Q1. Please provide complete bibliographic details for Refs. [48] and [49].

Heuristic and Neural Network Based Prediction of Project-Specific API Member Access

Lin Jiang, Hui Liu, He Jiang, *Member, IEEE*, Lu Zhang, and Hong Mei, *Fellow, IEEE*

Abstract—Code completion is to predict the rest of a statement a developer is typing. Although advanced code completion approaches have greatly improved the accuracy of code completion in modern IDEs, it remains challenging to predict project-specific API method invocations or field accesses because little knowledge about such elements could be learned in advance. To this end, in this paper we propose an accurate approach called HeeNAMA to suggesting the next project-specific API member access. HeeNAMA focuses on a specific but common case of code completion: suggesting the following member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. By focusing on such a specific case, HeeNAMA can take full advantages of the context of the code completion, including the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. All such information together enables highly accurate code completion. Given an incomplete assignment, HeeNAMA generates the initial candidate set according to the type of the base instance, and excludes those candidates that are not type compatible with the left hand side of the assignment. If the enclosing project contains assignments highly similar to the incomplete assignment, it makes suggestions based on such assignments. Otherwise, it selects the one from the initial candidate set that has the greatest lexical similarity with the left hand side of the assignment. Finally, it employs a neural network to filter out risky predictions, which guarantees high precision. Evaluation results on open-source applications suggest that compared to the state-of-the-art approaches and the state-of-the-practice tools HeeNAMA improves precision and recall by 70.68 and 25.23 percent, relatively.

Index Terms—Code completion, non-API, deep learning, heuristic, LSTM

1 Introduction

The purpose of code completion is to predict the rest (or a part of the rest) of a statement a developer is typing. Code completion feature provided by modern Integrated Development Environments (IDEs) plays an important role in software development process [1], [2]. The usage data collected from 41 Java software developers suggests that code completion is one of the most commonly used commands [3]. It is executed as frequently as the common editing commands, e.g., *delete*, *save*, *paste* and *copy*.

Code completion is widely and frequently employed for several reasons. First, code completion lightens the amount of memory work required of developers [4]. Second, powerful and accurate code completion tools encourage developers to choose longer and more descriptive identifier names because with code completion tools developers do not have to type in all characters of such names [5]. Third, it helps to reduce the number of characters that should be typed in manually [6]. The benefit of the reduction in manually typed characters is twofold. On one side, it speeds up coding. On the other side, it reduces misspelling. When developers type in source code, especially long identifiers, it is likely that typos are introduced. Because code completion tools greatly reduce the number of characters that should be typed in manually, the likelihood of introducing typos is reduced significantly as well.

In this paper, we focus on a common type of code completion: method invocation and field access completion (other forms of code completion include word completion [7], expression completion [8], method argument completion [9] and statement completion [10]). To improve the accuracy of code completion, a number of powerful code completion approaches have been proposed. The first category of such approaches is based on usage pattern mining algorithms [11], e.g., frequent item mining [2], [12], [13], frequent subsequence mining [14] and frequent subgraph mining [15]. These approaches discover code patterns from source code repositories and make code suggestions by matching the given source code against such patterns.

The second category of code completion approaches is based on statistical language models [16]. Such approaches take the assumption that programming languages are somewhat similar to natural languages, and thus the widely used natural language models could be applied to programming languages as well [7]. The most commonly employed language models include N-gram models [17], [18] and deep neural network based language models [19], [20]. An advantage of such language-model based approaches is that they are generic, and thus they can predict all kinds of tokens (e.g., the next character, identifier, member access or statement).

- *Lin Jiang, Hui Liu, He Jiang, and Hong Mei are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: {jianglin17, liuhui08, meihong}@bit.edu.cn, hejiang@ieee.org.*
- *Lu Zhang is with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China. E-mail: zhanglu@sei.pku.edu.cn.*

Manuscript received 11 Dec. 2019; revised 8 Aug. 2020; accepted 11 Aug. 2020. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Hui Liu.)
Recommended for acceptance by M. Nagappan.
Digital Object Identifier no. 10.1109/TSE.2020.3017794

TABLE 1
Subject Applications

Applications	Domain	Version	LOC
Ant	Software Build	1.10.1	270,028
Batik	SVG Toolkit	1.9	361,429
Cassandra	Database Management	3.11.1	592,595
Log4J	Log Management	2.10.0	236,825
Lucene-solr	Search Engine	7.2.0	1,591,582
Maven2	Software Build	2.2.1	91,760
Maven3	Software Build	3.5.2	169,988
Xalan-J	XSLT Processing	2.7.2	352,787
Xerces	XML parser	2.11.0	216,907

Although such advanced code completion approaches have greatly improved the accuracy of code completion in modern IDEs [12], [21], it remains challenging to predict project-specific API method invocations and project-specific API field accesses [22]. In this paper, we call methods and fields defined within the project under development as project-specific API methods and project-specific API fields, respectively. We also call method invocations and field accesses as member accesses for short in the rest of this paper. According to our empirical study on nine well-known open-source Java applications (as introduced in Table 1), public API member accesses account for less than half (39%=339,866/861,618) of the member accesses in source code, and the majority (more than 60 percent) is project-specific API member accesses. An empirical study conducted recently [22], however, suggests that existing approaches are often significantly less accurate in predicting project-specific API method accesses (what they call intra-project API completions) than public API member accesses. Our evaluation in Section 4 also confirms their conclusion: the accuracy of such approaches in predicting project-specific API member accesses deserves significant improvement.

To this end, in this paper we propose HeeNAMA, a heuristic and neural network based approach to predict project-specific API member accesses. It is challenging to predict project-specific API member accesses in general because little knowledge about such elements could be learned in advance. Consequently, in this paper we focus on a specific but common case of code completion: suggesting the following method call or field access whenever a project-specific API instance is followed by a dot (.) on the right hand side of an assignment (we call them member access on RHS for short). For example, once the developer types in "*String name = person.*", HeeNAMA would suggest "*getName()*" as the next token. We reuse nine open-source Java applications from previous code completion research [7], [18], [23], [24] to conduct our empirical study. They cover various domains such as software build, database management, and search engine. The size (LOC) of subject applications varies from 91,760 to 1,591,582. According to the empirical study, such cases of code completion (i.e., project-specific API member access on RHS) are common, and on average a single project contains 11,522 such cases. By focusing on such a specific case, HeeNAMA can take full advantages of the context of the code completion, including the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. All such information together enables highly accurate code completion.

Given an incomplete assignment, HeeNAMA works as follows to predict the next member access. First, it generates the initial candidate set according to the type of the base instance. Mandelin *et al.* [10] and Gvero *et al.* [8] have proved that such type information is helpful in code completion. Second, it looks for highly similar assignments within the project under development. If successful, it would make suggestions based on the retrieved samples. Third, it filters out candidates that are type incompatible with the left hand side expression of the assignment. After that, it ranks candidates in descending order according to their lexical similarity with the identifier on the left hand side of the assignment. Finally, it leverages a deep neural network to decide whether the top one on the candidate list should be recommended. The evaluation results on nine well-known open-source Java applications suggest that HeeNAMA is more accurate than the state-of-the-art approach as well as the state-of-the-practice tool.

The paper makes the following contributions:

- First, we propose an approach called HeeNAMA to recommending project-specific API member accesses on RHS. HeeNAMA takes full advantages of the context, i.e., the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. It also leverages a neural network based filter to exclude risky predictions, which significantly improves the precision of HeeNAMA. The combination of heuristics and neural network makes for a neat way of learning to avoid precisely the kinds of mistakes that heuristics make. To the best of our knowledge, HeeNAMA is the first one that is specially designed to predict project-specific API member accesses on RHS.
- Second, we implement HeeNAMA, and evaluate it on nine open-source Java applications. The evaluation results show that HeeNAMA is accurate.

The remainder of this paper is structured as follows. Section 2 presents a short overview of related research. Section 3 proposes our code completion approach. Section 4 presents an evaluation of the proposed approach on nine open-source applications. Section 5 provides conclusions and potential future work.

2 Related Work

2.1 Language Model Based Code Completion

N-gram models are well known in the natural language processing community. They were applied to source code for the first time by Hindle *et al.* [7] when they find the repetitiveness and predictability of source code. Based on N-gram, they estimate the occurrence probabilities for code sequences (at the granularity of token) in code corpus, and predict the next token according to the corresponding occurrence probabilities. Allamanis *et al.* [17] build a giga-token corpus of Java source code from a wide variety of domains to train a n-gram model. The resulting model can successfully deal with token prediction across different project domains. They also find that employing a large corpus in model training can increase the predictive capability of models. SLAMC [18] strengths n-grams with semantic information to present token sequences. Such semantic information includes the token roles, data types, scopes, and structural and data dependencies. It also combines the local context with the global technical concerns/functionality into a n-gram based topic model.

Tu *et al.* [23] find that source code has high repetitiveness not only in the global scope but also in the local scope. Based on this finding, they propose a cache language model by enhancing the conventional n-gram model with an efficient caching mechanism that captures the local repetitiveness of source code. They compute the probability of a sequence of tokens based on a global n-gram model (trained with public corpus) and a local n-gram model (trained with source files in the enclosing folder). Based on this cache language model [23], Franks *et al.* develop an Eclipse plug-in CACHECA [21]. It combines the native suggestions made by Eclipse IDE with suggestions made by the cache language model. The evaluation results suggest that the combination leads to higher accuracy.

The latest advance in N-gram based code completion was achieved by Hellendoorn *et al.* [20]. Based on cached language models, they proposed a nested and cached N-gram model to capture the local repetition within a given scope, and to apply it to the nested sub-scopes. The evaluation results suggest that their approach significantly outperforms existing approaches (both statistical language model based approaches and deep learning based approaches).

Advanced neural networks, e.g., RNN [25] and LSTM [26], have been successfully employed to model source code as well. Raychev *et al.* [19] employ RNN for code completion. They first extract sequences of method calls from large code bases, and learn their probabilities with statistical language models, i.e., RNN, N-gram, or a combination of them. Once given a program with holes, they leverage learned probabilities to synthesize sequences of calls for holes. White *et al.* [27] also apply the RNN language model to source code and show its high effectiveness in predicting sequential software tokens.

To address the enormous vocabulary problem in modeling source code with deep neural networks, Karampatsis and Sutton [28] present a new open-vocabulary neural language model for code that is not limited to a fixed vocabulary of identifier names. They employ a segmentation into subword units, i.e., subsequences of tokens chosen based on a compression criterion. Including all single characters as subword units will allow the model to predict all possible tokens, so there is no need for special out-of-vocabulary handling.

Graph-based statistical language models are successfully employed in code completion as well. Nguyen *et al.* [29] introduce GraLan, a graph-based statistical language model to statistically learn API usage (sub)graphs [30] from a source code corpus. Given an observed (sub)graphs that representing the context of code completion, GraLan recommends the next API by computing the appearance probabilities of new usage graphs. SALAD [30], [31] also employs the graph-based model to represent API usage patterns. Given bytecode and source code, SALAD generates a graph-based model for extracting API sequences from such model. Such API sequences are in turn employed to train a Hidden Markov Model [32] (called HAPI). According to their evaluation, the resulting HAPI is accurate in predicting the next method call.

