Shortening Overlong Method Names with Abbreviations

YANJIE JIANG, Peking University, China
HUI LIU*, Beijing Institute of Technology, China
SHING CHI CHEUNG, The Hong Kong University of Science and Technology, Hong Kong
LU ZHANG, Peking University, China

Methods should be named to summarize their responsibilities meaningfully. When a method has a non-trivial responsibility, it may require a naming using multiple words. However, overlong method names are susceptible to typos and reduced readability (e.g., displaying a statement partially in standard screen width or splitting it into multiple lines). Programming naming conventions commonly adopt a maximal length (in characters) for identifiers. In practice, developers may not necessarily find a meaningful name that follows such naming conventions when coding a non-trivial method. This paper presents the first automated technique (called `NameCompressor`) to shorten overlong method names. Our inspiration is that many lengthy words/phrases in an overlong method name have known and unambiguous abbreviations. The use of these abbreviations for method names is common. To shorten an overlong method name, `NameCompressor` employs three compression techniques, i.e., context-aware compression, probability-based compression, and machine learning-based compression, to find appropriate abbreviations for the words/phrases in the method name. We evaluate `NameCompressor` on a dataset of 700 overlong method names. It correctly generates 613 short names identical to those specified by the developers of these methods.

Additional Key Words and Phrases: Method Name, Abbreviation, Identifier, Code Quality

1 INTRODUCTION

Identifiers are indispensable for high-level programming languages, like Java, C, and Python. Such high-level programming languages employ numerous identifiers to name various software entities, aiming to improve the readability of complex source code. As a result of the heavy usage of identifiers, they account for up to 70% of source code in terms of characters [14]. Another consequence of the heavy usage of identifiers is that such identifiers become the major clue to the source code's readability, and thus the quality of identifiers has a crucial impact on the readability and maintainability of software applications.

Method names are a category of identifiers. Methods represent the smallest named units of aggregated behaviors [27, 41], and serve as a cornerstone of abstraction [8, 42]. They should be named to convey their intention (responsibility) [32, 35]. However, highly expressive and self-explanatory method names are often wordy, resulting in overlong method names. Typing overlong method names could be tedious and susceptible to typos. Overlong names may also result in lower readability, e.g., displaying a statement partially in standard screen width or splitting it into multiple lines. For example, it is hard to display the following statement, which consists of only five identifiers, on a single line in standard screen width.

*Corresponding author

Authors' addresses: Yanjie Jiang, yanjiejiang@pku.edu.cn, Peking University, China; Hui Liu, Beijing Institute of Technology, China, liuhui08@bit.edu.cn; Shing Chi Cheung, The Hong Kong University of Science and Technology, Hong Kong, scc@cse.ust.hk; Lu Zhang, Peking University, China, zhanglucs@pku.edu.cn.

ACM 1557-7392/2024/7-ART
https://doi.org/10.1145/3676959

```
ExceptionMessageHandingforLiabaryMethods(lib.ArgumentThrowsOtherThanNullPointerException(),
    getUtilityClass().getNullPointerExceptionMessage());
```

Breaking a single statement (like the preceding one) into multiple lines, however, can reduce its readability.

To reduce the use of overlong method names, experienced developers often replace the lengthy words or phrases in method names with their corresponding abbreviations when such abbreviations can convey the meaning of the original words or phrases. For example, the readability and meaning of the overlong method name "*getNullPointerExceptionMessage*" can be largely preserved by shortening it to "*getNPEmsg*". Notably, it remains debatable whether we should shorten method names (and other identifiers) with abbreviations. On one side, lengthy names may impose a burden on the memory (of software engineers) whereas the limited length of the identifiers may mitigate the burden [10]. On the other side, the major reason against the use of abbreviations is that they may reduce the readability of identifiers [16]. However, Lawrie et al. [27] found that abbreviations are often as understandable as longer names. Scanniello et al. [36] also find that properly used abbreviations in identifiers do not hinder novice programmers from locating and fixing faults. Even single-letter names could be high-quality identifiers if used properly [9]. We also observe that abbreviations are widely used in well-known and high-quality software applications. For example, in *SpringBoot* [6] and *Flink* [5], abbreviation-containing method names account for 22% and 20% of the method names, respectively. All such may suggest that in many cases developers tend to shorten lengthy names with abbreviations even though not all abbreviations (in identifiers) are beneficial.

However, manual abbreviations are susceptible to mistakes, especially for inexperienced developers or those unfamiliar with English. The abbreviations they choose can compromise the original method names' readability and meaning. To this end, in this paper, we propose the first automated approach (called NameCompressor) to shorten overlong method names with abbreviations that convey similar meanings. The approach is inspired by an observation that many long words/phrases in overlong method names have well-known and unambiguous abbreviations. Notably, automated shortening of method names involves two major challenges. First, the usage of abbreviations is often context-aware, depending on various factors like application domains, organization/-group/individual cultures, and the possibility of non-author developers reading/maintaining the code. Second, the appropriate abbreviation for a full term in a name often depends on the other terms in the name and the length of the name. As a result, it is often inaccurate to replace all full terms in abbreviation dictionaries with their corresponding abbreviations. To resolve the first challenge, we compress a given method name by first exploiting the usage of abbreviations within these method names that share highly similar contexts with the given method name. To resolve the second challenge, we build a rich abbreviation dictionary by mining open-source applications and identifying consistently used popular abbreviations. We also leverage a machine learning algorithm to synthesize various factors, e.g., the length of the method name, the number of abbreviations within the name, and the possibility for the full terms to be replaced with abbreviations.

We evaluate NameCompressor based on the method names in seven open-source applications. From each of the applications, we randomly pick up 100 method names containing abbreviations and follow a reproducible procedure to expand them into full words/phrases manually. As a result, we construct a dataset of 700 overlong method names. We apply NameCompressor to shorten these names and compare the shortened names with the original names used by the application developers. Out of the 659 shortened method names recommended by NameCompressor, 613 are correctly shortened in the sense that they are identical to the original names.

The paper makes the following contributions:

- We propose the first automated approach NameCompressor to shorten overlong method names. To realize the approach, we develop a context-aware name compression mechanism, build an abbreviation dictionary that augments each entry with its usage probability, and train a machine learning model for method name shortening.

- We construct a benchmark consisting of 700 overlong method names.
- We evaluate NameCompressor on the benchmark. Our experimental results confirm the effectiveness of NameCompressor. The NameCompressor tool, research artifact, and experimental results are made available to facilitate the reproduction of experimental results and future related studies [4].

2 BACKGROUND AND RELATED WORK

2.1 Automated Abbreviation of Paper Titles

AGRA proposed by Zhang et al. [43] is closely related to our work. "AGRA" stands for "*An Analysis-Generation-Ranking Framework for Automatic Abbreviation from Paper Titles*". AGRA is a deep-learning based approach to abbreviating a paper title into a single abbreviation (like "AGRA" itself). The approach is composed of three steps, i.e., description analysis, candidate generation, and abbreviation ranking. The abbreviation ranking is the key of the approach, which prefers abbreviations that are similar to existing words. The rationale of the preference is that such abbreviations are "usually easy to pronounce and familiar to people".

Our work is different from AGRA in two aspects. First, our approach focuses on method names whereas AGRA works on paper titles. Second, our approach leverages contexts of software entities, abbreviations discovered from open-source applications, and abbreviation expansion rules whereas AGRA does not leverage any of such information. In contrast, AGRA emphasizes the pronunciation of the resulting abbreviations (used as names of approaches or tools) so that they could be pronounced easily. Consequently, abbreviations that could be pronounced like normal words are preferred. However, most abbreviation-containing identifiers in source code, like "getNPEmsg", are not necessarily easy to pronounce. Consequently, it may not work to simply apply AGRA to overlong method names.

2.2 Abbreviations in Source Code

Abbreviations are common in source code, especially in identifiers [28, 29]. According to the case study conducted by Jiang et al. [23], 181,358 identifiers analyzed by them contain 66,622 abbreviations in total, and thus on average, every three identifiers contain one abbreviation. However, the benefits and drawbacks of such abbreviations are still debatable. Lawrie et al. [27] consulted over 100 developers and concluded that full-word identifiers (compared against identifiers with abbreviations) can improve software quality. However, they also found that in some cases replacing full terms with well-formed abbreviations would not significantly reduce software quality because shorter and meaningful identifiers are easier to remember than longer ones. Binkley et al. [10] conducted a case study to investigate the balance between longer, expressive names and limited programmer memory resources. The evaluation results suggest that longer names often take more time to process and reduce correctness in software engineering tasks. Scanniello et al. [36, 38] conducted a controlled experiment where students were asked to find and fix bugs in one of two alternative versions of the same program: One with full-word identifiers and the other with abbreviations. Their evaluation results suggest that abbreviations did not result in significant negative effect on the software engineering tasks (i.e., fault location and bug fixing). Hofmeister et al. [20] conducted a study with 72 professional C# developers, requesting them to locate bugs in source code snippets. The evaluation results suggest that full names can speed up fault localization compared to meaningless single letters and abbreviations. Similarly, Schankin et al. [37] requested 88 Java programmers to locate semantic defects in source code snippets, and their evaluation results suggest that longer and descriptive identifiers can speed up fault localization compared to shorter and less descriptive identifiers. However, the effect disappeared when the engineering task was changed to "searching for syntax errors".

