Which Abbreviations Should Be Expanded?

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

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
Liuhui08@bit.edu.cn

Yuxia Zhang
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Nan Niu
Department of Electrical Engineering and Computer Science, University of Cincinnati
Cincinnati, USA
nan.niu@uc.edu

Yuhai Zhao
School of Computer Science and Engineering, Northeastern University
Shenyang, China
zhaoyuhai@mail.neu.edu.cn

Lu Zhang
Key Laboratory of High Confidence Software Technologies, Peking University
Beijing, China
zhanglu@sei.pku.edu.cn

ABSTRACT

Abbreviations are common in source code. Properly designed abbreviations may significantly facilitate typing, typesetting, and reading of lengthy source code. However, abbreviations, if used improperly, may also significantly reduce the readability and maintainability of source code. Although a few automated approaches have been proposed to suggest full terms for given abbreviations, to the best of our knowledge, there is no automated approaches to suggest whether abbreviations are used properly, i.e., whether they should be replaced with corresponding full terms. Notably, it is often challenging for inexperienced developers and maintainers to make such decisions. To this end, in this paper, we propose an automated approach to assisting developers and maintainers in making the decisions. The rationale of the approach is that abbreviations should not be expanded if the expansion would result in unacceptably lengthy identifiers or if developers/maintainers can easily figure out the meaning (full terms) of the abbreviations based on their domain knowledge or contexts of the abbreviations. From a corpus of programs, we leverage data mining techniques to discover common abbreviations that are frequently employed by various developers in similar contexts. The key of the data mining is to turn the problem of mining common abbreviations into the *maximal clique problem* that has been extensively studied. We suggest to not expand given abbreviation if it matches at least one of the discovered common abbreviations. From the same corpus, we also calculate the probability distribution for the length of different types of identifier, e.g., variable names and method names. The probability distribution specifies how likely an identifier of type T is composed of exactly *n* characters. Our heuristic is to not expand the abbreviation if the probability of its enclosing identifier would be reduced by the expansion. Finally, we also suggest to not expand the abbreviation if its full terms are contained in surrounding contexts of the abbreviation, i.e., tokens on the same source code line. Other abbreviations that do not receive suggestions from the proposed approach are expected to be replaced with their full terms. Our evaluation results on 1,818 abbreviations from five open-source applications suggest that the proposed approach is accurate with a high accuracy of 95%.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software development techniques**.

KEYWORDS

Abbreviation, Expansion, Data Mining, Cliques, Software Quality

ACM Reference Format:
Yanjie Jiang, Hui Liu, Yuxia Zhang, Nan Niu, Yuhai Zhao, and Lu Zhang. 2021. Which Abbreviations Should Be Expanded?. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468616

1 INTRODUCTION

Identifiers, i.e., names of software entities, are common in source code. They account for the majority (70%) of source code in terms of characters [21]. Consequently, such identifiers composed of natural language terms serve as the major source for software comprehension [16, 17], and thus the importance of qualified identifiers is well recognized [25, 44]. For example, Avidan et al. [13] empirically investigated the effect of variable names on software comprehension, and their empirical results suggest that meaningful variable names are instrumental for comprehension, and can effectively serve as documentation for source code.

Abbreviations are widely employed to shorten identifiers [39, 41]. Developers often use a short abbreviation to replace a long term or a sequence of terms in identifiers. For example, they often use "e" to

*corresponding author

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468616

represent "exception", and prefer "XMLParser" to "ExtensibleMarkupLanguageParser". Properly designed abbreviations may significantly facilitate typing, typesetting, and reading of lengthy source code. For example, the lengthy class name "ExtensibleMarkupLanguageParser" makes it challenging to typeset the following simple variable declaration: "`public ExtensibleMarkupLanguageParser m_extensibleMarkupLanguageParser = new ExtensibleMarkupLanguageParser(ExtensibleMarkupLanguageParser.DEFAULT);`" because it is likely that the whole statement could not be shown entirely within a single line (depending on the size of monitors). The length can be significantly reduced by replacing "ExtensibleMarkupLanguage" with its well-known abbreviation "XML", and thus the lengthy statement becomes much shorter: "`public XMLParser m_XMLParser = new XMLParser(XML-Parser.DEFAULT);`". The latter is easier to type in and has a greater chance to be shown entirely within a single line, which facilitates program comprehension.

However, abbreviations, if used improperly, may also significantly reduce the readability and maintainability of source code [30, 38]. Abbreviations "s" (standing for "students") and "ds" (standing for "data sequence") are good examples of improper usage of abbreviations. It could be difficult for developers other than the authors to figure out the exact meaning of the abbreviations, which may result in misunderstanding and incorrect usage of the enclosing programs. To this end, a few automated novel approaches have been proposed to suggest full terms for given abbreviations [18, 29, 33]. Developers may replace the abbreviations with full terms suggested by such tools by renaming [42, 43]. Notably, it is highly beneficial for authors of source code to replace improperly used confusing abbreviations with full terms. The authors' replacement is highly accurate and is done once and for all. Although readers/maintainers of the source code may leverage expansion tools to guess the meaning of the confusing abbreviations, the guess could be inaccurate and it should be repeated by every reader/maintainer.

We conclude based on the preceding analysis that carefully designed abbreviations facilitate coding and improve readability of source code whereas improperly used abbreviations could reduce readability of source code and result in incorrect usages/modification of programs. Although novel approaches have been proposed to suggest full terms of given abbreviations, to the best of our knowledge, there is no automated approaches to suggest whether abbreviations require expansion, i.e., whether they should be replaced with corresponding full terms. Notably, it is often challenging for inexperienced developers and maintainers to make such decisions because there is no quantitative guidance for the decision, and it fully depends on developers' experience and intuitions.

To this end, in this paper, we propose an automated approach (called *SmartExpander*) to deciding whether a given abbreviation needs to be expanded at all. The rationale of the approach is that abbreviations should not be expanded if the expansion would result in lengthy identifiers or if developers/maintainers can easily figure out the meaning (full terms) of the abbreviations based on their domain knowledge or contexts of the abbreviations. Consequently, we design a sequence of heuristics according to the rationale to pick up such abbreviations that do not require expansion. From a corpus of source code, we leverage data mining techniques to discover common abbreviations that are frequently employed by different developers in similar contexts. The key of the data mining is to turn the problem of mining common abbreviations into the maximal clique problem [23] that has been extensively studied [15, 47]. The proposed approach suggests to not expand an abbreviation if it matches at least one of the discovered common abbreviations. From the same corpus of open-source programs, we also build the probability distribution model for the length of different types of identifier, e.g., variable names, method names, and class names. The model specifies how likely an identifier of type T is composed of exactly n characters, noted as $P(T, n)$. Suppose that a given abbreviation comes from identifier *id* of type T, and replacing the abbreviation with its full terms would increase the length of the identifier from k to j characters. Our approach suggests to not expand the abbreviation if $P(T, j) < P(T, k)$. Finally, the proposed approach suggests to not expand the abbreviation if its full terms are contained in the surrounding contexts of the abbreviation, i.e., tokens on the same source code line. Other abbreviations that do not receive suggestions from the proposed approach are expected to be replaced with their full terms.

The proposed approach has been evaluated on open-source applications and our evaluation results suggest that the proposed approach is accurate. We randomly sampled 1,818 abbreviations from five open-source applications, and manually decided whether they should be expanded. We applied the proposed approach to the sampled abbreviations, and compared its suggestions against manual decisions. Our evaluation results suggest that the accuracy of the proposed approach varies from 93% to 96%, with an average of 95%.

The paper makes the following contributions:

- An automated approach to deciding which abbreviations should (or should not) be expanded. To the best of our knowledge, it is the first automated approach for this task.
- An extensive evaluation of the proposed approach on open-source applications. The evaluation results suggest that the proposed approach is accurate.