2.2 Pattern Mining Based Code Completion

It is quite often that a group of related API methods are invoked in some order to accomplish a specific task. By mining code repositories, we may discover such patterns, i.e., the API methods in order [14], [33]. Such patterns, in turn, are employed to recommend the next API method invocation whenever the preceding API method invocations are typed in.

Bruch *et al.* [2] propose three similar intelligent code completion systems that learn API patterns from existing code repositories in different ways. The first system, called FreqCCS, counts API method invocations in code repositories, and recommends the most commonly invoked method as the next API method invocation. The second one, called ArCCS, mines association rules among API method invocations. An example of association rule is "If a new instance of *Text* is created, recommend *setText()*". ArCCS makes code completions based on such association rules. The last and most advanced system, called BMNCCS, adapts the K-Nearest-Neighbor (KNN) [34] machine learning algorithm to manage API patterns. First, it extracts and encodes the context information (including methods invoked on the same base instance) for each API method invocation in the repository as a binary feature vector. With these feature vectors, it computes the distances between the current context and the API example contexts based on Hamming distance. For API methods associated with the resulting nearest contexts, the approach sorts them according to their frequency in repositories, and the most frequently used one is recommended.

CSCC [12], [35] is another powerful pattern mining based code completion system. The major difference between BMNCCS and CSCC is that the latter takes more context information into usage patterns, i.e., all method calls, Java keywords and type names that appear within the four lines prior to the completion location. To speed up the search for patterns, CSCC employs two distance measures to compute the similarities between the current context and the usage contexts mined from repositories. PBN [13] further extends BMNCCS to tackle the issue of significantly increased model sizes. Unlike BMNCCS that uses a table of binary values to represent usages of different framework types, PBN encodes the same information as a Bayesian network. A key consequence is that PBN allows to merge different patterns and to denote probabilities (instead of boolean existence) for all context information.

MAPO [14] combines frequent sequence mining with clustering to summarize API usage patterns from source files. It mines API usage patterns from open source repositories automatically, and recommends the mined patterns and their associated samples on programmer's requests. MAPO also provides a recommender that integrates with Eclipse IDE.

GraPacc [15] extends the mining of API usage patterns successfully to higher-order patterns. It represents each pattern as a graph-based model [30] that captures the usage of multiple variables, method calls, control structures, and their data/control dependencies. The context features of

API methods from code repositories are extracted and used to search for the best-matched pattern concerning the current context.

As a conclusion, such pattern mining based approaches are highly accurate in recommending public API member accesses. However, they rely heavily on the rich invocation histories of the method to be recommended. Consequently, such approaches are often confined to popular public APIs only because there are rich invocation examples of such public API methods in open-source applications whereas it is challenging to collect large numbers of invocation examples of project-specific API methods.

2.3 Type Based Code Completion

Except for pattern mining and statistical language model based approaches, there are also type based approaches that complete code by searching for valid expressions of given data types.

Mandelin *et al.* [10] propose PROSPECTOR to synthesize jungloid code fragments automatically in response to user queries. Jungloids are the chain of method calls that receives a given input object and returns a desired output object. They first construct a jungloid graph from method signatures with every expression corresponding to a path in the graph. Examples of downcasts are then extracted from program corpus as jungloids, converted to paths and added to the graph. Finally, PROSPECTOR searches for the shortest path from the given type to the desired type in the graph and synthesize a complete code fragment with the path. HeeNAMA differs from PROSPECTOR in that PROSPECTOR constructs a code fragment that might consist of multiple statements whereas HeeNAMA recommends a member access only. Another difference is that they leverage different information for code completion: PROSPECTOR leverages type information and examples of downcasts to synthesize code fragments whereas HeeNAMA leverages type information, examples of member accesses, and lexical similarity.

Gvero *et al.* [8] presents a general code completion approach inspired by complete implementation of type inhabitation for typed lambda calculus. Their approach constructs an expression and inserts it at the given location so that the whole program type checks. They introduce a succinct representation for type judgments that merges types into equivalence classes to reduce the search space. They rank potential solutions by preferring closer declarations to the program point and more frequently occurring declarations from a corpus of code. The approach is complete completion [8] because each synthesized expression is complete in that method calls have all of their arguments synthesized. HeeNAMA differes from their approach in that their approach makes the program type check by inserting a complete expression whereas HeeNAMA recommends a single member access.

2.4 Lexical Similarity Between Identifiers

Identifier names chosen by developers convey rich information, and thus they play an important role in program comprehension and source code analysis [36], [37], [38]. As suggested by Lawrie *et al.* [39], there are two main sources of domain information: identifier names and comments. However, many developers do not write comments, so identifier names are critical for program comprehension.

A number of approaches have been proposed to exploit lexical similarity between semantically similar software entities. Liu *et al.* [9] present an empirical study of the lexical similarity between arguments and parameters of methods, and find that many arguments are more similar to the corresponding parameter than any alternative argument. Pradel and Gross [40], [41] exploit the lexical similarity between arguments and parameters to identify incorrect arguments. HeeNAMA also exploits the lexical similarity between semantically similar software entities. It differs from existing approaches in that it exploits the lexical similarity in code completion whereas exiting approaches [40], [41] exploits it in bug detection.

Cohen *et al.* [42] compare different string metrics, i.e., edit distance (also called Levenshtein distance) and cosine similarity, for matching names and records. The edit distance is used in our approach because it is simple and efficient.

3 Approach

3.1 Overview

In this section, we propose a heuristic and neural network based approach (HeeNAMA) for code completion. As stated in Section 1, HeeNAMA is confined to project-specific API member accesses that are defined as follows.

Definition 1 (Project-Specific API Member Access). *A project-specific API member access is a method call or a field access via a base instance whose class type is declared and implemented within the project where the member access appears.*

The base instance for a member access is the instance (object) whose member is accessed. For example, in the member access *a.b.c.d*, the base instance is *c*. For member access *a.b.c*, however, the base instance is *b*. For an incomplete assignment like "*x=a.b.*", HeeNAMA makes prediction only if the type of the base instance (*b* for the example) is declared and implemented within the project where the incomplete assignment is typed in. In the rest of this paper, *member access*, if not especially specified, refers specifically to *project-specific API member access*.

An overview of HeeNAMA is presented in Fig. 1. HeeNAMA is composed of two parts: a sequence of heuristics (notated as H_1, H_2, H_3, respectively) and a neural network based filter. The first part predicts the next member access based on a sequence of heuristics. Whereas the second part decides whether the prediction is accurate enough to be presented to developers.

Given an incomplete assignment (e.g., "*String name = person.*"), HeeNAMA works as follows to predict the next member access:

1) First, it parses the incomplete assignment, and decides whether the base instance (*person* for the illustrating example) is a project-specific API instance. If yes (i.e., the declaration and implementation of the data type of the based instance are found within the enclosing project), it goes to the next step for code completion. Otherwise, HeeNAMA suggests invoking API specific code completion algorithms.

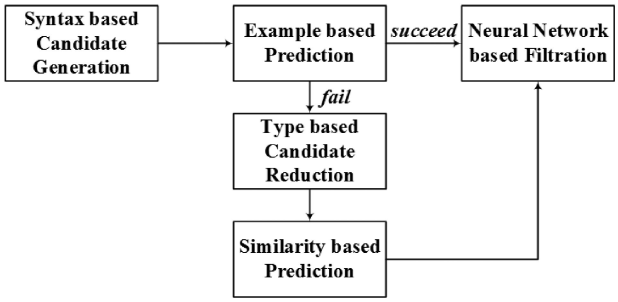


Fig. 1. Overview of HeeNAMA.

2) Second, it generates the initial candidates according to the type of the base instance as well as the location of the incomplete assignment. For the example "*String name = person.*", the initial candidates include all members of the base instance *person* that are accessible on the location where the assignment is typed in.
3) Third, it looks for highly similar assignments within the project under development. If successful, it would predict the next member access based on the retrieved examples and the initial candidates. The prediction is forward to the neural network based filtering (Step 6). If failed, however, it would go to the next step to make prediction with other heuristics.
4) It removes the candidates that are type incompatible with the left hand side expression of the assignment according to our *type compatibility assumption*, i.e., the next member access should be type compatible with the left-hand side expression. For the given example of "*String name = person.*", candidates that are not type compatible with *String* are removed from the candidate set.
5) It ranks the resulting candidates in descending order according to their lexical similarity with the identifier on the left hand side of the assignment ("*name*" for the illustrating example). The one on the top is taken as the most-likely member access.
6) Finally, it leverages a neural network to decide whether the most-likely member access should be recommended. Notably, the most-likely member access is potentially type incompatible with the left-hand side expression if it is predicted by Step 3.

According to the base instance on the right hand side of the incomplete assignment, HeeNAMA decides whether the member completion request is a project-specific API member access. Intuitively, HeeNAMA can make also such decisions according to the type of the left hand side expression as well: If the data type of the left hand side is defined within the project, the right hand side member completion request is a project-specific API member access. However, the decisions made in such a way could be inaccurate. Take "*String name = person.getName()*" as an illustrating example. The type of the right hand side base instance (i.e., *Person*) is project-specific, and thus the member completion request for "*String name = person.*" is project-specific and thus falls in the scope of HeeNAMA. However, the type of the left hand side expression (i.e., *String*) is not project-specific.

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

3.2 Syntax Based Candidate Generation

First of all, HeeNAMA generates initial candidates based on Java syntax. Given an incomplete assignment, we extract its sketch that presents the key information our approach exploits for code completion

$$sketch \ = \ < lType, lName, baseIns, lct >,$$

where $lType$ is the type of left hand side expression, $lName$ is the identifier name of left hand side, $baseIns$ is the base instance, and lct is the location of the assignment.

For the incomplete assignment "*String name = person.*", we have

$$\begin{aligned} lType \ &= '' \ String'' \\ lName \ &= '' \ name'' \\ baseIns \ &= '' \ person''. \end{aligned}$$

If the assignment is outside the package where the type of *person* (i.e., *Person*) is defined, the sketch of the assignment is

$$sketch = \ < \ String, name, person, outside \ > \ .$$

lct indicates the relative location of the incomplete assignment with respect to the type of $baseIns$, i.e., *nested* (nested in the type of $baseIns$), *inherited* (inherited from the type of $baseIns$), *inside* (inside the package of the type of $baseIns$) or *outside* (outside the package of the type of $baseIns$). Consequently, lct can decide what kind of members of the base instance are available at the location. Based on the sketch, we generate the initial candidates $cdtSet$ in two steps. First, we collect all members of the base instance $baseIns$. Second, we remove those members that are not accessible at the location (lct) of the assignment.

For the given example, if "*String name = person.*" is outside the package of class *Person*, the initial candidates are the public members of *Person*. However, if the assignment is within class *Person*, private and protected members of *Person* are taken as initial candidates as well.

3.3 Heuristic 1: Example Based Prediction

Repetitiveness is an important property of source code [7], [43]. Consequently, it is likely that we can predict the next member access based on highly similar member accesses. Algorithm 1 illustrates how HeeNAMA predicts the next member access based on sample assignments within the enclosing project.