The related work introduced in the preceding paragraph suggests that abbreviations in source code could be beneficial or harmful, depending on whether they are used properly.

2.3 Expansion of Abbreviations in Source Code

Expansion of abbreviations is to guess the full terms that the given abbreviations stand for. Expanding abbreviations in source code could be beneficial if the abbreviations are preventing developers and maintainers from understanding the source code. To this end, a few automated approaches have been proposed to expand abbreviations in source code, especially in identifiers [22].

An intuitive way to abbreviation expansion is to look up abbreviation dictionaries like that constructed by Adar [7]. However, such abbreviation dictionaries are often constructed manually, which significantly limits their size [11]. As a result, many abbreviations could not be expanded with such dictionaries [15]. Another problem with abbreviation dictionaries is that the same abbreviation may have different meanings depending on its context. As a result, looking up the abbreviation dictionaries alone may fail to select the correct expansion. To leverage the context of abbreviations, Corazza et al. [13], Lawrie et al. [26] and Guerrouj et al. [31] suggested searching for full terms in comments of source code. The rationale of such approaches is that comments in source code explain the semantics of source code, and thus it is likely that we can find the full terms of abbreviations from the comments associated with the source code where the abbreviations appear. Caprile et al. [11] and Carvalho et al. [12] suggested expanding abbreviations by looking for full terms from surrounding source code. Lawrie et al. [25] suggested matching a given abbreviation against full terms in the enclosing methods only. Hill et al. [19] proposed a complex approach to search for full terms in enclosing methods, enclosing classes, and enclosing projects in order. Jiang et al. [24] proposed an automated approach to expand parameter abbreviations only. By focusing on such a special subset of identifier abbreviations, the approach improves the performance significantly. However, it could not be applied to identifiers other than parameters. To break this obstruction, Jiang et al. [22] leveraged various semantic relationships among software entities, e.g., access of fields/variables, to abbreviation expansion. The rationale is that semantically related entities are likely to share some common concepts, and thus their names may contain some common terms. Newman et al. [33] conducted a study to analyze 861 abbreviation-expansion pairs extracted from five open-source applications, and their evaluation results suggest that documents are the most important source of full terms.

tfExpander proposed by Jiang et al. [21] represents the state of the art in this field. The key insight underlying this approach is that abbreviations in the name of software entity *e* are likely to be associated with their full terms in the names of other software entities that are semantically related to *e*. Based on this insight, tfExpander constructs a knowledge graph to represent software entities and their relationships, facilitating the search for full terms by graph-based search. Another key insight of *tfExpander* is that literally identical abbreviations within the same application are likely to have identical expansions. Therefore, expanding an abbreviation in one place may be transferred to lexically identical abbreviations in other places. Based on this insight, Jiang et al. [21] proposed transfer-learning-based expansion of abbreviations. Finally, they designed a series of heuristics and a learning-based approach to prioritize all heuristics inspired by the preceding key insights. Once an abbreviation is fed into *tfExpander*, it leverages the heuristics to expand it. Their evaluation results confirmed that *tfExpander* significantly improved the state of the art in abbreviation expansion by improving recall from 29% to 89% and improving precision from 39% to 92%. Notably, our approach leverages *tfExpander* to expand abbreviations in the context of the given method name, and tries to replace full terms in the given method name with such abbreviations. Consequently, the performance of *tfExpander* could substantially influence the performance of our approach. Besides that, in this paper, we also leverage *tfExpander* to build abbreviation dictionaries.

2.4 Method Body-based Name Generation

SGMNG, proposed by Qu et al. [34], represents the state of the art in generating method names according to method bodies. It combines the semantic and structural features of method bodies, which results in the best performance (accuracy=54%). Another novel approach *KG-MNGen* was proposed by Ge et al. [18]. It employs a

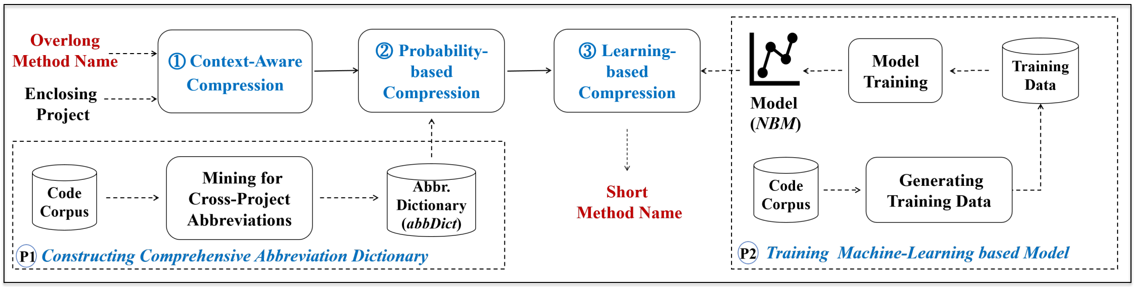
Overlong Method Name
Enclosing Project
① Context-Aware Compression
② Probability-based Compression
③ Learning-based Compression
Model (NBM)
Model Training
Training Data
Code Corpus
Mining for Cross-Project Abbreviations
Abbr. Dictionary (abbDict)
Short Method Name
Code Corpus
Generating Training Data
P1 Constructing Comprehensive Abbreviation Dictionary
P2 Training Machine-Learning based Model

Fig. 1. Overview of NameCompressor

graph neural network to extract keywords from method bodies, and utilizes a seq2seq model to generate method names. DeepName [30], a novel approach proposed recently, also generates method names according to method bodies although its ultimate goal is to resolve inconsistent names. Zügner et al. [44] proposed a new model to learn the context and structure of source code by using language-agnostic features. As an application of the approach, they leveraged their approach to generate method names based on method bodies.

Our work differs from such approaches in that it compresses existing overlong method names whereas method body-based method name generation approaches generate method names from scratch. Another difference is that our approach does not exploit method bodies on which their approaches heavily rely.

3 APPROACH

3.1 Overview

Fig. 1 presents the overview of `NameCompressor`. Overall, it shortens an overlong method name with three sub-algorithms: (1) context-aware compression, (2) probability-based compression, and (3) learning-based compression. Notably, the approach depends on two preprocessing, i.e., (p_1) constructing a rich abbreviation dictionary (noted as *abbDict*), and (p_2) training a machine-learning based model (noted as *NBM*).

For a given method name *mn*, `NameCompressor` shortens it as follows:

- First, `NameCompressor` collects all `<abbreviation, full term, frequency>` triples (noted as *abbList*) from method signatures within the enclosing class of *mn*. It then sorts *abbList* in descending order of frequency. For each triple <*abb*, ft, fc_1>, `NameCompressor` replaces the full term ft in *mn* with abbreviation *abb*.
- Second, on the resulting name mn', `NameCompressor` abbreviates such full terms that have high popularity and high certainty of being replaced with given abbreviations according to the cross-project abbreviation dictionary *abbDict*. The resulting name is noted as mn''.
- Finally, based on the trained `BernoulliNB` model *NBM*, `NameCompressor` predicts which phrase in mn'' has the greatest possibility to be abbreviated and which abbreviation should be used. It repeats the prediction (and abbreviation) until no more phrases could be abbreviated. The resulting name mn''' is recommended as the shortened name.