The rest of the paper is structured as follows. Section 2 presents a short review of related research. Section 3 proposes the approach to suggest whether abbreviations are used properly. Section 4 presents an evaluation of the proposed approach on well-known open-source applications. Section 5 provides conclusions and potential future work.

2 RELATED WORK

2.1 Expansion of Abbreviations in Source Code

Because of the popularity of abbreviations, a large number of automated approaches [33] have been proposed to suggesting full terms for abbreviations in source code, especially in identifiers. An intuitive way to expanding abbreviations is to compare abbreviations against generic English dictionaries and return dictionary words that match the abbreviations according to predefined rules [22]. For example, an abbreviation matches a dictionary word if the abbreviation is a prefix of the dictionary word. The advantage of looking up expansion in generic English dictionaries is twofold. First, such dictionaries are ready for reuse. Second, such dictionaries contain almost all possible English words, and thus in most cases we can find matching terms for given abbreviations. However, it is quite

often that for a given abbreviation there are a large number of matching terms from a generic English dictionary. It remains challenging to choose the correct one from such a large number of matching terms.

Another intuitive way to abbreviation expansion is to look up abbreviation dictionaries [10]. An abbreviation dictionary contains a list of well-known abbreviations as well as their corresponding full terms. By looking up the dictionary, we can retrieve the full terms for a given abbreviation accurately. However, such abbreviation dictionaries are often constructed manually, which significantly limits the size of such dictionaries [16]. As a result, matching algorithms may fail frequently to retrieve matching items from such dictionaries [26] in case the dictionaries do not contain the abbreviations. Another problem with abbreviation dictionaries is that the same abbreviation may have different meaning (and thus different full terms) depending on its contexts. As a result, looking up the abbreviation dictionaries alone may fail to select the correct expansion.

Leveraging the contexts of abbreviations (e.g., surrounding source code and comments) could significantly increase the accuracy of abbreviation expansion [33]. For example, Corazza et al. [20], Lawrie et al. [37] and Guerrouj et al. [45] suggested to search 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. However, developers rarely write comments, which significantly reduces the chance of finding full terms in comments. Caprile et al. [16] and Carvalho et al. [18] suggested to expand abbreviations by looking for full terms from surrounding source code. Lawrie et al. [35] suggested to match a given abbreviation against full terms in the enclosing methods only. Hill et al. [29] proposed a complex approach to search for full terms in enclosing methods, enclosing classes, and enclosing projects in order. Jiang et al. [34] proposed a parameter-specific approach to expand parameter abbreviations only by leveraging the relationship between arguments and parameter: If an abbreviation comes from an argument, they look for its full terms in the corresponding parameter, and vice versa. Later, they [32, 33] generalized the approach to leverage various semantic relationships among software entities, e.g., assignments between software entities, inclusion relation between software entities, invocation of methods, and access of fields/variables. 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. [46] conducted an empirical study to analyze 861 abbreviation-expansion pairs extracted from five open-source applications. One of their interesting findings is that documents of programming languages, project documents, and source code contain similar numbers of expansions whereas documents of programming languages are the primary source of unique expansions.

Besides the source of full terms, matching algorithms serve as another corner stone for abbreviation expansion [33]. For a given abbreviation and given source of full terms (e.g., dictionaries and code comments), abbreviation expansion tools should leverage matching algorithms to search for potential expansions from the given source (sequences of words). Apostolio et al. [12] proposed a matching algorithm where a word is taken as a potential expansion of an abbreviation if the abbreviation is a subsequence of the word. Lawrie et al. [36] make suggestions only if the given abbreviation has a single potential expansion, and ignores the cases where multiple potential expansions are retrieved. In contrast, Hill et al. [29] and Carvalho et al. [18] sort such potential expansions based on their frequency, and recommend the top one. Lawrie et al. [35] choose the one that has the highest lexical similarity with the given abbreviation. Guerrouj et al. [27] and Corazza et al. [20] employ graph-based matching algorithms to search for the most likely expansions.

Such novel approaches to abbreviation expansion have significantly facilitated program comprehension. However, none of them could be leveraged to suggest which abbreviations should (or should not) be expanded, as our approach does.

2.2 Influence of Abbreviations

Abbreviations are common in source code, and thus their negative and positive effect is a big concern for software developers. To this end, researchers have conducted empirical studies to investigate the negative/positive effect of such abbreviations. For example, Lawrie et al. [38, 39] consulted over 100 programmers and concluded that identifiers made up of full words often lead to higher software quality. However, they also suggested that in many 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. Swidan et al. [50] analyzed variables in Scratch, a popular block-based language aiming at children. Their analysis results suggest that Scratch programmers often prefer longer identifier names than developers in other languages. Hofmeister et al. [30] conducted an experimental study with 72 professional C# developers, requesting them to locate defects in source code snippets. Their evaluation results suggest that full term identifiers can speed up fault localization by 19% compared to meaningless single letters and abbreviations. Notably, single letters and abbreviations are roughly equivalent with regard to fault localization. Schankin et al. [49] requested 88 Java developers to locate semantic defects in source code snippets, and their evaluation results suggest that longer and more descriptive identifiers can speed up fault localization by 14% compared to shorter and less descriptive identifiers. However, this effect disappears when developers are searching for syntax errors where in-depth understanding of the code is often not required.

Although empirical studies introduced in the preceding paragraph suggest that longer identifiers often lead to better readability of source code, their negative impact is not negligible. For example, Binkley et al. [14] conducted a case study to investigate the balance between longer, more expressive names and limited programmer memory resources. Their evaluation results suggest that longer names often take more time to process and reduce correctness in software engineering tasks. Scanniello et al. [48, 51] conducted a controlled experiment where students were asked to find and fix faults in one of two alternative versions of the same program: One with full-termed 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).

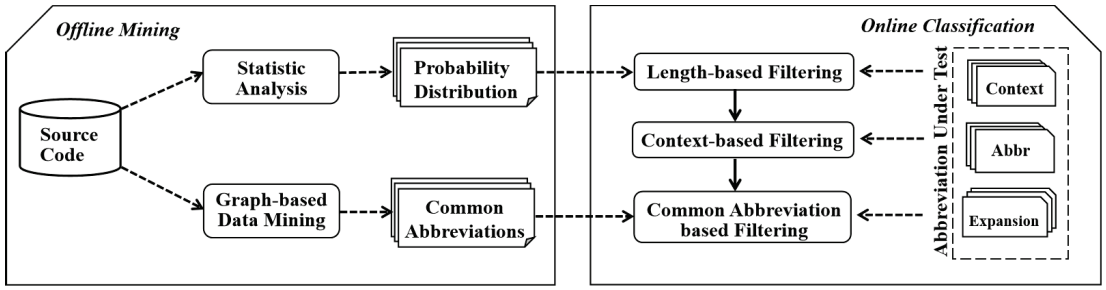

Figure 1: Overview

Such interesting findings reveal both positive and negative sides of abbreviations, which inspired the work presented in this paper.

3 APPROACH

In this section, we present an automated approach to assisting developers and maintainers in deciding which abbreviations should (or should not) be expanded. An overview is presented in Section 3.1, and details are presented in the following sections.