First, given an incomplete assignment, HeeNAMA extracts its sketch (noted as $sketch$) that includes the type of its left hand side expression, the identifier name of its left hand side, the base instance, and its location (Line 2). Second, HeeNAMA retrieves sample assignments from the project under development (Line 4). The retrieving process is presented in Algorithm 2. We employ the Java parser provided by Java Development Tools (JDT) to parse source code of the project into Abstract Syntax Trees (ASTs). From such ASTs, we retrieve all AST nodes that represent assignments (Line 3 in Algorithm 2). For an assignment in the application, there may exist multiple scenarios where HeeNAMA can make predictions. For example, for assignment "*String name = this.person.getName()*", HeeNAMA may make prediction when incomplete assignment "*String name = this.*" or "*String name = this.person.*" is typed in. Each of such incomplete assignments and its following member access are presented as a sample:

$$smp = < icpAsgn, memb >,$$

where $icpAsgn$ is the incomplete assignment and $memb$ is the member access that follows the incomplete assignment $icpAsgn$. Line 6 in Algorithm 2 extracts all such samples from a given assignment $asgn$.

Algorithm 1. Example Based Member Access Prediction

```
Input: icpAsgn //incomplete assignment to be completed
       cdtSet  //initial candidate set
       proj    //project under development
Output:member
 1: //construct a sketch of the incomplete assignment
 2: sketch ← constructSketch(icpAsgn)
 3: //extract sample assignments from the project
 4: smpSet ← extractSampleAssignments(proj)
 5: for each smp in smpSet do
 6:   //construct a sketch of the incomplete assignment in
      sample
 7:   skch ← constructSketch(smp.icpAsgn)
 8:   if skch.lType = sketch.lType and
 9:     skch.lName = sketch.lName and
10:     skch.baseIns = sketch.baseIns then
11:     for each cdt in cdtSet do
12:       if cdt = smp.memb then
13:         cdt.frequency ++
14:       end if
15:     end for
16:   end if
17: end for
18: //sort candidates by frequency in descending order
19: sort(cdtSet)
20: if cdtSet[0].frequency > 0 then
21:   member ← cdtSet[0]
22: else
23:   member ← null
24: end if
25: return member
```

Algorithm 2. Extraction of Sample Assignments

```
Input:proj //project under development
Output:smpSet //the set of samples
1: smpSet ← ∅
2: //retrieve all assignments from the project
3: asgns ← retrieveAsgns(proj)
4: for each asgn in asgns do
5:   //extract all samples from the assignment
6:   smps ← extractSamples(asgn)
7:   smpSet.add(smps)
8: end for
9: return smpSet
```

Third, HeeNAMA enumerates samples in the set $smpSet$, and extracts their sketches (Line 7). Lines 8-10 select samples that are highly similar to the incomplete assignment ($icpAsgn$, the first input of the algorithm) by comparing their sketches. A sample is regarded as highly similar to the incomplete assignment when the types of their left hand side expressions, the identifier names of their left hand side expressions and their base instances are the same, respectively. Lines 11-13 count the frequency of the candidate members in the resulting highly similar samples. Based on the frequency, HeeNAMA sorts the candidate set ($cdtSet$) in descending order (Line 19). If the top one in $cdtSet$ has a frequency greater than zero, it is regarded as the most-likely member to be accessed (Lines 20-25).

Taking the incomplete assignment ($icpAsgn$) "*String name = person.*" as an illustrating example, HeeNAMA first extracts its sketch

$$sketch = < String, name, person, outside >,$$

where $outside$ means the assignment is typed in outside the package of class *Person*, and then it retrieves all sample assignments from the project under development. Suppose that it retrieves four sample assignments

$$\begin{aligned} asgn_1: &\quad Stringname = this.person.getName() \\ asgn_2: &\quad Stringname = student.name \\ asgn_3: &\quad Stringname = person.getName() \\ asgn_4: &\quad Stringname = this.person.name. \end{aligned}$$

From these sample assignments, the approach extracts six samples

$$\begin{aligned} smp_{11} &= < skch_{11}, person > \\ &\quad skch_{11} = < String, name, this, outside > \\ smp_{12} &= < skch_{12}, getName() > \\ &\quad skch_{12} = < String, name, person, outside > \\ smp_2 &= < skch_2 , name > \\ &\quad skch_2 = < String, name, student, outside > \\ smp_3 &= < skch_3 , getName() > \\ &\quad skch_3 = < String, name, person, outside > \\ smp_{41} &= < skch_{41}, person > \\ &\quad skch_{41} = < String, name, this, outside > \\ smp_{42} &= < skch_{42}, name > \\ &\quad skch_{42} = < String, name, person, outside > . \end{aligned}$$

Among these samples, smp_{12}, smp_3 and smp_{42} share the same $lType$, $lName$, and $baseIns$ in their sketches with the given incomplete assignment $icpAsgn$, and thus they are taken as highly similar samples. HeeNAMA counts the occurrence frequency of members in the resulting highly similar samples. Because *getName()* has the highest frequency, HeeNAMA suggests to complete the incomplete assignment $icpAsgn$ with *getName()*.

3.4 Heuristic 2: Type Based Reduction of Candidate Set

If the first heuristic H_1 fails, HeeNAMA would generate recommendations with the other heuristics, i.e., the second heuristic H_2 and the third heuristic H_3. To make an assignment syntactically correct, the right-hand side expression of the assignment should be type compatible with the left-hand side expression. Consequently, the predicted member access (for a given incomplete assignment) should be type compatible with the left-hand side expression of the assignment if the member access is the final token of the assignment. However, if the member access is not the final token of the assignment, it is not necessarily type compatible with the left-hand side. When an incomplete assignment is typed in, code completion tools do not know whether the next token is the final token or not. Consequently, in theory code completion tools (including the proposed one) could not assume that the next member access is type compatible with the left-hand side of the assignment.

However, to simplify the prediction, HeeNAMA makes the assumption that the next member access is type compatible with the left-hand side expression (called type compatibility assumption). Although the type of the left-hand side expression is not by definition compatible with the next member access, we find that this is the case in more than 80 percent of member accesses (see Section 4.3.2 for details).

Based on the type compatibility assumption, HeeNAMA reduces the size of the initial candidate set by removing those elements that are not type compatible with the left-hand side expression of the enclosing assignment. If the resulting candidate set is empty, i.e., no candidate is type compatible with the left-hand side, HeeNAMA refuses to make any prediction.

3.5 Heuristic 3: Similarity Based Prediction

Identifiers chosen by developers convey rich information about the semantics of the software entities [39]. An empirical study also suggests that semantically related software entities, e.g., arguments and their corresponding parameters, often have lexically similar identifiers (entity names) [9]. The right-hand side of an assignment is semantically related to its left-hand side, and thus it is likely that they are lexically similar. Consequently, in this section we propose the third heuristic (H_3) to predict the next member access based on the similarity between the candidates and the left hand side variable of the assignment. Given an incomplete assignment (whose sketch is $sketch = < lType, lName, baseIns, lct >$) and its candidate set (*cdtSet*) generated by Heuristic 2, HeeNAMA works as follows to predict the next member access:

- First, for each candidate *cdt* in *cdtSet*, HeeNAMA calculates the Levenshtein distance (notated as $Lev(cdt, lName)$) between the identifier of *cdt* and that of $lName$.
- Second, based on the resulting Levenshtein distance, HeeNAMA calculates the lexical similarity between *cdt* and $lName$ as follows:

$$sim = 1 - \frac{Lev(cdt, lName)}{\max(len(cdt), len(lName))},$$

  where $len(cdt)$ is the length of *cdt* (in characters), $len(lName)$ is the length of $lName$, and $Lev(cdt, lName)$ is th Levenshtein distance.
- Third, HeeNAMA sorts candidates in *cdtSet* in descending order according to their similarities, and suggests to use the the top one as the next member access.

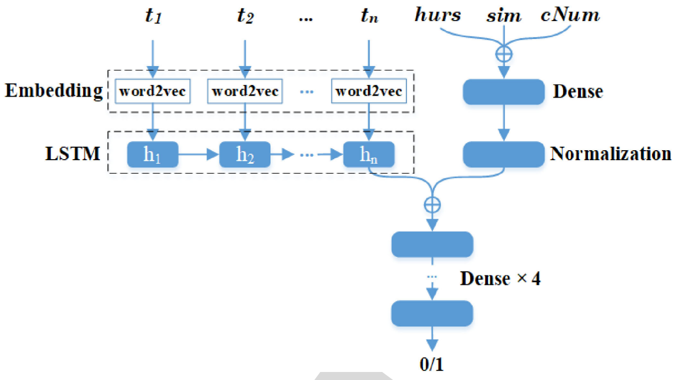


Fig. 2. Structure of the neural network based filter.

For the given example "*String name = person.*", suppose that the candidate set *cdtSet* contains four candidates: *name*, *getName()*, *age* and *getAge()*. HeeNAMA calculates their lexical similarity with the left-hand side variable "*name*". The resulting similarities are 1.00, 0.57, 0.50, and 0.33, respectively. Since the first candidate *name* has the greatest similarity, HeeNAMA recommends *name* as the next token.

3.6 Neural Network Based Filtering

In the preceding sections, we present a sequence of heuristics to predict the next member access according to a given incomplete assignment. In this section, we present a neural network based filtering to filter out risky predictions that are likely incorrect.

The overall structure of the filter is presented in Fig. 2. On the left side is a LSTM [26] layer. Its input is a sequence of identifiers $< lType, lName, baseIns, memb >$ where $lType$ is the type of left hand side expression, $lName$ is the identifier name of left hand side, $baseIns$ is the base instance on which the member is accessed, and $memb$ is the member predicted by heuristics (H_1, H_2, or H_3). To feed such identifiers into the neural network, we take the following measures. First, we tokenize such identifiers (i.e., $lType$, $lName$, $baseIns$ and $memb$) into sequences of tokens according to the camel case naming convention, notated as $st(lType)$, $st(lName)$, $st(baseIns)$, and $st(memb)$, respectively. For the example of $< String, name, person, getName >$, we tokenize them into four sequences of $< String >$, $< name >$, $< person >$ and $< get, Name >$. Second, we lowercase all the tokens and concatenate such sequences as well as separators (notated as *sep*) into one sequence, noted as *cst*

$$\begin{aligned} cst = & < st(lType), sep, st(lName), sep, st(baseIns), \\ & sep, st(memb) > \\ = & < t_1, t_2, \ldots, t_n >, \end{aligned}$$

where n is the total number of tokens in $lType$, $lName$, $baseIns$, $memb$ and separators. Consequently, the resulted sequence for the given example is $< string, sep, name, sep, person, sep, get, name >$.