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

3.2 Context-Aware Compression

It is quite often that the same phrase (full term) is used as it is in one application, but it is replaced by an abbreviation in another application. It is because the usage of abbreviations is often context-aware. The usage

Algorithm 1: Context-Aware Compression

```
Input: $mn$; // the original method name
       $ec$// enclosing class
Output: $mn$; // short name
$ms \leftarrow MethodSignatures(ec)$ ;
$allNames \leftarrow Identifiers(ms)$ ;
$pairs = tfExpander.Expand(allNames)$;
// pair=<abb,fterm>
$abbList' = abbList = \emptyset$ ;
for each $pair$ in $pairs$ do
    item=abbList.Retrieve(pair.abb,pair.fterm);
    if $item! = null$ then
        item.freq++ ;
    else
        abbList.add(pair.abb,pair.fterm,1);
    end
end
// keep only one item for each unique full term
for each $e$ in $abbList$ do
    $item$=$abbList'.Retrieve(e.fterm)$;
    if $item == null$ then
        $abbList'.add(e)$;
    else if $item.freq < e.freq$ then
        item.freq=e.freq ;
        item.abb=e.abb;
    end
end
// sort in descending order of frequency
$abbList'.sortByFreq()$;
for each $item$ in $abbList'$ do
    // shorten full term $fterm$ according to the selected item <$fterm$, $abb$>
    $mn$.replace($item.fterm$,$item.abb$);
end
// the resulting name $mn$ should be further processed by the next compression strategy
return $mn$;
```

depends on various factors, e.g., the domain of the application, the programming guidelines employed by the team, and the probability for the code to be read/maintained by non-author developers. We call all such factors together the *context* of an abbreviation. A big challenge is that it is difficult to accurately assess the context of an abbreviation or how the context influences the usage of the abbreviation.

To address the difficulty, our insight is to reference the usages of other methods that share highly similar contexts with the method under test. In this paper, we call this mechanism `context-aware compression` (CAC for short). For a given method *m* within class *ec*, CAC shortens its method name *mn* as specified in Algorithm 1.

First (at lines 1-2 of Algorithm 1), CAC collects all method signatures declared within the enclosing class *ec*, except for the method under compression (i.e., *m*). Each collected method signature consists of the return type of the method, the method name, parameters, and the data types of the parameters. Second (Line 3), CAC employs tfExpander [21] to expand abbreviations in the resulting method signatures. During the expansion, CAC collects all <`abb`, `fterm`> pairs where `abb` is an abbreviation appearing in the explored identifiers and `fterm` is the corresponding full term suggested by tfExpander. Notably, if the same abbreviation appears multiple times and multiple occurrences of the abbreviation are expanded into the same full term, the pair <`abb`, `fterm`> may appear multiple times. To this end, we record how many times such pairs appear with a triple <`abb`, `fterm`, `freq`> where `freq` is the frequency (Lines 5-12). All such triples are collected as a list *abbList*. Notably, we transfer plural terms (and their corresponding abbreviations) into singular before counting the frequency, and thus <"Dict", "Dictionary"> and <"Dicts", "Dictionaries"> are counted together. The counting is case-insensitive, and thus <"Dict", "Dictionary"> and <"dict","dictionary"> should be counted together. At lines 13-21 of Algorithm 1, it resolves conflicting items from the list: Two triples <abb_1, $fterm_1$, $freq_1$> and <abb_2, $fterm_2$, $freq_2$> are conflicting if $fterm_1$ equals to $fterm_2$. They are conflicting because the same full term would be shortened into different abbreviations according to the conflicting triples. To this end, `NameCompressor` keeps only the item with the highest frequency for each unique full term.

After resolving conflicting triples, `NameCompressor` ranks triples in $abbList'$ by frequency (`freq`) and thus the most popular abbreviations are ranked at the top (Line 22). For each triple <`abb`, `fterm`, `freq`>, `NameCompressor` replaces all occurrences of the `fterm` in *mn* with `abb` (Lines 23-25). We also replace the plural of `fterm` within *mn* with the plural of *abb*. For example, if $fterm$="Dictionary" and *abb*="Dict", we should replace all occurrences of "Dictionary" with "Dict", and occurrences of "Dictionaries" with "Dicts". Note that the word matching at line 24 is case-insensitive (so "dictionary" matches $fterm$="Dictionary") whereas the replacement is case-sensitive (so "Dictionary" and "dictionary" should be replaced with "Dict" and"dict", respectively).

3.3 Probability-based Compression

The context-aware compression presented in the preceding section may fail to replace full terms/phrases with widely used abbreviations if such abbreviations are not used by the method signatures within the same class. Consequently, in this section, we propose probability-based compression to shorten method names with widely used abbreviations automatically mined from open-source applications. The compression mechanism consists of two parts. The first part, as introduced in Section 3.3.1, builds an abbreviation dictionary by mining a large code base. The second part, as introduced in Section 3.3.2, shortens method names with the resulting dictionary.

3.3.1 Building rich Abbreviation Dictionary. The key idea of building a rich abbreviation dictionary is that most abbreviation dictionaries provide little information beyond <`abbreviation`, `full term`> pairs. To the best of our knowledge, none of the existing abbreviation dictionaries provide the exact probability for the full term $fterm$ to be replaced with a given abbreviation *abb*, or the exact probability for *abb* to represent $fterm$. Without such information, it is difficult for `NameCompressor` to decide which words/phrases could be replaced safely.

We build the abbreviation dictionary by expanding the abbreviations found in open-source applications. The expansion adapts the state-of-the-art abbreviation expansion technique tfExpander [21]. We do not simply reuse tfExpander because our application scenario is substantially different from that of traditional abbreviation expansion. tfExpander was originally designed for traditional abbreviation expansion. tfExpander had better suggest some likely full terms whenever an abbreviation is given, and the suggested full terms could be double-checked by developers. Consequently, tfExpander prefers a balance between its precision and recall. In contrast, to build an abbreviation dictionary by mining open-source applications, we do not have to expand all abbreviations: An abbreviation not expanded in one place may be expanded successfully in another place. Consequently, we may stand reduced recall for improved precision (i.e., fewer incorrect expansions). To this end, we set the following post-conditions to exclude some risky expansions:

- An abbreviation and its full term should share the same initial character. According to tfExpander, an abbreviation (e.g., "*nto*") matches a full term (e.g., "*translator*") if the former is a subsequence of the latter. This matching algorithm is known as "dropped letters". However, according to our experience, this algorithm often results in false positives especially when the abbreviation and the full term have different initial letters.
- An abbreviation should not be expanded into full terms that already exist in its enclosing identifier. For example, the abbreviation "*exo*" in the identifier "*ExoPlaybackException*" should not be expanded into "*exception*" because the latter itself already exists in the identifier. The rationale for the exclusion is that it is less likely for developers to employ an abbreviation and its full term at the same time within a single identifier.

With the adapted expansion algorithm, `NameCompressor` builds the abbreviation dictionary `abbDict` by Algorithm 2. `NameCompressor` collects all method names (except for constructors and destructors) within the given code base, and all such names together are noted as $allNames$ (Line 1 of Algorithm 2). Constructors and destructors are excluded because their names are essentially class names, instead of ordinary method names. It leverages the adapted algorithm to expand abbreviations in the collected method names (Line 2). Each of the expanded abbreviations results in a pair `<abb, fterm>`. All such pairs together are noted as a list $pList$. For each full term in $pList$, `NameCompressor` enumerates all cases where the full term appears as it in the collected identifiers ($allNames$), and appends $pList$ an item `<fterm, fterm>` for each case (Line 3). Based on the expansion, `NameCompressor` (Lines 4-12) builds a dictionary `abbDict` where each item is a quintet:

$$< fterm, abb, freq, p_1, p_2 > \tag{1}$$

`freq` and p_1 represent how often and how likely the full term `fterm` in method names are replaced by abbreviation `abb`, respectively. p_2 represents how likely the abbreviation `abb` in method names represents full term `fterm`. Detailed computation is presented at Lines 13-20 in Algorithm 2. Notably, if `abb` equals to `fterm`, p_1 represents the probability for the full term `fterm` to be used as it (instead of any shorter abbreviations) in method names.

3.3.2 *Shortening Names with Abbreviation Dictionary.*

`NameCompressor` employs the Algorithm 3 to shorten the overlong method name *mn* with the abbreviation dictionary `abbDict`. Notably, method `Shorten(mn,abbDict)` at line 1 shortens method name *mn* by using at most one abbreviation from `abbDict`. If it fails to shorten the name *mn* (i.e., the resulting name is identical to the original name), the probability-based abbreviation terminates. Otherwise, it takes the resulting short name as input (Line 3) and tries to shorten it in the same way (Line 4).

Details of `Shorten(mn,abbDict)` are presented at lines 7-21. It first splits the method name into a sequence of tokens according to capital letters, underscores, and digits (Line 8). Based on the sequence, at line 9 it invokes `EnumeratePhrases` to retrieve all potential phrases (noted as `phrases`) within the method name. It sorts the phrases (Line 10) so that longer phrases appear on the top. It retrieves dictionary items in `abbDict` whose full terms equal the longest phrase (noted as `phrase`) in `phrases`. At line 16, it ranks the resulting dictionary items and tries to find the item with the greatest p_1 (as defined in Equation 1). If p_1 is greater than a threshold (empirically set to 0.6), it shortens the method name according to this dictionary item (Line 18) and returns the resulting name. Once any phrase has been shortened, the method `Shorten` terminates, leaving abbreviation opportunities for the next iteration.