3.1 Overview

Fig.1 presents the overview of the proposed approach. The proposed approach is divided into two phases: offline mining (the left part of Fig.1) and online classification (the right part of Fig.1). During the offline mining phase, the proposed approach learns from a corpus of source code and discovers a list of common abbreviations as well as the probability distribution of identifiers' length. During the online classification phase, the proposed approach applies a sequence of heuristic-based filtering to determine whether a given abbreviation should be expanded. Overall, the proposed approach works as follows:

- From a corpus of source code, we extract all identifiers (i.e., names of software entities) and classify them according to entities' types (e.g., variable names, method names, and class names). For each type of identifiers, we calculate the probability distribution of their length. The probability distribution specifies how likely an identifier of type T is composed of exactly n characters, noted as $P(T, n)$.
- From the same corpus, we extract all abbreviations. Lexically-identical abbreviations regardless where they are extracted are presented on a single graph where each abbreviation is represented as a node and weight of edges represent the context similarity among lexically-identical abbreviations. In case there is a large subgraph where each pair of nodes is connected with a short edge, the subgraph (called *maximal clique*) represents a common abbreviation that is widely used in similar contexts.
- For a given abbreviation *abb* from identifier *id* of type T, replacing the abbreviation with its full terms would increase the length of *id* from k to j. We suggest to not expand the abbreviation if $P(T, j) < P(T, k)$ (noted as Case 1).
- If the abbreviation *abb* and its full terms appear on the same source code line where the enclosing identifier is defined (noted as Case 2), we suggest to not expand it.

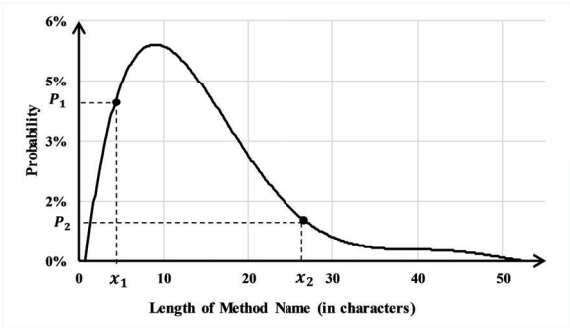

Figure 2: Probability Distribution of Identifiers' Length

- If there are a large number of abbreviations in the same project that are lexically identical to *abb* (noted as Case 3), we suggest to not expand the abbreviation.
- If there is a maximal clique whose abbreviations are lexically identical to *abb*, and the average context similarity between *abb* and nodes in the clique is greater than threshold β (noted as Case 4), we suggest to not expand the abbreviation.
- We suggest to expand *abb* if it does not belong to any of the preceding cases (i.e., Cases 1-4)

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

3.2 Statistic Analysis on Identifiers' Length

Lengthy identifiers have negative effect on the readability of source code [14]. Consequently, if expanding abbreviations results in lengthy identifiers, we should suggest to not expand them.

From a corpus of source code, we extract all identifiers, including variable names, parameter names, method names, class names, and field names. Notably, we extract all identifiers regardless of whether they contain abbreviations. The resulting identifiers are divided into five types: variable names (including field names), parameter names, method names, class names, and iteration indexes. Iteration indexes require a special category because such indexes (e.g., i and j) are often significantly shorter than other variables. For each type of identifiers, we calculate the probability distribution of their length. The probability distribution specifies how likely an identifier of type T is composed of exactly n characters, noted as $P(T, n)$. For example, the probability distribution for method names is presented in Fig.2 where the horizontal axis presents the length of method

names and vertical axis presents the probability for the length of a method name being equivalent to the horizontal axis.

Suppose that the abbreviation under test (noted as *abb*) comes from identifier *id* of type T, and replacing the abbreviation with its full terms would increase the length of *id* from x_1 to x_2. We look up the probability distribution model of type T (e.g., the one in Fig.2), and retrieve the corresponding probabilities for length x_1 and x_2, respectively:

$$p_1 = P(T, x_1) \quad (1)$$

$$p_2 = P(T, x_2) \quad (2)$$

We suggest to not expand the abbreviation *abb* in identifier *id* if $P(T, x_2) < P(T, x_1)$. The rationale of the heuristic is that the length of the identifier becomes less acceptable (i.e., less popular) because of the abbreviation expansion, and thus we had better not conduct the expansion. In this case, the proposed approach terminates the process on the given abbreviation. Otherwise, the proposed approach would apply other heuristics (as specified in the following sections) to the abbreviation to reach the final decision.

3.3 Graph-Based Data Mining

Well-known abbreviations (like "XML" and "AI") are less likely to hinder source code comprehension because developers and maintainers could easily figure out their meaning (and full terms). To this end, we would suggest to not expand such well-known abbreviations. A simple and intuitive way to tell whether a given abbreviation is well-known is to look up abbreviation dictionaries. However, such abbreviation dictionaries are often constructed manually and thus their sizes are limited [33]. As a result, some popular abbreviations, especially those that became popular recently, may not be included in such manually constructed abbreviation dictionaries.

To this end, in this paper, we leverage graph-based data mining techniques [23] to identify well-known abbreviations from a large corpus of source code. We extract all abbreviations from the corpus. Each of the resulting abbreviation is represented as a tuple:

$$abb_i =< str_i, cxt_i > \quad (3)$$

where str_i and cxt_i are the text and context of the abbreviation, respectively. The context is composed of a sequence of identifiers that appear in the enclosing document (e.g., a Java file) of the abbreviation in the same order as they appear in the document. Such identifiers are partitioned into tokens (including abbreviations) by the heuristics proposed by Jiang et al. [34]. The context is finally represented as a sequence of tokens:

$$cxt_i =< t_1, \ldots, t_n > \quad (4)$$

We represent the context as a fixed-length numerical vector with well-known Paragraph2vector [40].

$$V(cxt_i) = P2V(< t_1, \ldots, t_n >) \quad (5)$$

where $V(cxt_i)$ is the numerical vector and $P2V$ is the mapping from a sequence of tokens into a numerical vector, i.e., Paragraph2vector. Notably, Paragraph2vector [40] is a deep learning-based algorithm that is widely used to convert short texts in natural languages into fixed-length numerical vectors. A significant advantage of Paragraph2vector is that semantically related texts could result in similar vectors, and thus the similarity of the resulting vectors could be leveraged to represent the semantic similarity of the original texts. Consequently, we compute the context similarity between two abbreviations based on the cosine similarity [3] of the resulting vectors:

$$Sim_{cxt}(abb_i, abb_j) = Sim(V(cxt_i), V(cxt_j)) = \frac{V(cxt_i) \cdot V(cxt_j)}{\|V(cxt_i)\| \left\|V(cxt_j)\right\|} \quad (6)$$

Abbreviations from the corpus are partitioned into different groups according to the texts (i.e., str_i in Equation 3). Abbreviations within the same group are lexically identical, and thus we call them *lexically-identical* abbreviations. For each group of the abbreviations, we construct an undirected graph G where nodes represent abbreviations in the group and weight of edges represent the context similarity between abbreviations (as defined in Equation 6).

Graph-based data mining of the proposed approach is to identify common abbreviations that are frequently used by different developers in similar contexts. We convert this task into a well-studied *maximal clique problem* [23] as follows. A *clique* of a graph is a subgraph where any two of the vertices are adjacent. Two vertices in a undirected graph are *adjacent* if and only if they are connected by an edge in the graph. A *maximal clique* is a clique that cannot be extended by including one more adjacent vertex, i.e., it is not a subset of a larger clique. The task to find out the maximal clique of a given graph is called *maximal clique problem* [23]. This problem has been well-studied, and a large number of efficient approaches have been proposed [24]. Notably, in the original graph G, each pair of vertices are *adjacent* no matter how similar (or dissimilar) they are. Consequently, the whole graph itself is the only maximal clique. To mine common abbreviations used in highly similar contexts, we remove the edge between two vertices if the context similarity (as defined in Equation 6) of two abbreviations (represented by the two vertices) is smaller than a predefined threshold α. As a result, a maximal clique of the resulting graph (G') represents a popular abbreviation that is employed frequently in highly similar contexts, and the size of the clique indicates its popularity. A maximal clique could be represented as a triple:

$$clq_i =< TxT(clq_i), V(clq_i), E(clq_i) > \quad (7)$$

where clq_i is a maximal clique, and $TxT(clq_i)$ is the text of the abbreviations represented by the vertices on the clique. $V(clq_i)$ and $E(clq_i)$ are vertices and edges on the clique. Notably, to remove less popular abbreviations, we only keep such maximal cliques whose size (number of vertex) is no less threshold β.

