph embedding [22] on CFGs to detect code clones. Code2seq [10] also uses a set of compositional AST paths and uses attention to select the relevant paths while decoding. DeepSim [50] used feature values to encode control and data flows into a semantic matrix and a feed-forward neural network to learn code functional similarity. Graph-based Generative Modeling [18] is for general purpose with interleaving grammar-driven expansion steps, graph augmentation and neural message passing.

The other neural-network-based code modeling approaches using neural networks are for specific SE tasks including automated correction for syntax errors [15], fuzz testing [21], program synthesis [13], code clones [32, 42, 49], program summarization [7, 37], code similarity [11, 50], probabilistic model for code [16], and path-based representations [9, 11], code suggestion [27, 37], code mining [8], type resolution [39], pattern mining [20].

Machine learning has been often used to **generate texts/comments from code** [28, 29, 34, 47]. DeepCom [28] has a traversal on AST structure for flattening, and uses seq2seq model to produce code summary. CODE-NN [29] uses LSTM with attention on code sequence to model the conditional distribution of a summary to produce word by word. Wan *et al.* [47] incorporate AST structure and code sequence into a deep reinforcement learning framework. Haije treats this as code-to-text machine translation [25]. Zheng *et al.* [48] uses AST structure for such statistical machine translation to produce comments. Statistical NLP was used to generate code from text, *e.g.*, SWIM [40], DeepAPI [23], Anycode [24], etc.

## 8 CONCLUSION

We introduce MNire, a machine learning approach to suggest a method name and to detect method name inconsistencies. We have concluded the following results. First, to suggest a good name for a method, relying on the naturalness of the program entities in the contexts yields better results than using the AST or PDG structures. Second, method names are quite unique, however, the tokens composing them are repeated frequently. Thus, MNire exploits the regularity of the tokens in program entities' names for method name suggestion. Finally, our generative approach is more effective in producing new names than IR-based search approach in a corpus.

## ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

# REFERENCES


[1] [n.d.]. . https://doubledoubleblind.github.io/mnire/.

[2] [n.d.]. Apache Cassandra. http://cassandra.apache.org/

[3] [n.d.]. Apache Common IO. https://commons.apache.org/proper/commons-io/

[4] [n.d.]. Apache MINA. https://mina.apache.org/

[5] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM Press, 281–293.

[6] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 38–49.

[7] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016 (JMLR Workshop and Conference Proceedings)*, Vol. 48. JMLR.org, 2091–2100.

[8] M. Allamanis and C. Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR'13)*. IEEE CS, 207–216.

[9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[10] Uri Alon, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations (ICLR 2019)*. https://openreview.net/forum?id=H1gKYo09tX

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. *CoRR* abs/1803.09473 (2018). arXiv:1803.09473 http://arxiv.org/abs/1803.09473

[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[13] Matthew Amodio, Swarat Chaudhuri, and Thomas W. Reps. 2017. Neural Attribute Machines for Program Generation. *CoRR* abs/1705.09231 (2017). arXiv:1705.09231 http://arxiv.org/abs/1705.09231

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2014). arXiv:1409.0473 http://arxiv.org/abs/1409.0473

[15] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *CoRR* abs/1603.06129 (2016). arXiv:1603.06129 http://arxiv.org/abs/1603.06129

[16] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016) (Proceedings of Machine Learning Research)*, Vol. 48. PMLR, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

[17] Denny Britz, Anna Goldie, Thang Luong, and Quoc Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *ArXiv e-prints* (March 2017). arXiv:cs.CL/1703.03906

[18] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2018. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490* (2018).

[19] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009)*. 31–35. https://doi.org/10.1109/WCRE.2009.50

[20] Jaroslav M. Fowkes and Charles A. Sutton. 2015. Parameter-Free Probabilistic API Mining at GitHub Scale. *CoRR* abs/1512.05558 (2015). http://arxiv.org/abs/1512.05558

[21] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn and Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 50–59.

[22] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151 (2018), 78–94.

[23] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 631–642.

[24] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, 416–432.

[25] Tjalling Haije. [n.d.]. Automatic Comment Generation using a Neural Translation Mode.

[26] Jordan Henkel, Shuvendu Lahiri, Ben Liblit, and Thomas W. Reps. 2018. Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. *CoRR* abs/1803.06686 (2018). arXiv:1803.06686 http://arxiv.org/abs/1803.06686

[27] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE Press, 837–847.

[28] 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, 200–210. https://doi.org/10.1145/3196321.3196334

[29] 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 (Volume 1: Long Papers)*. Association for Computational Linguistics, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[30] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Automated Method Name Recommendation: How Far Are We. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE'19)*. IEEE CS.

[31] Jey Han Lau and Timothy Baldwin. 2016. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. In *Proceedings of the 1st Workshop on Representation Learning for NLP*. Association for Computational Linguistics, 78–86. https://doi.org/10.18653/v1/W16-1609

[32] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*. 249–260. https://doi.org/10.1109/ICSME.2017.46

[33] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to Spot and Refactor Inconsistent Method Names. In *Proceedings of the 41th International Conference on Software Engineering (ICSE '19)*. ACM, 1–12.

[34] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE'19)*. IEEE CS.

[35] Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. 2014. Addressing the Rare Word Problem in Neural Machine Translation. *CoRR* abs/1410.8206 (2014). arXiv:1410.8206 http://arxiv.org/abs/1410.8206

[36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems 2013 (NIPS'13)*. 3111–3119.

[37] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. *CoRR* abs/1409.5718 (2014). arXiv:1409.5718 http://arxiv.org/abs/1409.5718

[38] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, and Yang Liu. [n.d.]. graph2vec: Learning distributed representations of graphs. ([n. d.]).

[39] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, 632–642.

[40] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 357–367.

[41] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 111–124.

[42] Randy Smith and Susan Horwitz. 2009. Detecting and Measuring Similarity in Code Clones. In *Proceedings of the 2009 International Workshop on Software Clones (IWSC 2009)*. IEEE CS, 28–34.

[43] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

[44] Hieu Tran, Ngoc Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Recovering Variable Names for Minified Code with Usage Contexts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE Press, 1165–1175.

[45] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, 542–553. https://doi.org/10.1145/3196398.3196431

[46] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JS Names. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 683–693.

[47] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. *CoRR* abs/1811.07234 (2018). arXiv:1811.07234 http://arxiv.org/abs/1811.07234

[48] Ming Li Wenhao Zheng, Hongyu Zhou and Jianxin Wu. 2018. CodeAttention: translating source code to comments by exploiting the code constructs. *Frontiers*

*of Computer Science* 13 (2018), 565–578.

[49] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 87–98. https://doi.org/10.1145/2970276.2970326

[50] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 141–151. https://doi.org/10.1145/3236024.3236068