Given the sequence $cst = < t_1, t_2, \ldots, t_n >$, we map the ith token t_i into a D-dimensional vector $e_i = W \cdot o(t_i)$, where $W \in \mathbb{R}^{D \times V}$ is an embedding matrix pre-trained with *word2vec* model [44] on identifiers that could be collected from sample assignments, and $o(t_i)$ is a one-hot encoder converting t_i into a vector of V dimensions. We embed such tokens into numeric vectors with the well-known *word2vec* because of the following reasons. First, each of the identifiers are finally tokenized into a sequence of words that is essentially a short English phrase, and *word2vec* has been proved effective in vectoring short English phrases. Second, *word2vec* has been successfully employed by Nguyen *et al.* [45] to vectorize identifiers in source code. We pass such vectors into a recurrent neural network with long short-term memory units (LSTM) and compute the hidden vector at the ith time step as

$$h_i = f_{LSTM}(h_{i-1}, e_i), \quad i = 1, \ldots, n, \tag{1}$$

where $h_i \in \mathbb{R}^D$ and f_{LSTM} is the LSTM function. Then we take the final hidden state h_n as the output of LSTM layer. We employ LSTM because of two reasons. First, LSTM has been proved effective and efficient in natural language processing. In our case, all of the involved identifiers (i.e., *lType*, *lName*, *baseIns* and *memb*) are essentially natural language descriptions. Second, LSTM accepts variable-length input. In our case, the length of the input depends on how many words the given identifiers contain, and it may change dramatically from assignment to assignment. Consequently, LSTM fits well for our case.

Algorithm 3. Training Process of the Filter

```
Input: projs //sample projects
       filter //filter (neural network) to be trained
Output: filter //updated filter
 1: smpSet ← ∅
 2: for each proj in projs do
 3:   //extract sample assignments from the project
 4:   set ← extractSampleAssignments(proj)
 5:   smpSet ← smpSet + set
 6: end for
 7: //generate a training set with sample assignments
 8: trainSet ← generateTrainingSet(smpSet)
 9: //train filter with the training set
10: filter ← filter.train(trainingSet)
11: return filter
12:
13: function generateTrainingSet(smpSet)
14:   trainingSet ← ∅
15:   for each smp in smpSet do
16:     //make prediction by heuristics
17:     memb′, hurs, sim, cNum ←
18:         predict(smp.icpAsgn)
19:     skch ← constructSketch(smp.icpAsgn)
20:     input ← < skch.lType, skch.lName,
21:         skch.baseIns, memb′, hurs, sim, cNum >
22:     if memb′ = smp.memb then
23:       output ← 1
24:     else
25:       output ← 0
26:     end if
27:     item ← < input, output >
28:     trainingSet.add(item)
29:   end for
30:   return trainingSet
31: end function
```

On the right side is a normalization layer that normalizes *hurs*, *sim*, and *cNum*. *hurs* indicates which heuristic (H_1 or H_3) makes the prediction. *sim* is the lexical similarity between *memb* and *lName*. *cNum* is the number of candidates in the initial candidate set generated according to Java syntax (as introduced in Section 3.2). The three numerical values are concatenated into a three-dimensional vector and fed into the Dense layer which converts them into a D-dimensional vector. The normalization layer used here is a learned layer-normalization. We normalize these data because some of them (e.g., *cNum*) are usually much larger than others (e.g., *sim*). The output of the LSTM layer and the normalization layer is merged by concatenation and fed into dense layers whose output is either one (suggesting that the prediction is safe) or zero (suggesting that the prediction is risky).

The neural network based filter could be trained in advance with examples from open-source applications. Algorithm 3 presents the training process. The training process consists of three steps, i.e., extracting sample assignments from sample projects (Lines 1-5), generating the training set (Line 8), and training filter with the training set (Line 10).

On the first step, we extract sample assignments from corpus (Line 4) in the same way as we did in Section 3.3. Based on the resulting samples (noted as *smpSet*), we generate a training set (noted as *traingSet*) on Line 8. The generation process is explained as follows. For each sample *smp*, we employ the heuristics (H_1, H_2 and H_3) to make prediction for the incomplete assignment (*smp.icpAsgn*) on Lines 17-18. The output of the prediction includes the predicted member access (notated as $memb'$), the number of initial candidates (noted as *cNum*), the heuristic that makes the prediction (noted as *hurs*), and the lexical similarity between $memb'$ and *smp.icpAsgn.lName* (noted as *sim*). With the output and the constructed sketch *skch* (Line 19), we construct an input (Lines 20-21) for the filter as

$$\begin{aligned} input = &< lType, lName, baseIns, memb', \\ & hurs, sim, cNum > . \end{aligned}$$

If the predicted member access ($memb'$) is exactly the same as that in sample (*smp.memb*), i.e., $memb' = smp.memb$, the expected output of the network (notated as *output*) is one (Lines 22-23). Otherwise, *output* is zero (Lines 24-25). The resulting training item $item =< input, output >$ is added to the training set that is in turn used to train the neural network (Lines 27-28). If the prediction fails, i.e., no member access is recommended, we ignore the sample *smp*. No training items are generated based on this sample.

4 Evaluation

In this section, we evaluate HeeNAMA on open-source applications.

4.1 Research Questions

The evaluation investigates the following research questions:

- *RQ1*: How often do project-specific API member accesses appear in the right hand side of assignments? How often are they stacked or unstacked?
- *RQ2*: How often are project-specific API members in the right hand side of assignments type compatible with the left hand side?

- *RQ3*: Does HeeNAMA outperform the state-of-the-art approach or the state-of-the-practice tool? If yes, to what extent?
- *RQ4*: How do the heuristics and the LSTM based filter influence the performance of HeeNAMA?
- *RQ5*: Can HeeNAMA be extended to recommend project-specific API member accesses nested in method invocations?
- *RQ6*: How well do API-specific approaches work in suggesting project-specific API member accesses if they are trained on within-project code?
- *RQ7*: How well does HeeNAMA work if trained with all API member accesses (considering both project-specific and public ones)?
- *RQ8*: How well does HeeNAMA perform on recently created applications?

HeeNAMA is based on the assumption that there are a large number of project-specific API member accesses on the right hand side of assignments (called member accesses on RHS for short). If the assumption does not hold, HeeNAMA will not be employed frequently, and thus it may be useless. Answering the research question RQ1 helps to validate the assumption. Notably member accesses are further divided into *stacked* and *unstacked*. For example, member access c in assignment *x=a.b.c* is unstacked whereas b is stacked because b is followed immediately by another member access. Although HeeNAMA makes predictions for both stacked and unstacked member accesses, it is more challenging to predict stacked ones because the type based reduction of candidate sets (as introduced in Section 3.4) may not work for stacked ones. For example, while predicting b in assignment *x=a.b.c*, HeeNAMA filters candidates based on the assumption that the member access to be predicted (b in the example) is type compatible with the left hand side of the assignment (x in the example). However, it is not necessarily true for stacked member accesses: assignment *x=a.b.c* requires c (instead of b) to be type compatible with x. Investigating how often member accesses are stacked or unstacked may help to reveal how often the assumption taken by HeeNAMA holds.

H_2 in Section 3.4 is based on the assumption that the next project-specific API member access in the right hand side of assignment is often type compatible with the left-hand side expression (type compatibility assumption). Answering the research question RQ2 helps to validate the assumption.

RQ3 concerns the performance of HeeNAMA against the state-of-the-art approach and the state-of-the-practice tool. To answer RQ3, we compare HeeNAMA against SLP-Core and Eclipse. SLP-Core is the implementation of the state-of-the-art approach proposed by Hellendoorn *et al.* [20]. Eclipse is a well-known and widely used IDE. SLP-Core and Eclipse are selected for comparison because of the following reasons. First, SLP-Core is the state-of-the-art approach whereas Eclipse is the state-of-the-practice IDE. Second, both SLP-Core and Eclipse can predict project-specific API member accesses. Third, both SLP-Core and Eclipse are publicly available online, which facilitates readers to repeat the evaluation. Although some advanced approaches are reported highly accurate [18], [19], [46], [47], they are not selected for comparison because we fail to get their replication packages or the packages could not be easily adapted to Java. Both SLP-Core and Eclipse are generic, and they can predict all kinds of tokens or elements whereas HeeNAMA is confined to project-specific API member accesses on RHS. The purpose of the comparison is to investigate whether HeeNAMA can improve the performance of code completion by focusing on special cases and by taking specific and fine grained context of such cases.

As specified in Section 3, HeeNAMA is composed of a sequence of heuristics and a neural network based filter. Answering research question RQ4 helps to reveal how such heuristics and filter influence the performance (e.g., precision and recall) of HeeNAMA. We also conducted an experiment to explore how different learning algorithms influence the performance of the filter and HeeNAMA. The results can be found in the online appendix.[1] Answering the research question RQ4 also helps to explain why HeeNAMA works (or not works).

As specified in Section 1, HeeNAMA focuses on a specific but common case of code completion: suggesting the following member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. Notably, there is a similar case where HeeNAMA could be applied: suggesting the following member access when a project-specific API instance is nested in a method invocation, e.g., suggesting the member b in the example of *m(a.b)*. We call such project-specific API member accesses nested in method invocations as nested member accesses for short. RQ5 investigates the possibility of extending HeeNAMA to recommend nested member accesses. Answering the research question RQ5 helps to validate the practical usefulness of HeeNAMA.

Research question RQ6 concerns the performance of HeeNAMA against API-specific approaches that are trained on within-project code. API-specific approaches are often highly accurate in recommending API member accesses because they can discover frequent patterns in training corpus. If we simply train API-specific prediction models on within-project code, however, they might discover project-specific patterns as well, and thus the resulting models might suggest project-specific member accesses as HeeNAMA does. Assuming that a model is requested to recommend a member of class C in method M, if the model is continually updated with the code in the project, it would have the knowledge of all the code in the project with the exception of method M. Thus, if there are usage patterns of class C in the project, the model would be able to make a good recommendation. That is the rationale for the investigation of RQ6. To answer RQ6, we compare HeeNAMA against an API-specific approach CSCC [35] (trained on within-project code) in suggesting project-specific API member accesses. CSCC [35] is the latest pattern mining based API-specific approach. In the evaluation, CSCC is incrementally trained with the code in the test project.

Research question RQ7 concerns the performance of HeeNAMA when trained with all project-specific and public API member accesses. Although HeeNAMA is proposed for prediction of project-specific API member access, it can be applied to train on public API member access as well. To

1. https://github.com/CC-CG/HeeNAMA/tree/master/appendix

answer RQ7, we train HeeNAMA with all API member accesses on RHS in subject applications and then evaluate its performance on project-specific and public API member accesses on RHS separately.

Research question RQ8 concerns the performance of HeeNAMA on recently created applications. To answer RQ8, we collect a new dataset of nine open-source Java applications which are created in recent years (i.e., since January 1, 2015). With the new dataset, we evaluate the performance of HeeNAMA against SLP-Core, Eclipse and CSCC again. The comparison helps to reveal the impact of replacing evaluation applications on the performance of HeeNAMA.

4.2 Setup

4.2.1 Subject Applications

We conduct the evaluation on nine open-source applications as shown in Table 1. We select such applications because they have been employed to evaluate code completion approaches successfully [7], [18], [23], [24]. An overview of the subject applications is presented in Table 1. They cover various domains such as software build, database management, and search engine. The size (LOC) of subject applications varies from 91,760 to 1,591,582. In all nine applications, there are 861,618 API member accesses in total, and 521,752 of them are project-specific API member accesses. In 521,752 project-specific API member accesses, 103,695 members are accessed on the right hand side of assignments. Notably, our evaluation is only conducted on project-specific API member accesses on the right hand side of assignments, i.e., evaluated approaches are requested for code completion on the 103,695 member accesses on RHS in subject applications.