3.4 Common-Abbreviation Based Filtering

In the preceding section, we identify common abbreviations by applying data mining techniques to a corpus of source code, resulting in a set of maximal cliques. In this section, we compare abbreviations under test against such common abbreviations, to identify such abbreviations that are well-known to developers and maintainers.

Suppose that the abbreviation under test is $abb_i =< str_i, cxt_i >$. From the maximal cliques generated in the preceding section, we retrieve such cliques where abbreviations are lexically identical

to abb_i. For each of the resulting cliques we compute the average context similarity between abb_i and abbreviations on the clique:

$$avgSim(abb_i, clq_j) = \frac{\sum_{abb_k \in V(clq_j)} Sim_{cxt}(abb_i, abb_k)}{|V(clq_j)|}, \quad (8)$$

where $V(clq_j)$ (as specified in Equation 7) is the abbreviations (vertices) on the clique clq_j, and Sim_{cxt} computes the context similarity between a pair of abbreviations as specified in Equation 6. If the resulting average similarity $avgSim(abb_i, clq_j)$ is greater than α, we suggest to not expand the abbreviation abb_i, and the proposed approach terminates the processing on this abbreviation.

Some abbreviations, e.g, "PKCS" standing for "Public Key Cryptograph Standards", are domain-specific, and thus they may be missed by the graph-based data mining on large-scale source code as specified in the preceding section. To this end, we compare the abbreviation under test (noted as abb_i) against abbreviations within the same project to determine whether the same abbreviation is frequently used in different places within the same project. Consequently, we retrieve all abbreviations within the enclosing project of abb_i, and calculate the number of abbreviations that are lexically-identical to abb_i. If this number is greater than a threshold γ, we suggest to not expand the abbreviation.

3.5 Context-Based Filtering

Some abbreviations are easy to interpret/expand because their full terms appear in their surrounding contexts. Typical examples are presented in the following code snippet:

```
1  try {
2   BufferedReader bufReader = new BufferedReader(new FileReader(
        file));
3   String line;
4   while ((line = bufReader.readLine()) != null) {
5         if (!line.equals("")) {
6                int lenOfLine = line.length();
7                results.add(line + "," + lenOfLine);
8          }
9    }
10         bufReader.close();
11  }
12 catch (Exception e) {
13        e.printStackTrace();
14  }
```

This code snippet contains a few identifiers that are composed of abbreviations, i.e., "*bufReader*" on Line 2, "*lenOfLine*" on Line 6, and "*e*" on Line 12. None of the abbreviations, however, requires expansion because such abbreviations are easy to interpret. For the first one (i.e., "*bufReader*" on Line 2), we know its meaning according to its data type "BufferedReader"; For the second one (i.e., "*lenOfLine*" on Line 6), we can figure out its full term according to the right side of the assignment (i.e., "length()"); For the last one (i.e., "*e*" on Line 12), we know exactly what it means because of the data type ("Exception").

To identify such abbreviations that could be easily interpreted via their surrounding contexts, the proposed approach works as follows:

- First, for an abbreviation under test (noted as abb_i), we extract the whole line of source code where its enclosing identifier is defined as the context of the abbreviation, noted as $CTX(abb_i)$.
- Second, we decompose the context $CTX(abb_i)$ into a sequence of tokens according to blank spaces, capital letters, and special characters, e.g., "(" and ")". The resulting sequence is noted as $Seq(abb_i)$.
- Third, supposing the full terms of the abbreviation is a sequence of words $< w_1, \ldots, w_n >$, we suggest to not expand the abbreviation if all of the words have equivalent tokens in $Seq(abb_i)$. Two tokens (words) are equivalent if they are identical or share the same root. For example, "thread" and "threads" are not identical, but they do share the same root ("thread") and thus they are taken as equivalent. To this end, we apply stemming to such tokens with an open-source implementation from Stanford University [9].

Notably, if none of the heuristics specified in Sections 3.2-3.5 suggest to not expand a given abbreviation, the proposed approach by default would suggest to expand it.

4 EVALUATION

4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: Do all abbreviations in well-known open-source applications require expansion? If not, what percentage of the abbreviations should (or should not) be expanded?
- **RQ2**: Is the proposed approach (*SmartExpander*) accurate in identifying abbreviations that should (or should not) be expanded?
- **RQ3**: How does the setting of the thresholds employed by the proposed approach influence the performance of *SmartExpander*?
- **RQ4**: How does the heuristics influence the performance of *SmartExpander*?
- **RQ5**: Is *SmartExpander* scalable?

The proposed approach (called *SmartExpander*) is based on the assumption that not all abbreviations require expansion. Investigating **RQ1** would validate the assumption. **RQ2** concerns the performance (e.g., precision and recall) of SmartExpander in classifying abbreviations with regard to their necessity of expansion. **RQ3** concerns the setting of SmartExpander. As introduced in Section 3, SmartExpander leverages a set of thresholds, i.e., α (the minimal context similarity between abbreviations) in Section 3.3, β (the minimal size of the maximal cliques) in Section 3.3, and γ (the minimal number of lexically-identical abbreviations within the same project) in Section 3.4. Investigating **RQ3** helps developers and maintainers set the thresholds to maximize the performance of SmartExpander. **RQ4** concerns the effect of the employed heuristics. SmartExpander is essentially a sequence of heuristics: length-based heuristic (Section 3.2), common-abbreviation based heuristic (Section 3.4), and context-based heuristic (Section 3.5). By investigating **RQ4**, we may quantitatively reveal the effect of the employed heuristics. **RQ5** concerns the scalability of SmartExpander, i.e., whether SmartExpander can be applied efficiently to large projects.

4.2 Subject Applications

The evaluation is based on five well-known open-source applications. An overview of the subject applications is presented in Table 1. DavMail [4] is an exchange gateway allowing users to use any mail/-calendar client to communicate with an exchange server. Its goal is to provide standard compliant protocols in front of proprietary exchange. DocFetcher [5] is a desktop search application. It allows to search the contents of files on local computer. DrJava [6] is a

Table 1: Subject Applications

Application ID	Full Names	Domain	Major Developers	#Identifiers	#Abbreviations
Mail	DavMail Gateway	eMail	Mickaël Guessant, Oleksandr Huziy and André K	11,192	3,813
Doc	DocFetcher	Search Engine	Nam Quang Tran, Rob Crowe and bitnik	11,686	3,831
DrJ	DrJava	IDE	Corky Cartwright, Rebecca Smith, and Carolyn	66,099	26,105
Dubbo	Apache Dubbo	Distributed Computing	ken.lj, Ian Luo and Cvictory	60,277	20,897
jEdit	jEdit	Text Editor	Andrea Citti, Martin Raspe and Volker Friedritz	32,104	11,976
			Total:	181,358	66,622

Table 2: Not All Abbreviations Require Expansion

Applications	#Samples	#Positive	#Negative	Rate of Positive
Mail	346	71	275	21%
Doc	346	88	258	25%
DrJ	378	63	315	17%
Dubbo	377	52	325	14%
jEdit	371	75	296	20%
TOTAL	1,818	349	1,469	19%

lightweight programming environment for Java designed to foster test-driven software development. Apache Dubbo [1] is a high-performance, Java-based RPC framework. jEdit [7] is a programmer's text editor written in Java. These applications are selected because of the following reasons. First, all of them are well-known open-source applications, which facilitate replication of the evaluation. Second, they are from different domains, and developed by different programmers. Constructing such a comprehensive dataset may improve the generality of the conclusions drawn on the dataset.

