Empirical Software Engineering (2023) 28:108
https://doi.org/10.1007/s10664-023-10339-2

An automated detection of confusing variable pairs with highly similar compound names in Java and Python programs

Hirohisa Aman[1] · Sousuke Amasaki[2] · Tomoyuki Yokogawa[2] · Minoru Kawahara[1]

Accepted: 10 May 2023 / Published online: 9 August 2023
© The Author(s) 2023

Abstract
Variable names represent a significant source of information regarding the source code, and a successful naming of variables is key to producing readable code. Programmers often use a compound variable name by concatenating two or more words to make it more informative and enhance the code readability. While each compound variable name is descriptive, a collection of them sometimes produces "confusing" variable pairs if their names are highly similar, e.g., "shippingHeight," vs. "shippingWeight." A confusing variable pair would adversely affect the code readability because it can cause a misreading or mix-up of variables during the programming or code review activities. Toward automated support for enhancing code readability, this paper conducts a large-scale investigation of compound variable names in Java and Python programs. The investigation collects 116,921,127 pairs of compound-named variables from 1,876 open-source Java projects and 106,943,523 pairs of such variables from 2,427 open-source Python projects. Then, this study analyzes those variable pairs from two perspectives of name similarity: string similarity and semantic similarity. Through an evaluation study with 30 human participants, the data analyses show that both string and semantic similarity can help detect confusing variable pairs in Java and Python programs. In order to distill confusing variable pairs automatically, support tools for detecting confusing variable pairs are also developed in this study.

Communicated by: Simone Scalabrino

This paper belongs to a Topical Collection: *Code Legibility, Readability, and Understandability*
Guest Editors: Simone Scalabrino, Rocco Oliveto, Felipe Ebert, Fernanda Madeiral, Fernando Castor.

✉ Hirohisa Aman
aman@ehime-u.ac.jp

Sousuke Amasaki
amasaki@cse.oka-pu.ac.jp

Tomoyuki Yokogawa
t-yokoga@cse.oka-pu.ac.jp

Minoru Kawahara
kawahara@ehime-u.ac.jp

[1] Ehime University, Matsuyama, Ehime 790–8577, Japan

[2] Okayama Prefectural University, Soja, Okayama 719–1197, Japan

Keywords Confusing variable names · Compound names · String similarity · Semantic similarity

1 Introduction

Variables are fundamental components of programs, and variables' names significantly impact code readability (Deissenboeck and Pizka 2006; Knuth 2003; Lacomis et al. 2019; Pigoski 1996). They serve as an influential source of information regarding the program entities (Caprile and Tonella 2000). Well-chosen names would make the code readable and help developers review the program smoothly. In other words, poor-chosen names degrade the code readability: if we replace the variable names with unrelated (meaningless) strings, we can easily obfuscate our programs (Ceccato et al. 2014; Low 1998).

To make a variable name informative, many programmers use English dictionary words or well-known abbreviations for the name (Liblit et al. 2006). The usefulness of choosing fully spelled words or abbreviations for variable names has been empirically shown in the past (Lawrie et al. 2007; Scanniello et al. 2017). When the role of the variable becomes complex, it is helpful to use a *compound* name (Binkley et al. 2013) organized by two or more words, like "`positionOfPlayer`." Because a compound name corresponds to a short phrase describing the variable's role, many programmers and code reviewers can easily understand what the variable stores and how it works in the program. Empirical studies have also reported the positive effect provided by a compound name (Schankin et al. 2018).

However, there is also a negative effect of compound names on code readability. Tashima et al. (2018) proposed *confusing* variable pairs, where two variables have highly similar names like "`lineIndex`" and "`lineIndent`." Due to their high similarity, they cause a risk of misreading or mixing up variables during the programming or code review activities. Thus, they may adversely affect the code readability even though each variable name is informative. Aman et al. (2019) reported an empirical study showing that confusing variable pairs are related to the fault-proneness of Java methods. Hence, automatically detecting confusing variable pairs would be helpful for developers in successful programming and code review.

Although the previous studies (Tashima et al. 2018; Aman et al. 2019) measured the similarity between variable names using the Levenshtein distance (Gusfield 1997), they missed the semantic perspective of the name similarity. For example, we may consider the variable name "`sizeOfBlocks`" similar to "`blockSizes`." However, the Levenshtein distance-based measure in the previous study judges the pair as *dissimilar* since we need to edit 92% (11 out of 12) characters of "`sizeOfBlocks`" to convert it to "`blockSizes`." It is better to consider both the string and semantic similarity to examine the confusing variable pairs. To analyze the name similarity from both perspectives above, we conduct large-scale investigations of compound variable names in Java and Python programs. Then, to examine if those name similarities can contribute to distilling confusing variable pairs that decay the code readability, we perform an evaluation study using an ordinal scale with human participants on the perceived confusion of a given pair of variable names. Furthermore, we develop support tools for automatically detecting confusing variable pairs in Java and Python.