4.2.2 Process

On the nine open-source applications presented in Table 1, we carry out a *k*-fold (*k* = 9) cross-validation. On each fold, a single application is used as testing data set (noted as *testSet*) whereas the others (eight applications) are used as training data (noted as *trainingSet*). Each of the subject applications is used as testing data set for once.

Each fold of the evaluation follows the following process:

1) SLP-Core and the filter in HeeNAMA are trained with *trainingSet* independently.
2) For each project-specific API member access on RHS in the *testSet*, we remove source code after the dot of member access (including the member) in the enclosing file. The resulting incomplete assignment is used as a query to HeeNAMA, SLP-Core and Eclipse. This step simulates the scenarios where source code in each file is typed in from the top to the bottom.
3) For each query, each code completion system is asked to return a prediction of the missing member access. A prediction is correct if and only if the predicted member access is exactly the same as that in the original source code. After prediction, the original member access is used to train SLP-Core and the first heuristic in HeeNAMA in a incremental way.
4) Based on such predictions, we calculate the performance (precision and recall) for these code completion approaches.

4.2.3 Configuration

We empirically set the embedding dimension (100) for *word2vec* and the dimension (10) for the intermediate Dense layers. We also empirically set the activation function (*ELU* [48]) for dense layers, and their optimizer (*Nadam* [49]). Other settings of the evaluation could be found in the implementation of HeeNAMA that is publicly available at https://github.com/CC-CG/HeeNAMA [50]. For comparison with SLP-Core, we employ the nested cache n-gram model in the dynamic setting which is reported best-in-class [20]. During evaluation, SLP-Core is incrementally trained with each member access in the test project once the member access has been recommended, i.e., we employ the dynamic setting as in [20]. For a given completion, those member accesses that have been recommended before are left in their usual place and thus can be learned by the nested cache n-gram model of SLP-Core. Concerning Eclipse, for each member access in the test project, we programmatically invoke the default code recommender in Eclipse (version 4.5).

4.2.4 Metrics

To answer research questions RQ3 to RQ8, we calculate the precision of top k recommendation for various approaches in recommending member accesses as follows:

$$Precision@k = \frac{N_{accepted}@k}{N_{recommended}}, \tag{2}$$

where $N_{accepted}@k$ is the number of the cases where one of items within the top k recommendation list is accepted, and $N_{recommended}$ is the number of cases the evaluated approach tries. The recall of top k recommendation is calculated as follows:

$$Recall@k = \frac{N_{accepted}@k}{N_{tested}}, \tag{3}$$

where N_{tested} is the number of tested member accesses. To answer RQ3 and RQ6, the value of k is set to 1, 3 and 5. While for the other research questions, we only present the precision and recall of top 1 recommendation. We also compute the F-measure to summarize the precision and recall values of top 1 recommendation as follows:

$$F_\beta = (\beta^2 + 1) \cdot \frac{Precision@1 \cdot Recall@1}{\beta^2 \cdot Precision@1 + Recall@1}, \tag{4}$$

where $\beta \in \mathbb{R}$ is a harmonic coefficient.In this paper, we set β to 0.5, 1 and 2 to get evaluating metrics $F_{0.5}$-measure, F_1-measure and F_2-measure, respectively [51]. F_1-measure integrates precision and recall values by the same weight. F_2-measure assigns a larger weight to the precision value whereas $F_{0.5}$-measure assigns a larger weight to the recall value. That is to say, F_2-measure focuses more on the improvement of the recall value whereas $F_{0.5}$-measure focuses more on the improvement of the precision value. The three metrics evaluate the integrated performance of the approach from different aspects.

TABLE 2
Popularity of Project-Specific API Member Accesses on RHS

Applications	All Accesses (N_{all})	Accesses on RHS (N_{RHS})	$\frac{N_{RHS}}{N_{all}}$	Unstacked Accesses on RHS ($N_{unstacked}$)	$\frac{N_{unstacked}}{N_{RHS}}$	$\frac{N_{unstacked}}{N_{all}}$
Ant	23,702	3,400	14.34%	3,064	90.12%	12.93%
Batik	24,031	5,158	21.46%	4,773	92.54%	19.86%
Cassandra	100,575	17,724	17.62%	13,786	77.78%	13.71%
Log4J	31,927	7,137	22.35%	4,963	69.54%	15.54%
Lucene-solr	266,040	56,623	21.28%	47,373	83.66%	17.81%
Maven2	10,070	1,411	14.01%	1,209	85.68%	12.01%
Maven3	18,067	2,351	13.01%	2,037	86.64%	11.27%
Xalan-J	22,621	4,154	18.36%	3,946	94.99%	17.44%
Xerces	24,719	5,737	23.21%	5,484	95.59%	22.19%
Total	**521,752**	**103,695**	**19.87%**	**86,635**	**83.55%**	**16.60%**

4.3 Results and Analysis

4.3.1 RQ1: Project-Specific API Member Access on RHS

To address RQ1, we count the number of member accesses in subject applications as well as those on the right-hand side of assignments (RHS). We also count the number of unstacked member accesses on the right-hand sides of assignments. The results are presented in Table 2. The first column presents the names of subject applications. The second column presents the number of member accesses in subject applications. The third column and the forth column present the number of member accesses on RHS and the ratio of member accesses on RHS to all member accesses, respectively. The number of unstacked member accesses on RHS is presented in the fifth column. The ratio of them to member accesses on RHS and the ratio of them to all member accesses are presented in the last two columns.

From this table we make the following observations:

- First, the number of member accesses on RHS is quite large. For the nine subject applications, the total number is as much as 103,695. Consequently, highly accurate prediction approaches (if there is any) for such member accesses would be employed frequently, and thus could be beneficial for developers.
- Second, member accesses on RHS account for a significant proportion of member accesses in the source code. On average, they account for 19.87%=103,695/521,752 of the member accesses.
- Third, most (83.55 percent) of member accesses on RHS are unstacked, i.e., where HeeNAMA actually works. The total number of unstacked member accesses is as much as 86,635 for nine subject applications. They account for 16.60%=86,635/521,752 of all member accesses.

From the analysis in the preceding paragraphs, we conclude that there is a large number of member accesses on the right-hand side of assignments. Consequently, code completion approaches/tools confined to such member accesses could be useful as long as they are accurate.

4.3.2 RQ2: Type Compatibility of Project-Specific API Member Accesses

To address RQ2, we count the number of member accesses on RHS and the number of those which are type compatible with the left hand side expression. We present the results in Table 3. The first column shows the names of subject applications. The second column presents the number of member accesses on RHS. The third column presents the number of member accesses on RHS which are type compatible with the left hand side expression. The ratio of type compatible member accesses to those on RHS is presented in the last column.

From Table 3, we observe that the ratio is high. As shown in the table, the ratio varies from 70.42 to 92.27 percent. On average, 81.82%=84,842/103,695 of member accesses on RHS are type compatible with the left hand side.

From the analysis in the preceding paragraph, we conclude that in most cases the type compatibility assumption is correct.

4.3.3 RQ3: Comparison Against Existing Approaches

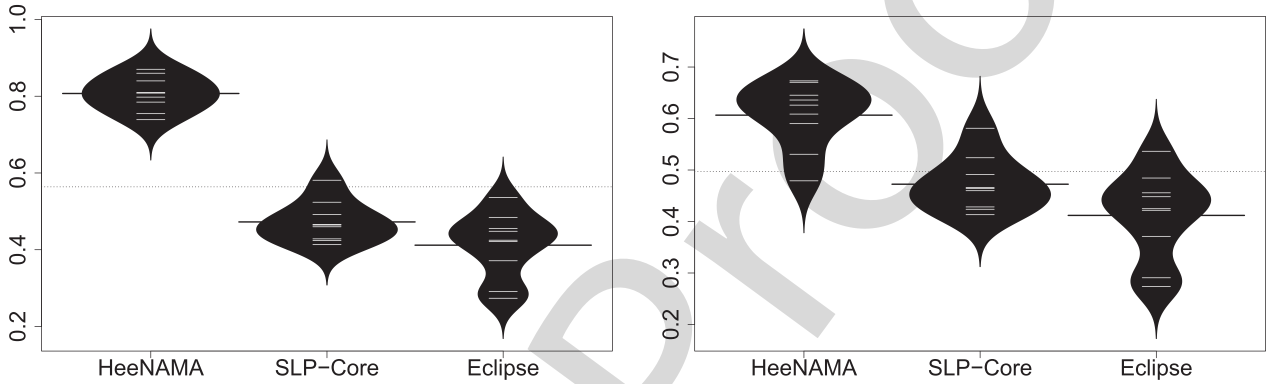
To address RQ3, we compare HeeNAMA against SLP-Core [20] and Eclipse IDE on nine open-source applications. The evaluation results are presented in Table 4 and Fig. 3. In the table, the first column presents the names of subject applications. The second to seventh columns present the precision and recall of HeeNAMA in the top k recommendation list, respectively. The precision of SLP-Core and Eclipse at top k is presented in the last six columns, respectively. Different from HeeNAMA, both SLP-Core and Eclipse always make recommendation whenever they are requested, i.e., the number of cases they try is equal to the number of tested member accesses. Consequently, for SLP-Core and Eclipse, recall is always equal to precision, and thus it is omitted from the table. Fig. 3 presents evaluation results at the top 1 recommendation with bean-plot. Each bean in Fig. 3 presents the resulting precision (sub-graph on the left) or recall (sub-graph on the right) of an evaluated approach on subjection

TABLE 3
Type Compatibility of Project-Specific API Member Accesses

Applications	Accesses on RHS (N_{RHS})	Type Compatible Accesses (N_{TC})	$\frac{N_{TC}}{N_{RHS}}$
Ant	3,400	3,042	89.47%
Batik	5,158	4,567	88.54%
Cassandra	17,724	13,357	75.36%
Log4J	7,137	5,026	70.42%
Lucene-solr	56,623	46,554	82.22%
Maven2	1,411	1,204	85.33%
Maven3	2,351	2,045	86.98%
Xalan-J	4,154	3,833	92.27%
Xerces	5,737	5,214	90.88%
Total	**103,695**	**84,842**	**81.82%**

TABLE 4
Comparison Against Existing Approaches

Applications	HeeNAMA						SLP-Core			Eclipse		
	Precision			Recall			Precision			Precision		
	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5
Ant	80.82%	85.17%	85.92%	63.59%	69.91%	71.44%	42.38%	51.12%	54.65%	48.44%	67.85%	78.15%
Batik	75.46%	83.53%	85.90%	60.86%	68.94%	71.56%	42.83%	51.01%	53.06%	42.48%	67.78%	76.93%
Cassandra	81.00%	84.96%	84.63%	59.00%	62.97%	63.48%	41.32%	48.10%	51.30%	44.81%	68.60%	79.87%
Log4J	84.00%	87.32%	88.02%	67.31%	79.04%	79.99%	49.15%	54.32%	56.79%	37.12%	59.41%	65.28%
Lucene-solr	86.00%	88.05%	87.94%	62.60%	67.05%	67.94%	52.38%	60.63%	63.61%	53.64%	75.64%	83.23%
Maven2	87.03%	91.92%	92.51%	67.04%	74.20%	75.27%	58.11%	65.27%	67.90%	45.57%	72.43%	82.99%
Maven3	73.89%	85.12%	85.46%	64.53%	72.27%	73.50%	45.98%	53.00%	57.38%	27.35%	64.99%	73.63%
Xalan-J	79.77%	79.57%	80.02%	53.06%	66.18%	67.96%	46.44%	53.71%	56.69%	42.18%	67.21%	75.83%
Xerces	78.47%	80.72%	80.69%	47.90%	59.32%	60.31%	46.59%	59.28%	62.40%	29.07%	47.25%	59.12%
Average	**83.36%**	**86.39%**	**86.51%**	**61.16%**	**67.12%**	**68.11%**	**48.84%**	**56.80%**	**59.79%**	**47.74%**	**70.48%**	**79.09%**



(a) Precision at Top 1

(b) Recall at Top 1

Fig. 3. Comparison against existing approaches.

applications. The white small lines represent the precision or recall on a single subject application, and the shape of the beans represents the distribution of the performance. The black lines crossing beans represent the average precision (or recall) of the evaluated approaches.