Notably, SmartExpander requests a large corpus of high quality source code for data mining as specified in Section 3.2 and Section 3.3. To this end, we reuse the dataset created by Alon et al. [11] as the corpus of source code. The dataset is composed of source code from 1,000 top-ranked Java applications on GitHub. Such applications are from different domains and are developed by different teams. In total, the dataset contains 39,631,241 lines of source code, 13,571,072 identifiers, and 1,048,552 abbreviations. Notably, this corpus does not contain any of the five subject applications in Table 1.

4.3 Process

To answer RQ1, we sampled abbreviations from each of the involved subject applications and manually decide which of them require expansion. The size of the samples is determined by the number of abbreviations in the subject applications, and we computed the minimal size of the samples with *Sample Size Calculator* [8] with an error margin of 5% and confidence level of 95%. The resulting size is presented in Table 2. Notably, all of the samples were picked up randomly. For each of the resulting 1,818 abbreviations, we asked three developers to manually determine whether they should be expanded. Each of the three developers had three to five years of programming, and more than three years of Java experience. We requested them to expand the abbreviations and make decisions independently. They were allowed to access the context, but we did not give them training or designated time to get familiar with the subject systems. In case of inconsistence (in either expansions or necessity of expansion), we requested them to discuss together to reach an agreement. During the evaluation, each sample received exactly three labels from three participants. Following the paper by Hallgren [28], we computed Cohen's Kappa coefficient [19] for all coder pairs and used the arithmetic mean as the final Kappa. The Cohen's Kappa coefficient of their classification is up to 0.76, suggesting an excellent agreement between different participants. Fleiss' Kappa (0.75) also confirms the high agreement among participants. Notably, the resulting dataset, noted as *GoldenSet*, would serve as a benchmark in the following evaluation. Sample abbreviations in *GoldenSet* are classified into two categories. The first category, called *positive samples*, are composed of abbreviations that require expansion. The others belong to the second category, called *negative samples*.

To answer RQ2, we applied SmartExpander to the *GoldenSet*, and compared the suggestions generated by the approach against manual decisions. A generated suggestion on a given abbreviation is correct if and only if it is identical to the manual decision on the same abbreviation (provided by the *GoldenSet*). To measure the performance of SmartExpander, we computed classification metrics, i.e., accuracy, precision, and recall.

To answer RQ3, we changed the value of the thresholds and repeated the evaluation as specified in the preceding paragraph. Notably, we changed the value of a single threshold at a time to explicitly reveal the impact of each threshold. To answer RQ4, we disabled one of the heuristics at a time and evaluated the changes of the performance of SmartExpander on the same dataset (i.e., GoldenSet). Each of the heuristics was disabled once. To answer RQ5, we recorded the time SmartExpander took to make suggestions, and estimated the time complexity of SmartExpander.

4.4 RQ1: Not All Abbreviations Require Expansion

Table 2 presents the results of the manual labeling (classification) of the sampled abbreviations. The second column of the table presents the size of the sample, i.e., how many abbreviations have been picked up for manual labeling. The third column presents the number of positive abbreviations (among the samples) that request expansion and the fourth column presents the number of negative abbreviations that developers suggested to not expand. The last column presents the rate of positive, i.e., how many percentages of the samples are suggested to be expanded by the participants.

From the table, we make the following observations:

Table 3: Performance of SmartExpander

Applications	#Sample	#TP	#FP	#TN	#FN	Accuracy	Negative Samples		Positive Samples	
							Precision	Recall	Precision	Recall
DavMail	346	71	12	263	0	96%	100%	96%	86%	100%
DocFetcher	346	77	14	244	11	93%	96%	95%	85%	88%
DrJava	378	59	10	305	4	96%	99%	97%	86%	94%
Dubbo	377	44	14	311	8	94%	97%	96%	76%	85%
jEdit	371	67	12	284	8	95%	97%	96%	85%	89%
Total	1,818	318	62	1,407	31	95%	98%	96%	84%	91%

- First, not all of the abbreviations require expansion. The participants suggested to not expand more than eighty percentage of the manually analyzed abbreviations, i.e., 1,469 out of the 1,818 are manually classified as negative samples. The finding may suggest that most of the abbreviations in well-known applications are used properly, and are preferred to their full terms. This finding may also explain why abbreviations are common.
- Second, the rate of positive/negative is stable, varying slightly among subject applications. For example, the rate of negative varies slightly from 75% to 86%. This finding may suggest that the assumption (i.e., not all abbreviations require expansion) holds regardless of application domains and developers of software applications.

Although only 19%=349/1,818 of the sampled abbreviations should be expanded, considering the large number of abbreviations in source code (e.g., more than twenty thousand abbreviations in project Dubbo), there could be thousands of abbreviations in a single project that should be expanded. Furthermore, our research can be in turn motivated by the fact that the majority of abbreviations should not be expanded indeed: Developers should not expand all abbreviations with automated expansion tools. Instead, they should employ our approach to validate the necessity before expansion.

To identify what kind of abbreviations may not require expansion, we manually analyzed the 1,469 negative abbreviations. Our analysis results suggest that a significant part (35%=515/1,469) of the negative abbreviations are popular and well-known abbreviations. The most popular ones include "dir" ("directory"), "buf" ("buffer"), "msg" ("message"), "url" ("uniform resource locator"), and "pos" ("position"). Such abbreviations are well-known, and thus could be used safely without significantly negative effect in readability of source code. We also observe that 47%=691/1,469 of the negative abbreviations come from variables/parameters whose data types contain the corresponding full terms. A typical example is "FileInputStream fis" where "fis" stands for "file input stream".

From the preceding analysis, we conclude that not all abbreviations require expansion. Consequently, highly accurate approaches to assisting developers and maintainers in deciding which abbreviations should be expanded could be valuable.

4.5 RQ2: SmartExpander Is Accurate

To answer *RQ2*, we applied SmartExpander to the sample abbreviations from the subject applications. Evaluation results are presented in Table3 where $\#TP$, $\#FP$, $\#TN$, and $\#FN$ represent the numbers

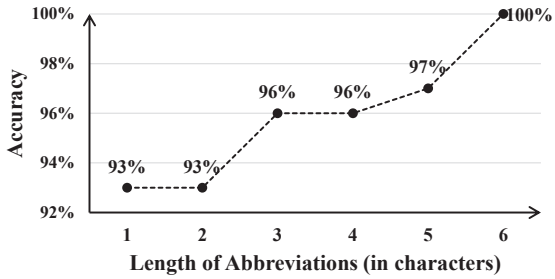

Figure 3: Impact of Abbreviations' Length

of true positives, false positives, true negatives, and false negatives, respectively. Notably, abbreviations that require expansion are called positive items, and thus true positives are those that should be expanded and are suggested to be expanded. Precision and recall in processing negative examples are $TN/(TN+FN)$ and $TN/(TN+FP)$, respectively. Precision and recall in processing positive examples are $TP/(TP+FP)$ and $TP/(TP+FN)$, respectively.

From the table, we make the following observations:

- First, SmartExpander is accurate. Most (95%) of the abbreviations are classified correctly, and only 5% of the abbreviations are misclassified.
- Second, SmartExpander is highly accurate in picking up negative examples, i.e., abbreviations that do not require expansion. In retrieving such negative abbreviations, SmartExpander reaches a high precision of 98% and a high recall of 96%.
- The performance varies slightly among subject applications. For example, its minimal and maximal accuracy is 93% (on DecFetcher) and 96% (on DavMail and DrJava), respectively. It may suggest that SmartExpander is accurate regardless of the application domains and developers of the subject applications.