Algorithm 2: : Constructing Abbreviation Dictionary

```
Input: $cd$; // the given code base
Output: $abbDict$ // the abbreviation dictionary
$allNames \leftarrow RetrieveMethodNames(cd)$ ;
// expand abbreviations in method names with adapted tfExpander
$pList = tfExpander'.Expand(allNames)$;
// collect full terms used in method names
$pList$.add(ScanFullTerms(allNames,pList.getFullTerms())) ;
// count the frequency for each dictionary entry
for each $tuple$ in $pList$ do
    $item$=$abbDict.retieve(tuple)$;
    if $item! = null$ then
        item.freq++ ;
    else
        $item =< tuple.fterm, tuple.abb, 1, 0, 0 >$ ;
        abbDict.add(item);
    end
end
for each $item$ in $abbDict$ do
    // all items where fterm=item.fterm
    $fItems$=$abbDict.retrieveByFterm(item.fterm)$;
    // all items where abb=item.abb
    $abbItems$=$abbDict.retrieveByAbb(item.abb)$;
    // sum of $freq$
    $fTotal$=TotalFrequency($fItems$);
    $abbTotal$=TotalFrequency($abbItems$);
    $item.p_1$=$item.freq/fTotal$ ;
    $item.p_2$=$item.freq/abbTotal$;
end
return $abbDict$
```

3.4 Learning-based Compression

The context-aware compression in Section 3.2 and probability-based compression in Section 3.3 are often highly reliable. However, the context-aware compression is confined to abbreviations within the same document whereas probability-based compression focuses on consistently used common abbreviations only. Consequently, it is likely that they may fail to shorten some lengthy words/phrases within overlong method names. To this end, in this section, we propose a learning-based technique to further shorten overlong method names. The key rationale is that from a large number of real-world samples, machine learning techniques have the potential to learn how to shorten method names.

3.4.1 Features and Learning Model. For a given overlong method name *mn* that has been shortened into *mn*′ by the context-aware compression in Section 3.2 and probability-based compression in Section 3.3, learning-based compression works as follows. First, `NameCompressor` enumerates all phrases (and single words) in the same

Algorithm 3: : Probability-based Abbreviation

```
   Input: $mn$; // the overlong method name
          $abbDict$ // the abbreviation dictionary
   Output: $mn'$ // shortened method name
 1 $mn' = Shorten(mn, abbDict)$;
 2 while $mn'$ != $mn$ do
 3     mn = $mn'$;
 4     $mn'$ = Shorten(mn, abbDict)
 5 end
 6 return $mn'$;
   // shorten $mn$ with $abbDic$
 7 Function Shorten($mn, abbDict$):
 8     tokens = split(mn); // split $mn$ into tokens
 9     phrases = EnumeratePhrases(tokens);
       // longer phrases appear on the top
10     phrases.SortByLength();
11     for each $phrase$ in $phrases$ do
12         items = abbDict.RetrieveByFterm(phrase);
13         if $items == null$ then
14             continue;
15         end
16         $items.SortByP_1()$;
17         if $items[0].p1>0.6$ then
18             return mn.Replace(phrase, items[0].abb);
19         end
20     end
21 return $mn;$
   // retrieve all potential phrases
22 Function EnumeratePhrases($tokens$):
23     $phrases \leftarrow \emptyset$ ;
24     for $i=0; i<tokens.len;i++$ do
25         for $j=i;j<tokens.len;j++$ do
26             phrases.add(tokens.subtokens[i,j]);
27         end
28     end
29 return $phrases;$
```

way as specified in Algorithm 3 (Lines 22-29). The resulting phrases are presented as a list:

$$phs =< ph_1, ph_2, \dots, ph_n > \qquad (2)$$

For each phrase ph_i, `NameCompressor` retrieves all items in the given abbreviation dictionary *abbDict* whose full terms are equal to ph_i. The retrieved items are presented as a list:

$$DictItems(ph_i) =< it_{i,1}, it_{i,2}, \dots, it_{i,k} > \tag{3}$$

Each of the items follows the definition in Equation 1. For each item $it_{i,j}$, `NameCompressor` should leverage a machine learning-based model to predict how likely that we can safely replace the phrase ph_i with the abbreviation in dictionary item $it_{i,j}$ (i.e., $it_{i,j}.abb$). To facilitate the prediction, `NameCompressor` extracts the following features as input to the model:

$$\begin{aligned} ft &= ExtractFeature(ph_i, it_{i,j}.abb, mn', abbDict) \\ &= < len_b, len_a, \Delta len, abbs, its, f, p > \end{aligned} \tag{4}$$

where len_b and len_a are the lengths (in characters) of the method name mn' before and after replacing phrase ph_i with $it_{i,j}.abb$. $\Delta len = len_b - len_a$ represents to what extent the replacement has shortened the method name. *abbs* represents how many phrases in *mn* have been replaced with abbreviations by context-aware compression and probability-based compression. *its* represents how many phrases in mn' have been replaced with abbreviations by the learning-based compression (i.e., by the current model). f equals $it_{i,j}.freq$, and p equals $it_{i,j}.p_1$.

The learning-based model M takes the feature ft as input and generates the probability to replacing ph_i in mn' with $it_{i,j}.abb$:

$$pos = M(ft) \tag{5}$$

We leverage BernoulliNB [1] to simulate the mapping M. BernoulliNB is a Naive Bayes classifier based on a multivariate Bernoulli model. We leverage a traditional machine learning technique instead of more advanced deep learning techniques because we prefer a lightweight model. We will integrate the proposed approach into IDEs, and expect it to be distributed to developers with the standard IDEs. However, employing advanced deep-learning based models, like Transformer [39], may result in a large model that is often larger than one GB. It is unlikely for IDEs to accept such large models only for one feature. For example, the size of the whole Eclipse is 120 MB, even smaller than a single deep-learning model. Besides, such deep learning-based models often rely heavily on GPUs that may not be available on the hosts of the IDEs.

Among all of the items tested by the model (i.e., all potential replacements of the phrases in *phs*), `NameCompressor` selects the one with the greatest output (of the model). If the greatest output is smaller than a predefined threshold β, `NameCompressor` terminates the learning-based compression and returns mn' as the suggested method name. Otherwise, it shortens mn' as suggested by the selected item. Notably, after replacing the first phrase (on the first iteration), `NameCompressor` would try to shorten the resulting name again (on the next iteration) in the same way unless the current iteration fails to shorten any phrase.

3.4.2 Generation of Training Data. The learning-based compression introduced in the preceding paragraph is based on the assumption that the model has been well trained. To this end, we employ the following algorithm to generate training data based on the given code base (high-quality applications). For each method name *mn* within the given code base, the training data generation process (DataGen) works as follows:

(1) DataGen leverages tfExpander to expand abbreviations in *mn*. If *mn* does not contain any abbreviation or tfExpander fails to expand any abbreviation in *mn*, DataGen terminates. Otherwise, it expands *mn* into mn' by replacing abbreviations $ABB =< abb_1, abb_2, \dots, abb_k >$ with their corresponding full term phrases $PHS =< ph_1, ph_2, \dots, ph_k >$.
(2) DataGen randomly selects a full term phrase ph_i in mn' and replaces it with its corresponding abbreviation abb_i. The replacement turns method name mn' into mn''.
(3) DataGen generates an input item (to the model) by invoking $ft = ExtractFeature(ph_i, abb_i, mn', abbDict)$ that is defined by Equations 4-5. Since the ground truth (*mn*) suggests that the full term phrase ph_i in mn'

should be replaced by abb_i, the expected output of the model (i.e., $M(ft)$) is one. For convenience, we call such items whose expected output equals one as `positive items`.

(4) From method name mn', DataGen randomly selects a full term ft that 1) is not introduced by the expansion on the first step, i.e., $ft \notin PHS$; and 2) has one or more corresponding abbreviations (noted as $Abbs(ft)$) according to the given abbreviation dictionary $abbDict$.

(5) DataGen generates an input item (to the model) by invoking $ft = ExtractFeature(ft, abb, mn', abbDict)$ where abb is randomly selected from $Abbs(ft)$. Since ft should not be shortened according to the ground truth (*mn*), the expected output for this item (i.e.,$M(ft)$) should be zero. We call such items (with zero as expected output) `negative items`.

(6) DataGen removes ph_i from *PHS*, and removes abb_i from *ABB*. It also assigns mn'' to mn', and turns back to the second step for the next iteration. DataGen terminates when *PHS* is empty.