From Table 4 and Fig. 3, we make the following observations:

- First, HeeNAMA is precise. The precision at top 1 recommendation varies from 73.89 to 87.03 percent, and the average precision is as much as 83.36 percent. In other words, in most cases (more than 83 percent) the approach suggests the member access exactly the same as developers want.
- Second, HeeNAMA is significantly more precise than SLP-Core and Eclipse. On each of the subject applications, the precision of HeeNAMA at top k is always greater than that of SLP-Core and Eclipse. We also compare their precision at top 1 in Fig. 3a where the distance between different approaches is obvious. On average, HeeNAMA improves precision at top 1 significantly by 70.68%=(83.36%-48.84%)/48.84%.
- Third, HeeNAMA improves recall at top 1 recommendation significantly. Although its recall varies dramatically from 47.90 to 67.31 percent, it is always greater than that of SLP-Core and Eclipse. The bean plot in Fig. 3b visually illustrates the distance among such approaches. On average, it improves recall at top 1 by 25.23%=(61.16%-48.84%)/48.84%.

Notably, SLP-Core and Eclipse work well on challenging project-specific API member accesses, achieving a precision/recall of 48.84 and 47.74 percent, respectively. One possible reason for the success of SLP-Core is that it leverages examples of member accesses in the test project (i.e., within-project code) because its nested cache n-gram model is continually updated while recommendation [20]. Eclipse succeeds frequently because it recommends member accesses according to type information of the left hand side which is also leveraged by HeeNAMA and thus it makes some correct recommendations.

HeeNAMA refuses to make recommendations when it lacks of confidence. We present the frequency of HeeNAMA refusing to make recommendations in Table 5. From this table, we observe that on more than a quarter (26.62 percent) cases HeeNAMA refuses to make recommendations. Comparing Table 4 against Table 5, we observe that the recall is influenced by the frequency of refusal. The results are reasonable in that if HeeNAMA makes fewer recommendations, it has smaller chance to make correct recommendations (and thus lower recall).

As suggested by Table 2, project-specific API member accesses could be further divided into stacked and unstacked ones. To this end, we further investigate how well HeeNAMA works at top 1 recommendation on such subsets. On

TABLE 5
Frequency of HeeNAMA Refusing to Make Recommendations

Applications	Accesses on RHS (N_{RHS})	Refused Accesses (N_{Ref})	$\frac{N_{Ref}}{N_{RHS}}$
Ant	3,400	725	21.32%
Batik	5,158	998	19.35%
Cassandra	17,724	4,814	27.16%
Log4J	7,137	1,418	19.87%
Lucene-solr	56,623	15,405	27.21%
Maven2	1,411	324	22.96%
Maven3	2,351	298	12.68%
Xalan-J	4,154	1,391	33.49%
Xerces	5,737	2,235	38.96%
Total	**103,695**	**27,608**	**26.62%**

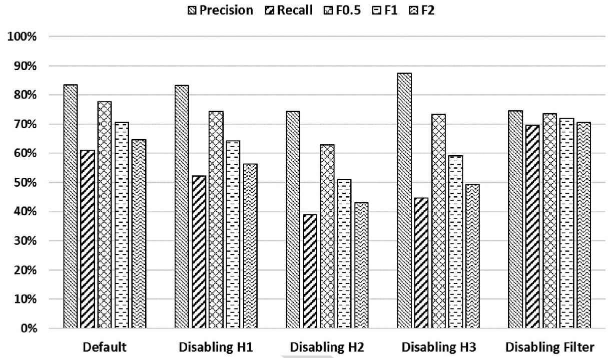


Fig. 4. Impacts of heuristics and filter.

the stacked ones, HeeNAMA achieves a precision of 74.86 percent and a recall of 45.84 percent whereas the precision (and recall) of SLP-Core and Eclipse IDE is 48.12 and 7.11 percent, respectively. On the unstacked ones, HeeNAMA achieves a precision of 84.71 percent and a recall of 64.18 percent whereas the precision (and recall) of SLP-Core and Eclipse IDE is 48.98 and 55.75 percent, respectively. The results suggest that HeeNAMA works much better on unstacked ones than on stacked ones. Stacking does not affect SLP-core because SLP-core is completely based on token sequences captured by n-gram models. However, stacking does affect Eclipse negatively because Eclipse leverages the type information of the left hand side to make recommendations and stacking makes such type information useless (or even misleading). However, the results also suggest that even on the stacked ones, HeeNAMA outperforms SLP-Core and Eclipse IDE as well.

From the analysis in the preceding paragraphs, we conclude that HeeNAMA is precise, and it significantly outperforms both the state-of-the-art approach and the state-of-the-practice tool in suggesting the next member access for assignments.

4.3.4 RQ4: Impacts of Heuristics and Filter

As introduced in Section 3, HeeNAMA is composed of three heuristics and neural network based filter. The evaluation in the preceding sections suggests that HeeNAMA as a whole is accurate. To investigate how the heuristics and filter influence the performance of HeeNAMA, we repeat the evaluation (including both the training and testing phase) for four times. On the first three times, we disable the heuristics (i.e., H_1, H_2, H_3), respectively. Finally, we disable the neural network based filter, and repeat the evaluation for the last time. For example, when disabling H_1, the filter is presented with only type-compatible member accesses that are ranked first by H_3 according to lexical similarity.

The evaluation results are presented in Fig. 4 and Table 6. From Fig. 4 and Table 6, we make the following observations:

- First, disabling any of the three heuristics leads to significant reduction in recall. The reduction is as much as 14.60%=(61.16%-52.23%)/61.16%, 36.35%=(61.16%-38.93%)/61.16%, and 27.11%=(61.16%-44.58%)/61.16%, respectively. The evaluation results suggest that all of the heuristics are critical for HeeNAMA to achieve high recall.
- Second, disabling neural network based filter improves recall at the cost of reduced precision. Precision is reduced by 10.64%=(83.36%-74.49%)/83.36% whereas recall is improved by 13.73%=(69.56%-61.16%)/61.16%. Although disabling the filter results in more balanced precision and recall, the filter is beneficial because of the following reasons. First, although the filter reduces F_2 (that biases for recall) from 70 to 65 percent, F_1 keeps stable and $F_{0.5}$ (that biases for precision) increases from 73 to 78 percent. Second, for code completion, high precision (and thus few uncorrect recommendations) is critical because incorrect recommendations are often misleading and even worse than no recommendation at all. Code complete tools that frequently make incorrect recommendations would lose trust of developers, and will be finally discarded by developers. To this end, we introduce the filter to improve precision (at the cost of moderate reduction in recall) by removing risky recommendations. The evaluation results suggest that the filter works as expected.
- Third, disabling H_1 and H_3 has little influence on the precision of HeeNAMA. One possible reason is that the neural network based filter (working at the final phase of HeeNAMA) can filter out most of the risky predictions, and thus guarantees the final precision.

We also present in Table 7 the frequency of HeeNAMA refusing to make recommendations when one of the heuristics or the filter is disabled. From the table, we observe that disabling any of the three heuristics improves the frequency of refusal while disabling the filter reduces the frequency, which is consistent with the observation from Table 6. When disabling heuristics, HeeNAMA refuses to make recommendations more frequently and thus it achieves lower recall. However, when disabling the filter, the frequency of HeeNAMA refusing to make recommendations reduces greatly so its recall improves.

We investigate the impact of the features leveraged by the filter and present the evaluation results in Table 8. From the table, we conclude that each of the features is useful. We also investigate the impact of reducing one or two hidden layers used by the filter. The results suggest that reducing one or two hidden layers would result in reduction (1.56 and 2.99 percent, respectively) in F_1.

To further investigate the contribution of heuristics, we evaluate the performance of pair-wise disabling (i.e., activating each heuristic on subject applications). The evaluation

TABLE 6
Impacts of Heuristics and Filter

Applications	Default		Disabling H_1		Disabling H_2		Disabling H_3		Disabling Filter	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Ant	80.82%	63.59%	82.75%	48.12%	77.53%	44.15%	89.00%	35.21%	71.27%	67.79%
Batik	75.46%	60.86%	77.34%	48.10%	61.04%	35.21%	85.27%	38.95%	68.59%	64.99%
Cassandra	81.00%	59.00%	80.28%	51.84%	81.05%	39.33%	85.61%	39.35%	73.07%	66.82%
Log4J	84.00%	67.31%	84.64%	60.08%	69.57%	42.19%	87.82%	53.13%	78.97%	73.28%
Lucene-solr	86.00%	62.60%	85.63%	53.46%	73.84%	38.81%	89.12%	48.65%	77.22%	71.79%
Maven2	87.03%	67.04%	87.49%	53.01%	90.09%	60.60%	91.02%	33.03%	77.87%	72.08%
Maven3	73.89%	64.53%	76.13%	50.32%	85.66%	58.70%	78.77%	31.09%	71.58%	67.59%
Xalan-J	79.77%	53.06%	80.98%	44.37%	65.52%	32.62%	86.21%	36.11%	63.89%	62.52%
Xerces	78.47%	47.90%	73.91%	44.00%	72.04%	26.18%	77.21%	35.14%	62.82%	61.74%
Average	**83.36%**	**61.16%**	**83.14%**	**52.23%**	**74.32%**	**38.93%**	**87.45%**	**44.58%**	**74.49%**	**69.56%**

TABLE 7
Frequency of HeeNAMA Refusing to Make Recommendations

Applications	Accesses on RHS	Default	Disabling H_1	Disabling H_2	Disabling H_3	Disabling Filter
Ant	3,400	21.32%	41.85%	43.05%	60.44%	4.88%
Batik	5,158	19.35%	37.81%	42.32%	54.32%	5.25%
Cassandra	17,724	27.16%	35.43%	51.47%	54.04%	8.55%
Log4J	7,137	19.87%	29.02%	39.36%	39.50%	7.21%
Lucene-solr	56,623	27.21%	37.57%	47.44%	45.41%	7.03%
Maven2	1,411	22.96%	39.41%	32.73%	63.71%	7.44%
Maven3	2,351	12.68%	33.90%	31.47%	60.53%	5.57%
Xalan-J	4,154	33.49%	45.21%	50.21%	58.11%	2.14%
Xerces	5,737	38.96%	40.47%	63.66%	54.49%	1.72%
Total	**103,695**	**26.62%**	**37.18%**	**47.62%**	**49.02%**	**6.62%**

results are presented in Table 9. The table does not present the option of activating the filter alone because the filter cannot work without the candidate items generated by heuristics. From the table, we observe that single activation results in significant reduction in performance. For example, activating H_1 only increases precision slightly by 1.36%= (84.51%-83.36%)/83.36% but reduces recall significantly by 23.09%=(61.16%-47.04%)/61.16%. Single activation for H_2 or H_3 significantly reduces both precision and recall.