To further investigate the reasons for incorrect classification, we manually analyzed the misclassified 93 abbreviations. Our analysis results suggest that shorter abbreviations are more challenging to classify than longer ones. Fig.3 illustrates how abbreviations' length influences the performance (accuracy) of SmartExpander. From this figure, we observe that the accuracy increases with the increase in abbreviations' length. We also notice that 53% (=49/93) of the misclassified abbreviations are composed of no more than two characters. One possible reason for the misclassification of shorter abbreviations is that the same short abbreviation often has

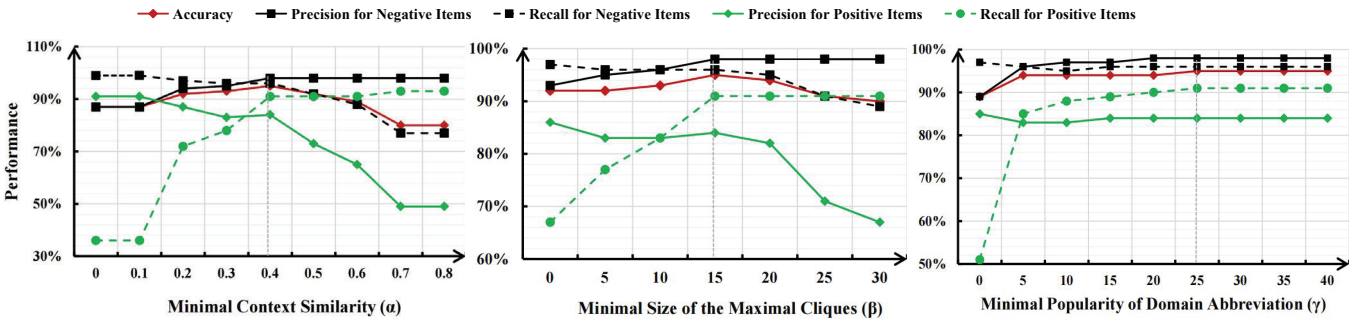

Figure 4: Influence of Thresholds

different interpretation (full terms) in different contexts. A typical example is the abbreviation "sb" in variable declaration "int sb" from jEdit [2]. SmartExpander takes it as a common abbreviation for the given domain and suggests to not expand it because abbreviation "sb" appears frequently within the same application. However, the expansion ("small buckets") of this abbreviation is significantly different from that ("StringBuilder") of other common abbreviations. Consequently, expanding this abbreviation helps reduce the confusion, and thus this abbreviation is manually labelled as positive (requiring expansion). Notably, SmartExpander cannot distinguish different abbreviations by their full expansions during the offline data mining phase because their full expansions are often unavailable.

Another reason for the misclassification is the limited corpus of source code leveraged by SmartExpander to discover common abbreviations. For the evaluation, we only leveraged 1,000 open-source Java applications for this task. As a result, some common abbreviations, e.g., "JDK" (standing for "Java Development Kit"), were missed because most of the leveraged applications did not contain such abbreviations. Consequently, to further improve SmartExpander, we should collect much more source code for the offline data mining in future.

We conclude based on the preceding analysis that SmartExpander is accurate regardless of application domains and developers of the subject applications.

4.6 RQ3: Influence of Thresholds

SmartExpander leverages the following thresholds:

- α: The minimal context similarity between abbreviations on the same maximal clique (called *minimal context similarity*).
- β: The minimal size of the maximal cliques that represent common abbreviations (called *minimal size of the maximal cliques*).
- γ: The minimal times a domain-specific common abbreviation should appear within a single project (called *minimal popularity of domain abbreviation*).

In the preceding evaluation, we leveraged the default values of the thresholds ($\alpha = 0.4, \beta = 15, \gamma = 25$) that were set empirically. To figure out the influence of different thresholds, we changed the value of a single threshold at a time and kept intact the default values of other thresholds. Our evaluation results are presented in Fig. 4.

From this figure, we make the following observations:

- First, all of the thresholds have significant influence on recall in retrieving positive/negative items. Decrease in any of such thresholds results in reduced recall in retrieving positive items and increased recall in retrieving negative items. Notably, such thresholds are leveraged to select potentially negative items, and thus reducing such thresholds (i.e., relaxing the condition for the selection) would result in more predicted negatives, which in turn leads to few predicted positives. As a result, recall in retrieving positive items is reduced whereas recall in retrieving negative items is increased.
- Second, increase in any of such thresholds resulted in increased precision in retrieving negative items and decreased precision in retrieving positive items. Increasing such thresholds makes the heuristics more conservative in predicting negative items. As a result, the precision in retrieving negative items is increased.
- Finally, the effect on the accuracy is more complex. The default values of such thresholds ($\alpha = 0.4, \beta = 15, \gamma = 25$) resulted in the maximal accuracy whereas decreasing or increasing the threshold values decreased in the accuracy of the approach.

We conclude based on the preceding analysis that the thresholds have significant influence on the performance of SmartExpander, and the default values result in the maximal accuracy.

4.7 RQ4: Effect of Heuristics

As introduced in Section 3, SmartExpander is composed of a sequence of heuristics: length-based heuristic, context-based heuristic, and common-abbreviation based heuristic. To quantitatively investigate the effect of such heuristics, we disabled one of them at a time and repeated the evaluation. Evaluation results are presented in Table 4 where the first column specifies which heuristic is disabled.

From the table, we make the following observations:

- All of the heuristics are indispensable. Disabling any of them resulted in significant reduction in accuracy of SmartExpander. The reduction varied from 4 (disabling length-based heuristic) to 33 percentage points (disabling context-based heuristic). The precision for positive abbreviations and recall for negative abbreviations suffers similar reductions as well. This finding may suggest that all the heuristics are useful and should not be dropped.
- Disabling any one of the heuristics could slightly increase the recall in retrieving positive samples. The increase varies from 2

Table 4: Effect of Heuristics

Setting	Accuracy	Negative Samples		Positive Samples	
		Precision	Recall	Precision	Recall
Enabling All Heuristics (Default Setting)	95%	98%	96%	84%	**91%**
Disabling Length-based Heuristic	91%	98%	91%	71%	93%
Disabling Context-based Heuristic	**62%**	98%	**54%**	**33%**	95%
Disabling Common-abbreviation based Heuristic	74%	98%	69%	42%	95%

to 4 percentage points. Notably, all the heuristics are designed to select negative samples, and items not selected by such heuristics are predicted as positive. Consequently, disabling any of the heuristics would select few negative samples, and more samples would be predicted as positive. As a result, the recall in retrieving positive samples increased. However, we also notice that the increase is minor. One possible reason is that such heuristics were highly accurate in selecting negative items, and thus only a small part (31 out 1,438) of the positive items were improperly predicted as negative (i.e., false negatives) by such heuristics.

- Disabling any of the heuristics dose not have any non-negligible effect on the precision in processing negative abbreviations. The actual precision in the third column of Table 4 should be 97.84%, 98.095%, 97.83%, and 98.16%, respectively. From such values, we can observe minor changes in the precision. However, we present all such values in Table 4 as 98% by rounding them to the nearest whole numbers. One possibility reason for such minor impact is that all the heuristics are leveraged independently to pick up negative abbreviations, and all of them are highly accurate. As a result, disabling any of them would not reduce the precision because the remaining heuristics were highly accurate in picking negative abbreviations.
- Disabling any of the heuristics resulted in significant reduction in recall for negative abbreviations. The reduction varies from 5 to 42 percentage points. The reason is that the heuristics were leveraged in parallel to pick up negative abbreviations, and thus disabling any of them would reduce the number of true negatives (and predicted negatives as well) that in turn reduced recall for negative items.
- Context-based heuristic has the greatest effect on the accuracy of SmartExpander. Disabling it reduced the accuracy from 95% to 62%. One possible reason is that a large number of abbreviations come from names of variables and parameters, and their full terms appear in the data types of the enclosing variables/parameters. A typical example is parameter "Exception e". Disabling the context-based heuristic would misclassify such abbreviations, which results in significant reduction in the performance of SmartExpander. Notably, common-abbreviation based heuristic has the second greatest effect, and disabling it reduced the accuracy significantly from 95% to 74%. One possible reason is that this heuristic identified a large number of common abbreviations successfully and suggested correctly to not expand them.