Although our variable name evaluation primarily focuses on code readability, not only variable namings can evaluate that quality attribute. Comments play another valuable role in making the code readable as well. Moreover, the program's visual and structural aspects, such as proper indentations, well-inserted blank lines, and well-designed branch or loop structures,

are also essential for readable code (Scalabrino et al. 2021). Since code readability is a complicated quality attribute, in this paper, we cover a part of the code readability aspects—*the ease of identifying each variable when two or more variables are available within a region of the program*. As mentioned above, even though each variable has a well-chosen name and is easy to read, a collection of similar variable names might decrease the ease of identifying those variables because it might be harder to distinguish them. Highly similar (confusing) variable names may cause a misreading or mixing up of variables. Our goal is to support developers by automatically detecting confusing variable names to enhance code readability.

This paper is an extended version of the previous study (Aman et al. 2021a) presented at the 1st Workshop on Automated Support to Improve code Readability (AeSIR2021):

1. We investigated not only Java programs but also Python programs in this study to see a difference in naming trends between these two major languages.
2. To enhance the generalizability of our data analysis findings, we collected about seven times more data (pairs of compound variable names: 116,921,127 pairs found in 1,876 Java projects and 106,943,523 ones found in 2,427 Python projects) than in the previous study (31,807,693 pairs found in 684 Java projects). Then, we performed an evaluation study using an ordinal scale with human participants to examine if the name similarity notion contributes to detecting confusing variable pairs.
3. Moreover, we developed and published support tools for detecting confusing variable pairs in Java/Python programs, which are available from our supplementary site: `https://github.com/amanhirohisa/cvpfinder`.

The remainder of this paper is organized as follows. Section 2 describes the related work with the research motivation of this paper. Section 3 explains the compound variable name and the notion of confusing variable pairs. Then, to quantify the degree to which a variable pair is confusing, Section 4 elaborates on how to measure the name similarity from two perspectives: string similarity and semantic similarity. Section 5 describes the development of support tools for evaluating similarities of variable pairs (detecting confusing variable pairs) in Java and Python programs. Section 6 reports and discusses the results of our investigation on compound variable names to examine if the name similarities can contribute to distilling confusing variable pairs. Finally, Section 7 presents our conclusion and future work.

2 Related Work

Variables' names have been studied as fundamental program elements influencing program comprehension and code readability. This section briefly describes the related work focusing on the variable names and our research motivation.

We often see single-letter variable names in many programs, and the single-letter name would be a naming category attracting many programmers' interest. Beniamini et al. (2017) conducted a large-scale data collection and analysis of single-letter variable names in C, Java, JavaScript, PHP, and Perl programs. They collected 1,000 popular Git repositories from GitHub and conducted their data analysis. As a result, they showed that the single-letter name is not the most common in the studied four languages besides JavaScript; most variables in JavaScript programs are single-letter or two-letter names. Furthermore, they conducted an online survey and reported that some single-letter names are commonly used for specific usage, e.g., "`i`, `j`, `k`" for loop indexes, "`s`" for strings, and "`t`" for times. Swidan et al. (2017) also analyzed single-letter names in Scratch programs using the dataset presented

by Aivaloglou and Hermans (2016). The analysis showed that single-letter names are less common in Scratch programs: the percentage of such names was only 4%. As the above previous work reported, while single-letter names are fundamental variable names, they do not seem to form the majority group in the real world of naming variables. In other words, other naming styles (categories) would be more attractive for programmers. For example, Aman et al. (2021b) reported that the English word names and compound names share about 40% and 34%, respectively, through a large-scale investigation of Java programs from GitHub; in comparison, the percentage of single-letter names was about 17%. Moreover, its percentage decreases to about 5% when the variable's scope becomes broad.

Next, we focus on the related work regarding more descriptive names like English word names and compound names. Lawrie et al. (2007) examined the impact of naming style on program comprehension through an experiment using three different naming styles of variables: single-letter (e.g., "`c`"), abbreviation (e.g., "`cnt`"), and fully spelled word (e.g., "`count`"). Their experiment with 128 programmers statistically showed that fully spelled words and abbreviations are better than single-letter names in program comprehension. Scanniello et al. (2017) also conducted empirical studies involving 100 programmers to compare the above three kinds of names regarding program comprehension and fault detection/fixing. Their empirical results support the finding of Lawrie et al. (2007). Schankin et al. (2018) empirically demonstrated the positive aspect of descriptive compound names. Through an empirical study with 88 programmers, they reported that descriptive names aid programmers to detect faults quicker than short, non-descriptive names. As the previous work showed, making variable names descriptive is a better way to name variables. This trend also underlies recent name-related studies: for example, Tran et al. (2019) studied a way to recover meaningful variable names from the shortened names in JavaScript programs; Lacomis et al. (2019) proposed a technique to reconstruct meaningful variable names in the program decompiled from the binary.