We conclude from the preceding analysis that the proposed heuristics and the filter are useful.

4.3.5 RQ5: Performance on Nested Member Accesses

To address RQ5, we evaluate HeeNAMA on nested member accesses from nine open-source applications. We also compare the performance of HeeNAMA against SLP-Core and Eclipse IDE. The evaluation results are presented in Table 10. In the table, the first column presents the names of subject applications. The second column presents the number of member accesses in subject applications. The third column and the forth column present the number of nested member accesses and the ratio of nested member accesses to all member accesses, respectively. The fifth column and the sixth column present the precision and recall of HeeNAMA, respectively. The precision of SLP-Core and Eclipse is presented in the last two columns. For SLP-Core and Eclipse, recall is always equal to precision, and thus it is omitted from the table.

From Table 10, we make the following observations:

- First, nested member accesses are popular. On average, nested member accesses account for a significant proportion (25.89 percent) of all member accesses. Consequently, applying HeeNAMA to such member accesses would make it more general and therefore also much stronger.
- Second, HeeNAMA is precise in suggesting nested member accesses. It achieves a precision of 71.64 percent, which is much higher than SLP-Core and Eclipse.
- Third, the recall of HeeNAMA is higher than that of Eclipse but lower than that of SLP-Core. One possible reason is that the filter of HeeNAMA improves precision at the cost of reduced recall.

From the analysis in the preceding paragraph, we conclude that HeeNAMA can be extended to recommend nested member accesses, which improves the practical usefulness of HeeNAMA.

TABLE 8
Impact of the Features Leveraged by the Filter

Settings	Performance of HeeNAMA	
	Precision	Recall
Default	83.36%	61.16%
Disabling $hurs$	75.16%	59.06%
Disabling sim	73.34%	56.05%
Disabling $cNum$	80.95%	60.86%
Disabling all	69.97%	51.28%

TABLE 9
Impacts of Heuristics

Applications	Default		Activating H_1		Activating H_2		Activating H_3	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Ant	80.82%	63.59%	87.29%	35.97%	42.96%	41.18%	28.48%	28.35%
Batik	75.46%	60.86%	82.48%	39.90%	40.38%	38.52%	25.97%	25.71%
Cassandra	81.00%	59.00%	83.79%	42.60%	45.93%	41.38%	29.25%	29.14%
Log4J	84.00%	67.31%	85.76%	55.11%	51.48%	46.11%	27.76%	27.73%
Lucene-solr	86.00%	62.60%	87.52%	50.67%	52.83%	47.49%	29.56%	29.39%
Maven2	87.03%	67.04%	76.63%	35.08%	53.02%	46.70%	39.65%	38.98%
Maven3	73.89%	64.53%	69.01%	33.35%	56.64%	52.40%	40.82%	40.49%
Xalan-J	79.77%	53.06%	79.68%	38.81%	35.56%	34.95%	26.33%	26.29%
Xerces	78.47%	47.90%	66.86%	42.41%	26.48%	25.83%	32.32%	32.30%
Average	**83.36%**	**61.16%**	**84.51%**	**47.04%**	**48.37%**	**44.10%**	**29.58%**	**29.44%**

TABLE 10
Performance on Nested Member Accesses

Applications	All Accesses (N_{all})	Nested Accesses (N_{nested})	$\frac{N_{nested}}{N_{all}}$	HeeNAMA		SLP-Core	Eclipse
				Precision	Recall	Precision	Precision
Ant	23,702	5,329	22.48%	75.89%	32.01%	48.13%	19.23%
Batik	24,031	3,997	16.63%	70.15%	17.99%	28.72%	11.88%
Cassandra	100,575	29,143	28.98%	75.93%	36.37%	43.50%	33.17%
Log4J	31,927	9,626	30.15%	70.59%	33.01%	46.88%	17.52%
Lucene-solr	266,040	69,924	26.28%	69.38%	43.26%	51.47%	29.96%
Maven2	10,070	3,045	30.24%	75.05%	40.69%	51.17%	23.84%
Maven3	18,067	5,912	32.72%	76.13%	46.33%	49.81%	14.09%
Xalan-J	22,621	3,685	16.29%	75.22%	21.17%	40.87%	15.25%
Xerces	24,719	4,440	17.96%	76.17%	36.64%	44.21%	29.73%
Total	**521,752**	**135,101**	**25.89%**	**71.64%**	**39.11%**	**48.01%**	**27.57%**

4.3.6 RQ6: Comparison Against API-Specific Approaches Trained on Within-Project Code

To address RQ6, we compare HeeNAMA against CSCC on subject applications. Notably, we incrementally train CSCC with within-project member accesses in the evaluation, i.e., trained with member accesses that have been recommended before the current one in the test project. The evaluation results are presented in Table 11. In the table, the first column presents the names of subject applications. The second to seventh columns present the precision and recall of HeeNAMA at the top k recommendation, respectively. The last six columns present the precision and recall of CSCC at the top k recommendation, respectively.

From the table, we make the following observations:

- First, CSCC works well on project-specific API member accesses. It achieves a high precision of 64.40 percent at top 1, 77.01 percent at top 3 and 79.48 percent at top 5 recommendation.
- Second, HeeNAMA is significantly more precise than CSCC in predicting project-specific API member accesses. On each application, its precision at top k is always higher than the precision of CSCC. For example, at top 1 recommendation, it improves precision by 29.44%=(83.36%-64.40%)/64.40%.
- Third, HeeNAMA achieves higher recall than CSCC. The recall at top 1, 3 and 5 recommendation is improved by 20.46%=(61.16%-50.77%)/50.77%, 10.58%=(67.12%-60.70%)/60.70% and 8.72%=(68.11%-62.65%)/62.65%, respectively.

From the analysis in the preceding paragraph, we conclude that HeeNAMA significantly outperforms API-specific approaches in suggesting project-specific API member accesses even if they are trained on within-project code.

4.3.7 RQ7: Performance of HeeNAMA When Trained With All API Member Accesses

To answer RQ7, we train HeeNAMA with all API member accesses and evaluate it on project-specific and non-project-specific (i.e., public API) member accesses separately. On project-specific member accesses, the precision and recall of HeeNAMA are 81.28 and 64.85 percent, respectively. On non-project-specific member accesses, the precision and recall of HeeNAMA are 86.00 and 57.67 percent, respectively. The performance of HeeNAMA on non-project-specific member accesses is comparable to that on project-specific accesses because HeeNAMA takes non-project-specific member accesses as project-specific ones in fact. The performance on non-project-specific member accesses is not significantly higher than that on project-specific accesses

TABLE 11
Comparison Against API-Specific Approach

Applications	HeeNAMA						CSCC					
	Precision			Recall			Precision			Recall		
	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5
Ant	80.82%	85.17%	85.92%	63.59%	69.91%	71.44%	57.20%	67.20%	70.58%	39.85%	46.82%	49.18%
Batik	75.46%	83.53%	85.90%	60.86%	68.94%	71.56%	61.44%	71.76%	73.59%	42.26%	49.36%	50.62%
Cassandra	81.00%	84.96%	84.63%	59.00%	62.97%	63.48%	58.06%	70.09%	72.08%	42.19%	50.93%	52.38%
Log4J	84.00%	87.32%	88.02%	67.31%	79.04%	79.99%	69.27%	81.38%	84.32%	55.51%	65.22%	67.58%
Lucene-solr	86.00%	88.05%	87.94%	62.60%	67.05%	67.94%	68.13%	81.47%	84.04%	57.40%	68.63%	70.80%
Maven2	87.03%	91.92%	92.51%	67.04%	74.20%	75.27%	56.29%	64.33%	66.29%	38.70%	44.22%	45.57%
Maven3	73.89%	85.12%	85.46%	64.53%	72.27%	73.50%	51.29%	58.98%	60.81%	34.62%	39.81%	41.05%
Xalan-J	79.77%	79.57%	80.02%	53.06%	66.18%	67.96%	51.62%	65.34%	68.17%	39.50%	50.00%	52.17%
Xerces	78.47%	80.72%	80.69%	47.90%	59.32%	60.31%	57.24%	69.49%	71.63%	37.74%	45.81%	47.22%
Average	**83.36%**	**86.39%**	**86.51%**	**61.16%**	**67.12%**	**68.11%**	**64.40%**	**77.01%**	**79.48%**	**50.77%**	**60.70%**	**62.65%**

TABLE 12
Performance on Project-Specific Member Accesses in the New Dataset

Applications	All Accesses (N_{all})	Accesses on RHS (N_{RHS})	$\frac{N_{RHS}}{N_{all}}$	HeeNAMA		SLP-Core	Eclipse	CSCC	
				Precision	Recall	Precision	Precision	Precision	Recall
Atlas	11,466	2,062	17.98%	81.11%	65.81%	44.76%	26.72%	53.12%	35.94%
Carbondata	30,718	6,416	20.89%	81.31%	65.96%	40.10%	44.08%	58.41%	45.46%
Fluo	7,263	1,234	16.99%	86.72%	62.97%	49.92%	52.84%	70.01%	52.59%
Giraph	13,028	2,200	16.89%	82.95%	49.09%	46.68%	39.91%	61.10%	32.27%
Ignite	284,169	53,702	18.90%	83.85%	54.34%	45.83%	30.71%	57.66%	47.70%
Johnzon	4,025	765	19.01%	79.20%	46.80%	53.20%	34.90%	61.12%	51.37%
Nifi	107,159	22,751	21.23%	79.43%	60.04%	41.84%	33.42%	68.26%	44.18%
Plc4x	9,189	1,614	17.56%	84.40%	65.37%	57.43%	48.39%	65.87%	37.67%
Streams	16,370	3,268	19.96%	38.76%	36.66%	33.51%	25.46%	40.79%	33.14%
Total	**483,387**	**94,012**	**19.45%**	**80.31%**	**56.27%**	**44.36%**	**32.85%**	**59.52%**	**45.49%**

because HeeNAMA does not leverage any unique properties of non-project-specific APIs, e.g., patterns of API usage.

4.3.8 RQ8: Performance on the New Dataset

To answer RQ8, we evaluate HeeNAMA against SLP-Core, Eclipse and CSCC on nine open-source Java applications that are recently created. The evaluation results are presented in Table 12. In the table, the first column presents the names of subject applications. The second column presents the number of all project-specific member accesses in subject applications. The third column and the forth column present the number of member accesses on RHS and the ratio of them to all project-specific member accesses, respectively. The fifth column and the sixth column present the precision and recall of HeeNAMA, respectively. The precision of SLP-Core and Eclipse is presented in the seventh and eighth columns, respectively. The last two columns present the precision and recall of CSCC. For SLP-Core and Eclipse, recall is always equal to precision, and thus it is omitted from the table.