The key insight of the data generation is that for the given high-quality method name *mn* and its expanded version mn', we know exactly which phrases in the longer version (mn') should be shortened (with which abbreviations), and which phrases should not be shortened. As a result, we can create positive items based on should-be-shorten phrases and negative items based on should-not-be-shorten phrases. By creating a positive item (the third step) and a negative item (the fifth step) on each iteration, we achieve a balance between negative and positive items. Notably, the size of the training data could be increased as needed because the data generation is completely automated and there are plenty of high-quality open-source applications.

4 EVALUATION

4.1 Setup

4.1.1 Research Questions. The evaluation is designed to investigate the following research questions:

- **RQ1:** Can `NameCompressor` accurately shorten real-world method names?
- **RQ2:** Can `NameCompressor` outperform potential alternatives?
- **RQ3:** How do different parts of `NameCompressor` contribute to the overall performance?
- **RQ4:** Is the performance of `NameCompressor` sensitive to threshold settings?
- **RQ5:** Is the performance of `NameCompressor` sensitive to the types of full terms or the length of inputted names?
- **RQ6:** How often do developers prefer short method names generated by `NameCompressor`?

4.1.2 Dataset. The evaluation used three datasets. The first dataset, called `Dict-Data`, was used to build a rich abbreviation dictionary. The second dataset, called `Training-Data`, was used to train the learning-based model proposed in Section 3.4. The last dataset, called `Testing-Data`, was used to evaluate the proposed approach. To construct these datasets, we retrieved the top 1,207 Java applications from GitHub (sorted by stars): The top 7 applications were used to create `Testing-Data`, the next 1,000 are taken as `Dict-Data`, and the last 200 were used to create `Training-Data`. Note that `Dict-Data` did not require preprocessing. `Training-Data` were processed automatically by the proposed approach (as specified in Section 3.4.2).

We constructed `Testing-Data` from the method names in the top 7 applications as follows. From each application, we randomly sampled 100 method names (except for constructors and destructors) that contain abbreviated terms. Two software engineers (with more than five years of Java experience) manually expanded the abbreviated terms in the sampled method names (referred to as `original names`). The expanded names were then used for the `Testing-Data`, and their original names served as the ground truth for `Testing-Data`. The two software engineers expanded all the names independently. On 677 sampled names, they resulted in identical expansions. In 15 cases, at least one of them failed to expand the names, and thus such names were discarded. In 5 cases, their initial expansions were inconsistent but they succeeded in reaching a consensus after discussion. In

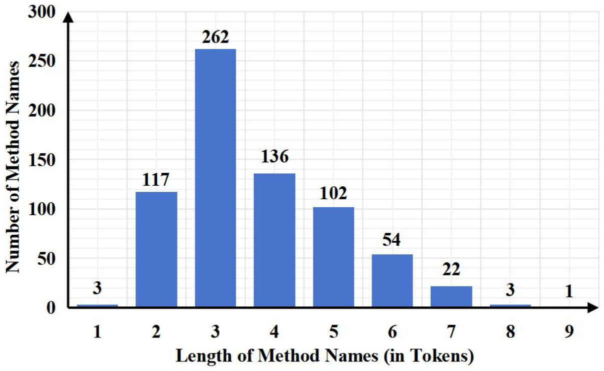

Fig. 2. Length of Method Names

3 cases, they failed to reach any consensus, and thus the names were discarded. We replaced the 18 discarded names with additional samples. As a result, we built a benchmark (noted as `abbInMethodNames`) consisting of 700 overlong method names. It contains 2,588 tokens in 700 long method names. The distribution of long method names based on their token size is presented in Fig.2. The longest method name has 9 tokens, and the most(90.8%) method names have more than 3 tokens.

Notably, the abbreviations in the open-source software are not necessarily the optimal selection for the enclosing method names. To validate how often such abbreviations in the benchmark are optimal, we invited three experienced developers to check the identifiers manually. They all had more than three years of programming experience and more than four years of Java experience. First, we asked one developer to manually compress the long method names in the benchmark and asked the other two developers to check whether the manually compressed names were identical to the ground truth. If not, we requested them to discuss together whether the original short name should be replaced by the manually constructed short name or the latter should be added as an alternative ground truth, i.e., the given long method name could be shortened into either this newly added method name or the original method name as presented in the original source code. In total, they added 8 short method names in `abbInMethodNames` whereas none of the original short names were replaced. A typical example is "*getParameterTypesDescription*" which was shortened as "*getParameterTypesDesc*" by the original developer. However, the participant shortened it into "*getParamTypesDesc*". The participants agreed that it was comparable to the original one, and "*getParameterTypesDescription*" could be shortened as either "*getParameterTypesDesc*" or "*getParamTypesDesc*".

4.1.3 Process and Metrics. The evaluation was conducted as follows. First, we leveraged the proposed approach to build the abbreviation dictionary *abbDict* with `Dict-Data` (1,000 applications) collected in the preceding section. Second, we leveraged the proposed approach to generate training data with `Training-Data` (200 applications) collected in the preceding section, and trained the learning-based model of the approach. Third, we applied `NameCompressor` to the 700 manually expanded method names in `Testing-Data`, compared the shortened names with the original method names specified by developers (i.e., the method names that appear in the selected applications), and computed the performance.

Table 1. Performance of NameCompressor

Applications	Identifier-Level			Token-Level		
	Precision (P_{id})	Recall (R_{id})	F_1	Precision (P_{token})	Recall (R_{token})	F_1
Apollo	97%	90%	93%	99%	93%	96%
Batik	91%	89%	90%	93%	86%	89%
Dubbo	89%	85%	87%	97%	93%	95%
Guava	93%	85%	89%	98%	91%	95%
Jadx	94%	91%	92%	96%	93%	95%
Springboot	95%	90%	92%	98%	95%	97%
Zookeeper	92%	83%	87%	96%	90%	93%
Average	93%	88%	90%	97%	92%	94%

We leveraged the widely used precision and recall to assess the performance in method name compression. We assessed the identifier-level performance with the following adapted precision and recall:

$$P_{id} = \frac{\#CorrectlyShortenedNames}{\#ShortenedNames} \tag{6}$$

$$R_{id} = \frac{\#CorrectlyShortenedNames}{\#TestedNames} \tag{7}$$

where P_{id} and R_{id} represent the precision and recall of the compression in terms of identifiers. Note that a method name is `correctly shortened` if and only if the shortened name is the same as the original one specified by developers.

To assess the performance in terms of fine-grained tokens, we used the following metrics:

$$P_{token} = \frac{\#CorrectlyShortenedTokens}{\#ShortenedTokens} \tag{8}$$

$$R_{token} = \frac{\#CorrectlyShortenedTokens}{\#TokensShouldBeShorten} \tag{9}$$

where P_{token} and R_{token} represent the precision and recall in terms of tokens. A token is shortened correctly if and only if its shortened form is the same as that adopted by developers.

Besides the precision and recall, we also used F_1 score that considers both precision and recall in our assessment.

4.2 RQ1: Performance

We applied `NameCompressor` to the selected 700 overlong method names (as introduced in Section 4.1.2). The evaluation results are presented in Table 1, which shows that `NameCompressor` can achieve high F_1 scores for the compression of identifiers and tokens. The average identifier-level precision and recall are 93% and 88%, respectively. More than nine out of ten times, the shortened names recommended by `NameCompressor` were identical to those specified by developers. The high precision is desirable because generating many incorrect abbreviations could turn developers away from using the assistant tool. `NameCompressor` was not only highly reliable but also powerful: Most (88% on average) of the overlong method names were shortened correctly by `NameCompressor`. The high recall suggests that `NameCompressor` has the potential to be used frequently.

The token-level performance was even more promising: The token-level precision and recall were further improved to 97% and 92%, respectively. It suggests that almost all the abbreviations introduced by `NameCompressor` were the same as what developers have manually coined. It is not surprising that token-level outperforms identifier-level. Only when all tokens within a given identifier are shortened correctly, the compression of the identifier as a whole is correct.

Our further analysis suggests that `NameCompressor` shortened 659 out of the 700 overlong method names, where 613 were shortened correctly and 46 were shortened incorrectly. Among the 46 that were shortened incorrectly, 5 were shortened incorrectly because `NameCompressor` shortened more terms than those specified by developers. An example is "getApplicationDirectory". The original name is "getAppDirectory", while the shortened name given by `NameCompressor` is "getAppDir". `NameCompressor` shortened "Directory" to "Dir", which is widely used to represent "Directory" in open-source applications. "Directory" appears 404 times in method names in the selected 1,000 applications whereas "Dir" appears 2,141 times, and in most cases (84%) "Dir" represents the full term "Directory". However, the developers in this case used the full term "Directory" in the method name.