From the preceding analysis, we conclude that all of the leveraged heuristics are indispensable.

Table 5: Runtime of Online Classification (in Seconds)

Applications	KLOC	#Samples	Time (s)	Time(s) per Abbreviation
Mail	44	346	108	0.31
Doc	38	346	110	0.32
DrJ	185	378	130	0.34
Dubbo	186	377	154	0.41
jEdit	212	371	163	0.44
TOTAL	666	1,818	665	0.37

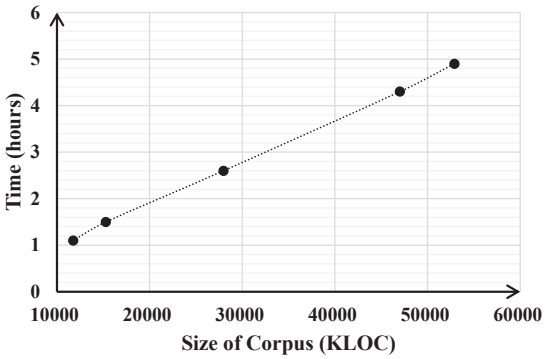

Figure 5: Scalability of Offline Data Mining

4.8 RQ5: Scalability

To answer RQ5, we investigated the scalability of SmartExpander by analyzing the average execution time of SmartExpander on a single abbreviation. The evaluation is conducted on a personal computer with Intel Core i7-6700, 16GB RAM, and Windows 10. Our evaluation results are presented in Table 5 whose last column presents how many seconds SmartExpander took to classify a single abbreviation (i.e., *Time* divide by *#Samples*). From this table, we observe that SmartExpander is highly efficient in classifying abbreviations. On average, it took only 0.37 seconds to classify a single abbreviation. We also observe that time increased slightly with the increase of application size. On the smallest applications (*Mail*), the average processing time is 0.31 seconds whereas it increased to 0.44 seconds on the largest application (*jEdit*). One possible reason is that the approach should visit all identifications with the application to determine whether the abbreviation under test represents a domain-specific common abbreviation (as specified in Section 3.4).

The time complexity of the visit is linearly dependent on the size of the applications.

We also investigate the time of offline data mining as specified in Sections 3.2-3.3. Compared to the online classification, offline data mining is much more time-consuming. It took 4.9 hours to finish the mining on 1,000 applications (containing more than 50 million lines of source code). To further analyze how the execution time is influenced by the size of source code (called *corpus*) involved in the data mining, we reduced the size by removing 200 application at a time and repeated the offline data mining on the updated corpus. Our evaluation results are present in Fig. 5. From this figure, we observe that the execution time is overall linearly dependent on the size of the corpus.

We conclude from the preceding analysis that SmartExpander (including both offline data mining and online classification) is efficient and scalable.

4.9 Threats to Validity

A threat to construct validity is that the manual labelling (classification) of the sampled abbreviations could be incorrect. On one side, the participants lack system knowledge of the subject applications. On the other side, the classification is essentially subjective, and thus it is likely that different participants assign different labels to the same abbreviations. However, such inaccurate labels served as the ground truth for the evaluation, and thus the evaluation results could be inaccurate. To reduce the threat, we asked three participants to label abbreviations independently, and requested them to reach an agreement by discussion in case of inconsistency.

Another threat to validity concerning the manual golden-set creation is biased to or against SmartExpander. If the participants known in advance how SmartExpander works, they might classify the abbreviations as SmartExpander does, which may seriously bias the evaluation results. To reduce the threat, we excluded the authors of the paper from the manual creation, and recruited developers who were not aware of SmartExpander.

A threat to the external validity is that only 5 applications and 1,818 abbreviations were involved in the evaluation. The limited number of involved applications and abbreviations may threaten the generality of the conclusions, i.e., the extent to which such conclusions can be generalized to other situations (other applications and abbreviations). To reduce the threat, we selected well-known applications from different domains and developed by different teams. Evaluation results suggest that SmartExpander is accurate on each of the subject applications, regardless of their difference in domains and developers. To further improve the generality of the conclusions in future, however, evaluation on more applications and more abbreviations should be conducted.

Another threat to the external validity is that we sampled abbreviations without considering their associated element types (e.g., variable or class names). If the distribution of abbreviations per type in the sample does not follow the distribution in the population, it may negatively impact the validity of the study.

The replication package, including implementation of SmartExpander, subject applications, manually labelled abbreviations, and the corpus of source code, is publicly available at [31].

5 CONCLUSIONS AND FUTURE WORK

Abbreviations are frequently used to shorten identifiers in source code, which significantly facilitates typing and typesetting of source code, especially complex and lengthy expressions. However, if used improperly, abbreviations may result in low readability and maintainability of source code. Although approaches haven been proposed to suggest full terms for abbreviations, we still lack automated tools in deciding which abbreviations should (or should not) be replaced with their full terms. To this end, in this paper, we propose such an automated approach based on a sequence of heuristics. Such heuristics concern different aspects of the abbreviations, i.e., length, popularity, and contexts. We evaluated the proposed approach with a large dataset containing 1,818 abbreviations. Our evaluation results suggest that the proposed approach is accurate, and the average accuracy is 95%.

In future, we would like to investigate how well the proposed approach can cooperate with abbreviation expansion tools. The collaboration is twofold. On one side, our approach could be leveraged to exclude a large number of abbreviations that do not require expansion. As a result, abbreviation expansion tools can ignore such abbreviations, and suggest developers with expansions (full terms) for other abbreviations only. On the other side, the length-based heuristic (Section 3.2) and the context-based heuristic (Section 3.5) of the proposed approach request full terms of the abbreviations under test. In the evaluation part, we suppose that such full terms are available (provided manually or by automated tools). In future, we would like to investigate whether the automated abbreviation expansion tools could serve as the source of such full terms requested by the proposed approach, and to what extent its performance would be influenced by the inaccurate expansions suggested by such abbreviation expansion tools. It could be fruitful to explore in more depth the factors considered by developers to make their decision in abbreviation expansion, and to investigate the causes of disagreement. Finally, to increase the impact of the approach, we will deploy our tool in some open-source projects and ask for developers' feedback on the use of the tool.

ACKNOWLEDGMENTS

The authors would like to say thanks to anonymous reviewers for their insightful comments and suggestions. This work was sponsored in part by the National Natural Science Foundation of China (61772071, 61690205, and 61772124).