From Table 12, we make the following observations:

- First, the ratio of project-specific member accesses on RHS in the new dataset is close to that of the original dataset (19.45 versus 19.87 percent).
- Second, the performance of HeeNAMA on the new dataset is comparable to its default performance on the original dataset. The precision and recall of HeeNAMA on the new dataset are 80.31 and 56.27 percent, respectively.
- Third, HeeNAMA significantly outperforms SLP-Core, Eclipse and CSCC on the new dataset. It improves the precision by 34.93%=(80.31%-59.52%)/59.52% and the recall by 23.70%=(56.27%-45.49%)/45.49%, respectively.

4.4 Threats to Validity

A threat to the external validity is that only nine applications are involved in the evaluation and such applications may be unrepresentative. Consequently, the evaluation results may not hold if other subject applications are involved. To reduce the threat, we reuse the subject applications that have been successfully employed in related work, and such applications contain more than one hundred thousand training items that have been used to evaluate HeeNAMA. Another threat to the external validity is that HeeNAMA is only evaluated on Java applications. Conclusions on Java applications may not hold for applications written in other languages, e.g., C++.

A threat to construct validity is that the evaluation is based on the assumption: the involved assignments in the subject applications are correct. We evaluate the suggested member access against that chosen by the original developers (i.e., the member access in the downloaded source code). If they are identical, we say the prediction is correct. Otherwise, it is declared incorrect. However, it is likely that the original developers may have chosen incorrect member access (and thus caused a bug), which makes the evaluation potentially incorrect. To reduce the threat, we select mature and well-known applications for evaluation because such applications are likely to contain fewer bugs.

A threat to internal validity is that the simulated code completion scenarios may be different from real scenarios. In the evaluation, we simulate the scenarios where source code in each document is typed in from the top to the bottom. In other words, context for code completion is the source code before the current cursor (where the suggested token will be inserted). However, in real world, especially in software maintenance and bug fixing phases, there may exist source code after the current cursor, and such code could be exploited for code completion as well. Another threat to internal validity is that in the simulated scenarios Java source code files (∗.java) are created in the alphabetical order of the file names which may not be the real case. The creation order may influence the performance of code completion because it exploits existing code within the enclosing project. We take such an order because we fail to find their creation time from the web where the source code is downloaded.

5 Conclusions and Future Work

In this paper we highlight the necessity of code completion for project-specific API member access on the right hand side of assignment. We also propose an automatic and accurate approach to suggesting the next member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. The approach is accurate because it takes full advantages of the context of the code completion, including the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. It also employs a neural network to filter out risky prediction, which guarantees high precision of code completion. HeeNAMA has been evaluated on nine open-source applications. Our evaluation results suggest that compared to the state-of-the-art approach and the state-of-the-practice tool HeeNAMA improves both precision and recall significantly.

Findings presented in the paper are valuable to the research community in the following aspects:

- First, we empirically reveal that public API member accesses are less popular than project-specific API member accesses (39 versus 61 percent), which is consistent with the conclusion of a recent empirical study conducted on C# repositories [22]. Considering that most of the related work (as introduced in Section 2) focuses on APIs, this finding may help researchers identify better target scenarios for their research: focusing on project-specific API member accesses could be more fruitful and more useful.
- Second, we empirically reveal that applying API specific approaches to suggest project-specific API member accesses without significant adaption often results in inaccuracy. This finding may serve as an open call for automatic code completion approaches on project-specific APIs.
- Third, by focusing on a special case of code completion (project-specific API member accesses on the right hand side of assignments), the proposed approach significantly outperforms existing generic ones on this special case. The results may inspire future research on highly accurate code completion by focusing on special cases, like suggestion of parameters, recommendation of declarations, and completion of conditional statements.

A limitation of HeeNAMA is that it applies only to a specific but common case (accounting for 19.87 percent of project-specific API member accesses): suggesting the following member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. Although we extend HeeNAMA to recommend project-specific API member accesses nested in method invocations (accounting for 25.89 percent of project-specific API member accesses) in Section 4.3.5, there are still about 52 percent project-specific API member accesses that HeeNAMA cannot recommend. Recommendation for such cases can be more challenging since little context information could be leveraged to make predictions. For example, when the developer types a dot after a base instance at the first line of a method, we can hardly extract any useful information within the method body. However, in the future, considering all kinds of project-specific API member accesses would make HeeNAMA much stronger.

Future work is needed to evaluate HeeNAMA further. More subject applications should be involved in further evaluation. Evaluation of HeeNAMA on other programming languages such as C++ and C# should also be involved in the future work. HeeNAMA should also be evaluated in real scenarios, and get feedback from developers. The final target of code completion is to facilitate coding. Consequently, it is critical for the success of HeeNAMA that developers are willing to use it in practice. In the future, HeeNAMA could also be combined with API-specific techniques (e.g., CSCC) to support both public API and project-specific API member access recommendations.

Acknowledgment

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

References

[1] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2008, pp. 317–326. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.42

[2] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595728

[3] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the eclipse IDE?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, Jul./Aug. 2006. [Online]. Available: http://dx.doi.org/10.1109/MS.2006.105

[4] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 233–242.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[6] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 332–343.

[7] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337322

[8] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462192

[9] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est Omen: Exploring and exploiting similarities between argument and parameter names," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1063–1073. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884841

[10] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 48–61. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065018

[11] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Mining Knowl. Discov.*, vol. 15, no. 1, pp. 55–86, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10618-006-0059-1

[12] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "CSCC: Simple, efficient, context sensitive code completion," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 71–80.

[13] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with Bayesian networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 3:1–3:31, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2744200

[14] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proc. 23rd Eur. Conf. Object-Oriented Program.*, 2009, pp. 318–343. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_15

[15] A. T. Nguyen *et al.*, "Graph-based pattern-oriented, context-sensitive source code completion," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 69–79. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337232

[16] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[17] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 207–216.

[18] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[19] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[20] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 763–773. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106290

[21] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn, "CACHECA: A cache language model based code suggestion tool," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 705–708. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819143

[22] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 960–970.

[23] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[24] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2902362

[25] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *CoRR*, vol. abs/1506.00019, 2015. [Online]. Available: http://arxiv.org/abs/1506.00019

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[27] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[28] R. Karampatsis and C. A. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: http://arxiv.org/abs/1903.05734

[29] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[30] 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 *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[31] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning API usages from bytecode: A statistical approach," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 416–427. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884873

[32] L. Rabiner and B. Juang, "An introduction to hidden Markov models," *IEEE ASSp Mag.*, vol. 3, no. 1, pp. 4–16, Jan. 1986.

[33] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287630

[34] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967.

[35] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "A simple, efficient, context-sensitive approach for code completion," *J. Softw. Evol. Process*, vol. 28, no. 7, pp. 512–541, Jul. 2016. [Online]. Available: https://doi.org/10.1002/smr.1791

[36] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, 2010, pp. 156–165. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2010.27

[37] C. Caprile and P. Tonella, "Nomen est Omen: Analyzing the language of function identifiers," in *Proc. 6th Work. Conf. Reverse Eng.*, 1999, pp. 112–122.

[38] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proc. Int. Conf. Softw. Maintenance*, 2000, pp. 97–107. [Online]. Available: http://dl.acm.org/citation.cfm?id=850948.853439

[39] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *Proc. 14th IEEE Int. Conf. Program Comprehension*, 2006, pp. 3–12.

[40] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 232–242. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001448

[41] M. Pradel and T. R. Gross, "Name-based analysis of equally typed method arguments," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1127–1143, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2013.7

[42] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proc. Int. Conf. Inf. Integr. Web*, 2003, pp. 73–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104278.3104293

[43] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 362–373. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901759

[44] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[45] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 438–449. [Online]. Available: https://doi.org/10.1109/ICSE.2017.47

[46] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2016, pp. 761–774. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837671

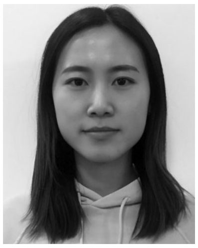
[47] P. Bielik, V. Raychev, and M. Vechev, "PHOG: Probabilistic model for code," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, 2016, pp. 2933–2942. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045390.3045699

[48] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (ELUs)," 2015, *arXiv:1511.07289*.

[49] T. Dozat, "Incorporating Nesterov momentum into adam," 2016.

[50] CC-CG, "Cc-CG/HeeNAMA v1.0," Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3559330

[51] A. Maratea, A. Petrosino, and M. Manzo, "Adjusted F-measure and kernel scaling for imbalanced data learning," *Inf. Sci.*, vol. 257, pp. 331–341, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025513003137



Lin Jiang received the BS degree from the College of Information and Electrical Engineering, China Agricultural University, Beijing, China, in 2016. She is currently working toward the PhD degree in the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, under the supervision of Dr. H. Mei. She is interested in software evolution and computer programming.

Hui Liu received the BS degree in control science from Shandong University, Jinan, China, in 2001, the MS degree in computer science from Shanghai University, Shanghai, China, in 2004, and the PhD degree in computer science from Peking University, Beijing, China, in 2008. He is a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He was a visiting research fellow with the Centre for Research on Evolution, Search and Testing (CREST), University College London, United Kingdom. He served on the program committees and organizing committees of prestigious conferences, such as ICSME and RE. He is particularly interested in software refactoring, software evolution, and software quality. He is also interested in developing practical tools to assist software engineers.

He Jiang (Member, IEEE) received the PhD degree in computer science from the University of Science and Technology of China, Hefei, China. He is currently a professor with the Dalian University of Technology and an adjunct professor with the Beijing Institute of Technology. He is also a member of the ACM and the CCF (China Computer Federation). He is one of the ten supervisors for the Outstanding Doctoral Dissertation of the CCF in 2014. His current research interests include search-based software engineering (SBSE) and mining software repositories (MSR). His work has been published at premier venues like ICSE, SANER, and GECCO, as well as in major IEEE transactions like the *IEEE Transactions on Software Engineering*, the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, the *IEEE Transactions on Cybernetics*, and the *IEEE Transactions on Services Computing*.

Lu Zhang received the BSc and PhD degrees in computer science from Peking University, Beijing, China, in 1995 and 2000, respectively. He is a professor with the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a postdoctoral researcher with Oxford Brookes University and University of Liverpool, United Kingdom. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of the *Journal of Software Maintenance and Evolution: Research and Practice* and the *Software Testing, Verification and Reliability*. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.

Hong Mei (Fellow, IEEE) received the BA and MS degrees from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1984 and 1987, respectively, and the PhD degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 1992. From 1992 to 1994, he was a postdoctoral research fellow with Peking University. He is a professor with the Beijing Institute of Technology, Shanghai Jiao Tong University, and Peking University. He was the dean of the School of EECS, Peking University from 2006–2014, the vice president for research with Shanghai Jiao Tong University from 2013 to 2016, and had been the vice president for human resource and international affairs with the Beijing Institute of Technology since 2016. His research interests include software engineering and system software. He is a member of the Chinese Academy of Sciences, a fellow of the CCF and TWAS.