Among the 46 incorrectly shortened names, 12 were partially shortened. `NameCompressor` correctly shortened some (but not all) terms that were manually shortened by the original developers. An example is "getApplicationIdentifier". The original method name is "getAppID", while `NameCompressor` suggests "getAppIdentifier". `NameCompressor` failed to replace "Identifier" with "ID" because the latter did not appear in sibling method names, and it was not a common abbreviation in the selected 1,000 applications. Enlarging the code base for future dictionary mining might avoid some cases.

Out of the 700 tested method names, `NameCompressor` failed to shorten any terms in 41 method names, which had a substantial negative impact on its identifier-level recall although it did not influence the precision of `NameCompressor`. An example is the method name "FloatPointPattern". The original developer shortened it to "fpPattern" whereas `NameCompressor` suggested not to replace any terms. `NameCompressor` did not shorten "FloatPoint" because it was uncommon in the selected code base, and it was not used by any other method names within the same class. It may suggest that `NameCompressor` could miss uncommon abbreviations if they do not appear within the same class.

The experimental results suggest that `NameCompressor` can generate accurate abbreviations for long method names.

4.3 RQ2: Potential Baselines and Alternatives

`AGRA` proposed by Zhang et al. [43] is the most relevant work, and is potentially the best baseline for comparison. It is a deep-learning-based approach to abbreviating paper titles. However, we failed to get its implementation even though we had tried to contact the authors. Approaches that generate method names from scratch according to method bodies were not taken as baselines, either. Our approach requests the original overlong method names whereas such approaches do not exploit the original names, and thus the comparison could be unfair.

Shortening identifiers is a sequence2sequence task of strong resemblance to natural language translation (NLT). Consequently, we can likely reuse advanced deep-learning-based NLT models to translate an overlong method name into a short one. To investigate the possibility, we employed the widely-used NLT model Transformer [40] for this task, taking the input (a method name) as a sequence of characters and the output (the short name) as another sequence of characters. Notably, we did not leverage well-known pre-trained NLT models (like CodeT5) because they had been pre-trained with sequences of words (common natural languages) whereas our data are sequences of characters. To the best of our knowledge, Transformer is the state-of-the-art NLT model that could be trained from scratch with sequences of characters. We trained Transformer with the 1,000 open-source applications in *Dict-Data* collected in Section 3.4.2. We expanded all (256,191 in total) abbreviation-containing method names by tfExpander. The expanded names and the original names served as the inputs and expected outputs of Transformer, respectively. We applied the resulting Transformer to the 700 overlong method names

Table 2. Impact of Different Components

Setup	Identifier-Level			Token-Level		
	Precision	Recall	F_1	Precision	Recall	F_1
Default	93%	88%	90%	97%	92%	94%
Disabling CAC	83%	69%	75%	93%	81%	87%
Disabling PBC	91%	85%	88%	96%	90%	93%
Disabling LBC	93%	81%	87%	97%	88%	92%

used to evaluate NameCompressor in Section 4.2. Our evaluation results suggest that Transformer was less accurate than NameCompressor. Out of the 700 method names, only 219 were shortened correctly by Transformer, substantially fewer than those (613) correctly shortened by NameCompressor. Its identifier-level precision and recall are 40% and 31% respectively, substantially lower than those (93% and 88%) of NameCompressor.

We also took the state-of-the-art large language model, GPT-4, as a baseline. General large language models (LLMs) are proficient in natural language understanding and generation. The versatility of chatGPT4 allows it to understand the contexts of a method name. Consequently, GPT-4 has the potential to compress long method names. To leverage GPT-4, we manually designed a prompt template as follows:

"*Please try your best to shorten the long method name by using their official or commonly used abbreviations and context. Please ensure: (1) the shortened method name has exactly the same semantics as the original one;(2) the shortened method name should follow the camel case naming convention; and (3) the shortened name should follow good coding practice. The long method name is: "[ORIGINAL NAME]", and its context is:"[CONTEXT]". Please output the shortened method name only without any explanation.*".

Our evaluation results suggest that GPT-4 correctly shortened 455 out of the 700 method names. Its identifier-level precision and recall are 66% and 65%, respectively, which are substantially lower than that of NameCompressor. We take the method name "*handleDocumentObjectModelAttributeModifiedEvent*" as an example for explanation. The original developers of the project shortened the method name as "*handleDOMAttrModifiedEvent*", i.e., shortening "*Attribute*" as "*Attr*", and "*DocumentObjectModel*" as "*DOM*". Although the proposed approach shortened it into "*handleDOMAttrModifiedEvent*" correctly, GPT-4 generated a short method name "*handleDocObjModelAttrModifiedEvt*" that is substantially different from the ground truth. The major reason for the failure is that GPT-4 tried to shorten each token in the phase "*DocumentObjectModel*" independently, i.e., replacing "*Document*" with "*Doc*", replacing "*Object*" with "*Obj*". In contrast, our approach took the phase as a whole and retrieved its corresponding abbreviation "*DOM*". Because the resulting method name remains too long after replacing "*Document*" and "*Object*", GPT-4 further shortened the method name by replacing "*Attribute*" with "*Attr*" and replacing "*Event*" with "*Evt*". Our approach did not make such a replacement because its resulting method name was not lengthy anymore.

4.4 RQ3: Effect of Different Components

As shown in Fig. 1, the proposed approach is composed of three parts, i.e., context-aware compression (CAC), probability-based compression (PBC), and learning-based compression (LBC). To investigate the effect of each component, we repeated the evaluation in Section 4.1.3 by disabling them one at a time. Our evaluation results are presented in Table 2. From Table 2, we make the following observations:

- Disabling the context-aware compression (CAC) reduced both the precision and recall at identifier and token levels. The F_1 score for the identifier and the token level was reduced from 90% to 75% and from 94% to 87%, respectively.

Table 3. Effect of Individual Components

Setup	Identifier-Level			Token-Level		
	Precision	Recall	F_1	Precision	Recall	F_1
Default	93%	88%	90%	97%	92%	94%
CAC Only	92%	78%	85%	97%	83%	89%
PBC Only	87%	65%	75%	95%	78%	86%
LBC Only	77%	51%	62%	89%	58%	70%

- Disabling the learning-based compression (LBC) reduced the recall of the approach, especially the recall for the identifier level reducing from 88% to 81%.
- Disabling the probability-based compression (PBC) also resulted in a reduction in both precision and recall. It reduced the identifier-level precision and recall by two percent points, respectively. The reduction was smaller than that caused by disabling CAC or LBC. One possible reason is that LBC complements PBC. The full terms that should have been shortened by PBC, may finally be shortened by LBC. Notably, both LBC and PBC exploited the items within the given abbreviation dictionary, and therefore they can overlap.

Besides that, we also investigated the effect of individual components. We employed each component once (i.e., disabling other components) and repeated the evaluation in Section 4.1.3. The evaluation results are presented in Table 3. The table shows that the context-aware compression (CAC) was accurate. Its precision was 92% and 97% at identifier-level and token-level, respectively. However, its recall (78% at the identifier level and 83% at the token level) was substantially lower than that of the proposed approach. We also note that probability-based compression (PBC) and learning-based compression (LBC) were less accurate than CAC and NameCompressor. Nevertheless, both compressions successfully shortened many long method names that CAC failed to shorten, which helped improve the overall recall of the proposed approach. In total, they successfully shortened 67 method names that CAC refused to shorten, which substantially improved the recall from 69% to 88% (at identifier-level) and from 81% to 92% (at token-level).

We conclude based on the preceding analysis that all components of `NameCompressor` were useful.

4.5 RQ4: Thresholds' Influence

The proposed approach has two thresholds α and β. The former controls whether an item from the abbreviation dictionary could be used (i.e., the minimal p_1) whereas the latter controls whether the replacement suggested by the learning-based model should be adopted (i.e., the minimal confidence in prediction). Although they were empirically set to 0.6 and 0.95, respectively, changes to the setting may have substantial influence on the overall performance. To investigate this issue, we changed the setting and recorded how it influenced the performance.

The results are presented in Fig 3. From this figure, we observe that changing the setting did influence the performance of the proposed approach, including both precision and recall. We observe that the empirically set thresholds (i.e., $\alpha = 0.6$ and $\beta = 0.95$) resulted in the best performance. However, we also observe that slightly changing the values around the default setting resulted in minor changes in the performance of the approach.