REFERENCES

[1] 2021. Apache Dubbo. https://github.com/apache/dubbo.
[2] 2021. CBZip2OutputStream.java. https://github.com/romuloceccon/jedit/blob/master/installer/CBZip2OutputStream.java#L969.
[3] 2021. Cosine Similarity. https://en.wikipedia.org/wiki/Cosine_similarity.
[4] 2021. DavMail Gateway. https://github.com/mguessan/davmail.
[5] 2021. DocFetcher. https://github.com/vivainio/docfetcher.
[6] 2021. DrJava. https://github.com/DrJavaAtRice/drjava.
[7] 2021. jEdit. https://github.com/romuloceccon/jedit.
[8] 2021. Sample Size Calculator. https://www.surveysystem.com/sscalc.htm.
[9] 2021. Stanford NLP. https://github.com/stanfordnlp/CoreNLP/blob/master/src/edu/stanford/nlp/process/Stemmer.java.
[10] Eytan Adar. 2004. SaRAD: a Simple and Robust Abbreviation Dictionary. *Bioinform.* 20, 4 (2004), 527–533. https://doi.org/10.1093/bioinformatics/btg439
[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[12] A Apostolio and C Guerra. 1985. A fast linear space algorithm for computing longest common subsequences. (1985).

[13] Eran Avidan and Dror G. Feitelson. 2017. Effects of variable names on comprehension an empirical study. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, Giuseppe Scanniello, David Lo, and Alexander Serebrenik (Eds.). IEEE Computer Society, 55–65. https://doi.org/10.1109/ICPC.2017.27

[14] David W. Binkley, Dawn J. Lawrie, Steve Maex, and Christopher Morrell. 2009. Identifier length and limited programmer memory. *Sci. Comput. Program.* 74, 7 (2009), 430–445. https://doi.org/10.1016/j.scico.2009.02.006

[15] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. 1999. The Maximum Clique Problem. In *Handbook of Combinatorial Optimization*, Ding-Zhu Du and Panos M. Pardalos (Eds.). Springer, 1–74. https://doi.org/10.1007/978-1-4757-3023-4_1

[16] Caprile and Tonella. 2000. Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*. 97–107. https://doi.org/10.1109/ICSM.2000.883022

[17] C. Caprile and P. Tonella. 1999. Nomen est omen: analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*. 112–122. https://doi.org/10.1109/WCRE.1999.806952

[18] Nuno Ramos Carvalho, José João Almeida, Pedro Rangel Henriques, and Maria João Varanda Pereira. 2015. From source code identifiers to natural language terms. *J. Syst. Softw.* 100 (2015), 117–128. https://doi.org/10.1016/j.jss.2014.10.013

[19] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[20] Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2012. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 233–242. https://doi.org/10.1109/ICSM.2012.6405277

[21] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Softw. Qual. J.* 14, 3 (2006), 261–282. https://doi.org/10.1007/s11219-006-9219-1

[22] Oxford English Dictionary. 1989. Oxford English Dictionary. *Simpson, John A. & Weiner, Edmund S. C.* (1989).

[23] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing all maximal cliques in large sparse real-world graphs. *Journal of Experimental Algorithmics (JEA)* 18 (2013), 3–1.

[24] David Eppstein and Darren Strash. 2011. Listing all maximal cliques in large sparse real-world graphs. In *International Symposium on Experimental Algorithms*. Springer, 364–375.

[25] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola O. Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 286–296. https://doi.org/10.1145/3196321.3196347

[26] 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)*.

[27] Latifa Guerrouj, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. 2012. TRIS: A Fast and Accurate Identifiers Splitting and Expansion Algorithm. In *2012 19th Working Conference on Reverse Engineering*. 103–112. https://doi.org/10.1109/WCRE.2012.20

[28] Kevin A Hallgren. 2012. Computing Inter-Rater Reliability for Observational Data: An Overview and Tutorial. *Tutor Quant Methods Psychol* 8, 1 (2012), 23–34.

[29] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori L. 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, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*, Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey (Eds.). ACM, 79–88. https://doi.org/10.1145/1370750.1370771

[30] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter identifier names take longer to comprehend. *Empir. Softw. Eng.* 24, 1 (2019), 417–443. https://doi.org/10.1007/s10664-018-9621-x

[31] Yanjie Jiang. 2021. SmartExpander and Replication Package. https://github.com/jiangyanjie/smartExpander.

[32] Yanjie Jiang, Hui Liu, Jiahao Jin, and Lu Zhang. 2020. Automated Expansion of Abbreviations Based on Semantic Relation and Transfer Expansion. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2995736

[33] 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3338906.3338929

[34] Yanjie Jiang, Hui Liu, Jiaqi Zhu, and Lu Zhang. 2020. Automatic and Accurate Expansion of Abbreviations in Parameters. *IEEE Transactions on Software Engineering* 46, 7 (2020), 732–747. https://doi.org/10.1109/TSE.2018.2868762

[35] Dawn J. Lawrie and David W. Binkley. 2011. Expanding identifiers to normalize source code vocabulary. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 113–122. https://doi.org/10.1109/ICSM.2011.6080778

[36] Dawn J. Lawrie, David W. Binkley, and Christopher Morrell. 2010. Normalizing Source Code Vocabulary. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky (Eds.). IEEE Computer Society, 3–12. https://doi.org/10.1109/WCRE.2010.10

[37] Dawn J. Lawrie, Henry Feild, and David W. Binkley. 2007. Extracting Meaning from Abbreviated Identifiers. In *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007), September 30 - October 1, 2007, Paris, France*. IEEE Computer Society, 213–222. https://doi.org/10.1109/SCAM.2007.17

[38] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*. IEEE Computer Society, 3–12. https://doi.org/10.1109/ICPC.2006.51

[39] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. 2007. Effective identifier names for comprehension and memory. *Innov. Syst. Softw. Eng.* 3, 4 (2007), 303–318. https://doi.org/10.1007/s11334-007-0031-2

[40] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1188–1196. http://proceedings.mlr.press/v32/le14.html

[41] Guangjie Li, Hui Liu, Ge Li, Sijie Shen, and Hanlin Tang. 2020. LSTM-based argument recommendation for non-API methods. *Sci. China Inf. Sci.* 63, 9 (2020), 1–22. https://doi.org/10.1007/s11432-019-2830-8

[42] Guangjie Li, Hui Liu, and Ally S. Nyamawe. 2020. A Survey on Renamings of Software Entities. *ACM Comput. Surv.* 53, 2 (2020), 41:1–41:38. https://doi.org/10.1145/3379443

[43] Hui Liu, Qiurong Liu, Yang Liu, and Zhouding Wang. 2015. Identifying Renaming Opportunities by Expanding Conducted Rename Refactorings. *IEEE Transactions on Software Engineering* 41, 9 (2015), 887–900. https://doi.org/10.1109/TSE.2015.2427831

[44] Mircea Lungu and Jan Kurš. 2013. On planning an evaluation of the impact of identifier names on the readability and quality of smalltalk programs. In *2013 2nd International Workshop on User Evaluations for Software Engineering Researchers (USER)*. 13–15. https://doi.org/10.1109/USER.2013.6603079

[45] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2010. Recognizing Words from Source Code Identifiers Using Speech Recognition Techniques. In *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, Rafael Capilla, Rudolf Ferenc, and Juan C. Dueñas (Eds.). IEEE Computer Society, 68–77. https://doi.org/10.1109/CSMR.2010.31

[46] Christian Donald Newman, Michael John 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 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 269–279. https://doi.org/10.1109/ICSME.2019.00040

[47] Patric RJ Östergård. 2002. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics* 120, 1-3 (2002), 197–207.

[48] 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, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 190–199. https://doi.org/10.1109/ICSM.2013.30

[49] 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 *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 31–40. https://doi.org/10.1145/3196321.3196332

[50] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How do Scratch Programmers Name Variables and Procedures?. In *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*. IEEE Computer Society, 51–60. https://doi.org/10.1109/SCAM.2017.12

[51] Porfirio Tramontana, Michele Risi, and Giuseppe Scanniello. 2014. Studying abbreviated vs. full-word identifier names when dealing with faults: an external replication. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, Maurizio Morisio, Tore Dybå, and Marco Torchiano (Eds.). ACM, 64:1. https://doi.org/10.1145/2652524.2652593