4.6 RQ5: Sensitivity

It is likely that some special categories of terms, like "United Kingdom" and "Internet Protocol" are easier to shorten. To investigate how the proposed approach works on different categories of terms, we requested three graduates to manually classify abbreviations in the 700 selected method names into three categories: Proper

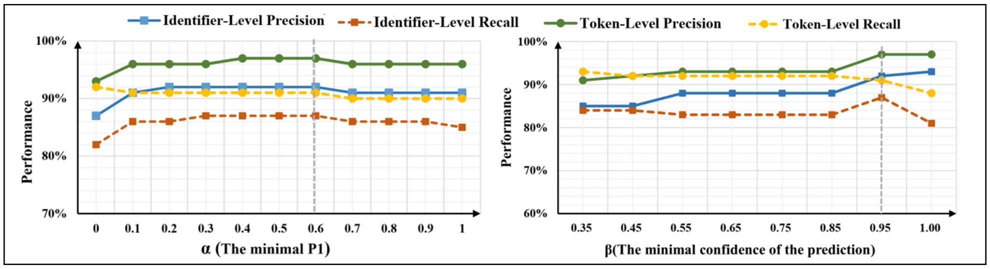

Fig. 3. Influence of Threshold Setting

Table 4. Performance on Different Categories of Terms

Categories	# Total	#Shortened Correctly	#Shortened Incorrectly	#Missed
Terminologies	182	157 (86.3%)	15 (8.2%)	10 (5.5%)
Proper nouns	0	0	0	0
Others	557	491 (88.2%)	34(6.1%)	34 (6.1%)

nouns, technical terms, and others. These three graduates are master-level students and are familiar with Java. However, they are not aware of the proposed approach. `Terminologies` (like "IP" and "XML") refer to "the set of technical words or expressions used in a particular subject" [2]. `Proper Nouns` include "personal names, place names, names of companies and organizations, and the titles of books, films, songs, and other media" [3]. `Others` refer to all abbreviations except for proper nouns and terminologies. All the abbreviations were discussed until they reached a consensus.

The results of the evaluation are presented in Table 4. The percentages in parentheses represent how many percentages of the given category of full terms have been shortened correctly, shortened incorrectly, or missed by the approach. For example, the 5.5% on the last column suggests that 5.5% (=10/182) of the terminologies have been missed by the approach. Note that the 700 method names do not contain any proper nouns, which may suggest that source code identifiers are substantially different from natural language documents. In contrast, the method names involve numerous terminologies. From Table 4, we observe that the proposed approach had similar performance on terminologies and non-terminologies. For example, around 86.3% of the terminologies had been shortened correctly, rather close to the ratio (88.2%) of "others". The most common non-terminology full terms (that should be shortened) include "message", "length", and "parameter", and the proposed approach worked well on them. The evaluation results may suggest that the proposed approach could be employed to shorten both terminologies and non-terminologies.

We also tested how sensitive the approach was to the length of the inputted names. We divided all of the tested 700 overlong method names into four equally sized subsets (i.e., Q1, Q2, Q3, and Q4) according to their length (in characters). Q4 was composed of the top 25% longest names whereas Q1 was composed of shortest ones. We evaluated `NameCompressor` on Q1, Q2, Q3, and Q4 independently and computed its performance on each of the subsets. Our evaluation results are presented in Table 5. From the table, we observe that the approach worked well regardless of the length of inputted names. For example, its performance on the longest method names in Q4 (containing 29-57 characters) is comparable to that on the shortest method names in Q1 (containing less than 17 characters): The identifier-level F_1 score on Q4 is exactly the same as that on Q1.

Table 5. Impact of Method Names' Length

Length	Identifier-Level			Token-Level		
	Precision	Recall	F_1	Precision	Recall	F_1
$Q1$	95%	89%	92%	97%	91%	94%
$Q2$	92%	82%	87%	94%	81%	87%
$Q3$	92%	89%	90%	96%	92%	94%
$Q4$	92%	89%	91%	98%	96%	97%

Finally, we investigated how sensitive the approach was to the domain of the applications. Note that the seven subject applications were from different domains and developed by different teams. Consequently, by investigating how the performance of the approach varies among the subject application, it may reveal how sensitive it is to application domains. From Table 1, we observe that the performance varies slightly across the subject applications. For example, the identifier-level F_1 score varies slightly between 87% and 93%. It could suggest that the proposed approach works well regardless of application domains.

Based on the preceding analysis, we conclude that the proposed approach works well regardless of the types, length, and domain of the inputted method names.

4.7 RQ6: User Study

`NameCompressor` has achieved high effectiveness in shortening long method. However, the current evaluation only focuses on measuring the correctness of the abbreviation process. It does not investigate whether developers appreciate method names with abbreviations. To this end, in this section, we conduct an empirical study with developers to investigate this issue.

We randomly selected 220 items (noted as *namePair*) from the benchmark `abbInMethodName`, and each item contains an overlong method name as well as its corresponding shortened name. After that, we invited ten experienced developers to manually check each selected item, indicating their preference between the short and the long versions. All developers had more than five years of programming and Java experience. Notably, they were different from the developers involved in Section 4.1.2. Besides the preference of method names, we also planned to collect the rationale for their preference. To this end, we asked the participants to explain why the selected names were preferred. Noting that the rationale could be highly similar (even identical) in most cases, we decided to collect potential answers with a pilot survey and provided such potential answers as options in the following full-scale survey. In the pilot survey, we selected 20 items from *namePair*, and each participant was randomly assigned 5 items. In the full-scale survey, the remaining items in *namePair* were assigned to the ten participants where each participant was randomly assigned 50 items. Notably, in the full-scale survey, participants may select any of the potential options (to explain the rationale for their preference) or type in their own explanation (answer) from scratch. The questionnairs and responses are available at [4].

The user study suggests that short names were frequently preferred. Among the collected 550=10*(5+50) choices, 501 (account for 91%) prefers short names. On 180 of the 220 sampled items, all participants unanimously preferred the short names containing abbreviations to their corresponding lengthy names. However, we also notice that on 3 out of the 220 cases, all participants unanimously preferred the long names. A typical example is the full name *floatingPointPattern*. Its corresponding short name in the benchmark is "*fpPattern*" where "*fp*" is difficult to interpret. Although *floating point* is not rare, developers usually use built-in data types (e.g., *float* and *double*) to represent floating points. As a result, most developers, including those in the user study, are not familiar with the abbreviation "*fp*", which increases the difficulty in understanding the abbreviation. The long name, i.e., "*floatingPointPattern*", was preferred because it is easy to read and its length is acceptable.

According to the survey, we observe that the major reasons for preferring short names include "*use of widely recognized abbreviations that do not hinder the understanding of the source code* (41.6%)", "*consistency with the naming conventions of identifiers within the method body*" (accounting for 33.6%), and "*excessively lengthy method names significantly impact code readability* (24.7%)". The major reasons for preferring long names include "*method names with full terms are not excessively long and full terms tend to be more readable*" (83.3%) and "*less common abbreviation influences the understanding of the source code*" (16.7%).

4.8 Threats to Validity

The first threat to external validity is that the testing dataset contains only 700 method names. The conclusions drawn on such a small dataset may not be generalizable. However, enlarging the dataset is challenging because constructing such a testing dataset is too labor expensive. To reduce the threat, we randomly sampled 100 method names from each of the 7 subject applications, and all the applications were from different domains and were developed/maintained by different teams. Besides, the answers to RQ5 also help reduce this threat. Investigation results of RQ5 in Section 4.6 suggest that the proposed approach works well on different categories of method names. Therefore, potential bias in data sampling may not significantly influence the performance of the proposed approach.

Another threat to external validity is that the evaluation was confined to Java, and thus the conclusions of the evaluation may not be generalizable to other programming languages. Although the approach itself is not confined to Java, the current prototype implementation can only parse Java methods, and thus the evaluation was confined to Java source code.

A threat to the construct validity is that the manual expansion involved in the evaluation could be inaccurate. The quality of the expansion had a substantial influence on the accuracy of the performance assessment. To reduce the threats, we requested two experienced software engineers (with more than five years of Java experience) to expand the abbreviations independently, and they resulted in a high Kappa coefficient [17] of 0.99. We also discarded items that failed to expand or reach a consensus.

Another threat to the construct validity is that the manual ranking in the user study could be subjective and inaccurate. To avoid potential bias, we invited non-authors of the involved source code for the user study. However, lacking a full understanding of the involved source code and background of software domains, the participants may fail to identify the optimal names for the given software entities. To reduce the threat, we requested multiple participants to rank each of the samples, and a high Kappa 0.78 suggests they frequently resulted in consistent ranking.

5 DISCUSSION

The learning-based compression in Section 3.4 does not yet directly exploit the substring of the identifiers. Although such substrings could be useful for identifying possible abbreviations, exploiting them could complicate the learning model by feeding it with plain text because they would result in complex and lengthy representations (i.e., digital vectors). As a result, the mapping from the model's input to its output could be complex, and the model training requires much more training data that is not yet available. In the future, however, with more high-quality training data and more powerful learning models, we may explicitly exploit the substrings.

It can be hard to establish a one-to-one correspondence between abbreviations and full terms, resulting in multiple full-term candidates for a single abbreviation. To minimize the risk, the context-aware compression only leverages <*abbreviation*, *full term*> pairs that have already been leveraged in the enclosing file. That is, it only uses abbreviations that have been used by developers in the same file. Such abbreviations are likely unambiguous within the given contexts. However, probability-based compression and learning-based compression may introduce ambiguous abbreviations because they lack such constraints.

`NameCompressor` is specially designed to shorten overlong method names although it has the potential to be applied to other identifiers with essential adaption. Notably, different types of identifiers may have different length distributions, and thus the learning-based model proposed in Section 3.4 should be specific to a given type of identifier, and different types of identifiers should have their unique model. Besides that, context-aware compression should be explicitly adapted for different types of identifiers because they may have different contexts. For example, the context of a method name is defined by its enclosing class. In contrast, the context of class names should be larger, containing different classes (documents). Although such adaption is indispensable, the rationale behind the proposed approach is generic and should apply to different types of identifiers.

6 CONCLUSIONS AND FUTURE WORK

Proper usage of abbreviations in identifiers, like method names, is critical for the readability and maintainability of source code. However, manual abbreviations are susceptible to mistakes, especially for inexperienced developers or those unfamiliar with English. Note that simply replacing full terms with abbreviations from abbreviation dictionaries may not work because the usage of abbreviations is often context-aware and the same full term appearing in different identifiers may be replaced by different abbreviations. To facilitate the task, we propose a novel approach to shortening overlong method names with abbreviations. Our evaluation results on open-source applications suggest that the proposed approach is accurate, and it has a great chance (93% on average) that the suggested method names are identical to those coined manually by the original developers.

`NameCompressor` is designed to deduce abbreviations based on the naming convention of method names. Note that naming conventions vary across different types of identifiers. For example, method names often adopt verb phrases, while variable and class names often adopt noun phrases. Consequently, the trained model *NBM* and the abbreviation dictionary *abbDict* for method names are not readily applicable to other types of identifiers. Nevertheless, the presented methodology can be adapted to support other identifier types by retraining the model *NBM* with appropriate training data and rebuilding the abbreviation dictionary *abbDict* with abbreviations from given types of identifiers. Another limitation of `NameCompressor` is that it does not drop redundant or unimportant words in an overlong name when performing shortening. We plan to address this limitation and adapt `NameCompressor` to support other identifier types in future work.

7 DATA AVAILABILITY

The prototype implementation and the replication package are publicly available at GitHub [4] to facilitate third-party evaluation. Detailed construction on how to replicate the evaluation is also publicly available.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive suggestions. The work is partially supported by the National Natural Science Foundation of China (62232003 and 62172037), the China Postdoctoral Science Foundation (2023M740078 and BX20240008), and Hong Kong SAR RGC/GRF #16205821.

REFERENCES

[1] 2023. BernoulliNB. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html.
[2] 2023. Oxford Learners Dictionaries. https://www.oxfordlearnersdictionaries.com/.
[3] 2023. Proper Nouns. https://en.wikipedia.org/wiki/Proper_noun.
[4] 2023. Replication Package for NameCompressor. https://github.com/jiangyanjie/NameCompressorTool.
[5] 2024. Flink. https://github.com/apache/flink.
[6] 2024. SpringBoot. https://github.com/spring-projects/spring-boot.

[7] Eytan Adar. 2004. SaRAD: A Simple and Robust Abbreviation Dictionary. *Bioinformatics* 20, 4 (2004), 527–533.

[8] Eran Avidan and Dror G Feitelson. 2017. Effects of Variable Names on Comprehension: An Empirical Study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 55–65.

[9] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G Feitelson. 2017. Meaningful identifier names: The case of single-letter variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 45–54.

[10] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. 2009. Identifier Length and Limited Programmer Memory. *Science of Computer Programming* 74, 7 (2009), 430–445.

[11] Bruno Caprile and Paolo Tonella. 2000. Restructuring Program Identifier Names.. In *ICSM*. 97–107.

[12] Nuno Ramos Carvalho, José João Almeida, Pedro Rangel Henriques, and Maria João Varanda. 2015. From Source Code Identifiers to Natural Language Terms. *Journal of Systems and Software* 100 (2015), 117–128.

[13] Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2012. LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 233–242.

[14] Florian Deissenboeck and Markus Pizka. 2006. Concise and Consistent Naming. *Software Quality Journal* 14, 3 (2006), 261–282.

[15] Henry Feild, David Binkley, and Dawn Lawrie. 2006. An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06)*.

[16] Dror G Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2020. How developers choose names. *IEEE Transactions on Software Engineering* 48, 1 (2020), 37–52.

[17] Joseph L Fleiss. 1971. Measuring Nominal Scale Agreement among Many Raters. *Psychological bulletin* 76, 5 (1971), 378.

[18] Fan Ge and Li Kuang. 2021. Keywords Guided Method Name Generation. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 196–206.

[19] Emily Hill, Zachary P Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K Vijay-Shanker. 2008. AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. 79–88.

[20] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. 2017. Shorter Identifier Names Take Longer to Comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 217–227.

[21] Yanjie Jiang, Hui Liu, Jiahao Jin, and Lu Zhang. 2022. Automated Expansion of Abbreviations Based on Semantic Relation and Transfer Expansion. *IEEE Transactions on Software Engineering* 48, 2 (2022), 519–537.

[22] Yanjie Jiang, Hui Liu, and Lu Zhang. 2019. Semantic Relation based Expansion of Abbreviations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 131–141.

[23] Yanjie Jiang, Hui Liu, Yuxia Zhang, Nan Niu, Yuhai Zhao, and Lu Zhang. 2021. Which Abbreviations Should be Expanded?. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, August 23-28, 2021*. ACM, 578–589.

[24] Yanjie Jiang, Hui Liu, Jiaqi Zhu, and Lu Zhang. 2018. Automatic and Accurate Expansion of Abbreviations in Parameters. *IEEE Transactions on Software Engineering* 46, 7 (2018), 732–747.

[25] Dawn Lawrie and Dave Binkley. 2011. Expanding Identifiers to Normalize Source Code Vocabulary. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 113–122.

[26] Dawn Lawrie, Henry Feild, and David Binkley. 2007. Extracting Meaning from Aabbreviated Identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, 213–222.

[27] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 3–12.

[28] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering* 3, 4 (2007), 303–318.

[29] Guangjie Li, Hui Liu, Ge Li, Sijie Shen, and Hanlin Tang. 2020. LSTM-based Argument Recommendation for Non-API Methods. *Science China Information Sciences* 63, 9 (2020), 1–22.

[30] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 574–586.

[31] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2010. Recognizing Words from Source Code Identifiers using Speech Recognition Techniques. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 68–77.

[32] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2023. Automated Variable Renaming: Are We There Yet? *Empirical Software Engineering* 28, 2 (2023), 45.

[33] Christian D Newman, Michael J Decker, Reem S Alsuhaibani, Anthony Peruma, Dishant Kaushik, and Emily Hill. 2019. An Empirical Study of Abbreviations and Expansions in Software Artifacts. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 269–279.

[34] Zhiheng Qu, Yi Hu, Jianhui Zeng, Bo Cai, and Shun Yang. 2022. Method Name Generation Based on Code Structure Guidance. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1101–1110.

[35] V. Rajlich and N. Wilde. 2002. The Role of Concepts in Program Comprehension. In *Proceedings 10th International Workshop on Program Comprehension*. 271–278.

[36] Giuseppe Scanniello and Michele Risi. 2013. Dealing with faults in source code: Abbreviated vs. full-word identifier names. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 190–199.

[37] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 31–3109.

[38] Porfirio Tramontana, Michele Risi, and Giuseppe Scanniello. 2014. Studying Abbreviated vs. Full-Word Identifier Names when Dealing with Faults: An External Replication. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–1.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017,Long Beach, CA, USA*. 5998–6008.

[40] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 38–45.

[41] Jingxuan Zhang, Siyuan Liu, Lina Gong, Haoxiang Zhang, Zhiqiu Huang, and He Jiang. 2023. BEQAIN: An Effective and Efficient Identifier Normalization Approach With BERT and the Question Answering System. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2597–2620.

[42] Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An Accurate Identifier Renaming Prediction and Suggestion Approach. *ACM Transactions on Software Engineering and Methodology* (2023).

[43] Jianbing Zhang, Yixin Sun, Shujian Huang, Cam-Tu Nguyen, Xiaoliang Wang, Xinyu Dai, Jiajun Chen, and Yang Yu. 2017. AGRA: An Analysis-Generation-Ranking Framework for Automatic Abbreviation from Paper Titles.. In *IJCAI*. 4221–4227.

[44] Daniel Zugner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Gunnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *International Conference on Learning Representations (ICLR)*.