Because making variable names meaningful and descriptive leads to longer variable names, there is another concern regarding such naming. Binkley et al. (2009) focus on the human short-term memory during programming and warn about the harmfulness of long names. Figure 1 presents an example of variable names where each name is descriptive but not easy to read. The concern given by Binkley et al. motivated Tashima et al. (2018) to propose the notion of "*confusing* variable pair": even though each variable name is descriptive, their collection can cause another harmfulness to code readability when their names are highly similar. Tashima et al. quantified the similarity between variable names using Levenshtein distance and showed that Java methods having confusing (highly similar) variable pairs are more likely to experience fault fixes. Furthermore, Aman et al. (2019) reported that confusing variable pairs correlate with fault introduction changes (commits producing new faults) through an empirical study with ten open-source projects.

The studies by Tashima et al. (2018) and Aman et al. (2019) are the previous core work for this paper. Although those studies utilized Levenshtein distance to measure the similarity of variable names, they quantified only the "string" similarity but missed the "semantic"

```
distance_between_abscissae = first_abscissa - second_abscissa;
distance_between_ordinates = first_ordinate - second_ordinate;
cartesian_distance = square_root(
   distance_between_abscissae * distance_between_abscissae
   + distance_between_ordinates * distance_between_ordinates
);
```

Fig. 1 An example code presented in Binkley et al. (2009)

perspective. For example, let us consider two names, “sizeOfBlocks” and “blockSizes.” They are not similar to each other in terms of string similarity. However, when we focus on their semantic aspects, they can be similar names. Such a difference has motivated us to cover both name similarity concepts in this paper.

3 Compound Variable Name and Confusing Name

Making a variable’s name descriptive would be better to enhance code readability. However, when two or more variables with similar descriptive names appear in a program, we apprehend that such a collection of descriptive variable names forms “confusing” variable pairs as described above. Although each of those variables may positively contribute to code readability, we are concerned about the risk that a gathering of them negatively affects. To examine descriptive variable names, we focus on compound names consisting of two or more words in this paper. In this section, we describe the compound name and explain the notion of confusing variable pairs.

3.1 Compound Variable Name

To the best of the authors’ knowledge, most programming languages accept the variable names satisfying the following conditions[1]:

1. It must be a sequence of English alphabet characters, digits, or underscore (“_”), whose first character is an English alphabet character or underscore[2]. In other words, it must be a string matching the regular expression “`[a-zA-Z_][a-zA-Z_0-9]*`”.
2. It must not be a language’s keyword or a predefined literal (e.g., “true”, “false”, “null”).

Although programmers may use any names for their variables if they comply with the programming language grammar, it is better to choose meaningful names to express the variables’ roles, as many programming practices and coding standards commonly recommend (Kernighan and Pike 1999; Gosling et al. 2014; Free Software Foundation 2018; kernel development community 2016). To this end, programmers would choose a fully spelled word or its abbreviated form as a variable name. However, there are also cases where it is hard to express a variable’s name by a single word because its role is complicated. For such variables, programmers tend to use compound names that consist of two or more terms (words or abbreviations). Indeed, an investigation report says that about 34% of variable names are compound names, and their percentage surpasses 50% when the variable scopes are broad (Aman et al. 2021b).

The most popular style for a compound variable name includes the camelCase and the snake_case (Binkley et al. 2013). In the camelCase, we join all terms and capitalize the initial character of the second or later terms to indicate the separation. In the snake_case, we indicate the separation of terms by the underscore (“_”). For example, if we make a name from “data file name,” the compound names in the camelCase and the snake_case can be “dataFileName” and “data_file_name,” respectively.

Some programmers produce a compound variable name by concatenating two or more terms without separation, such as “testfile”(test + file) because the element terms are

[1] For convenience, we do not consider non-ASCII characters in our programs.

[2] There are some exceptions: we can use white space in Scratch and “.” in R, for example.

Fig. 2 An example of confusing variable names that we may encounter during our programming activity

simple and widely used. We regard a name as a compound name in this paper if we can split it into English dictionary words or well-known abbreviations[3]. We can utilize Spiral 1.1.0[4] (Hucka 2018), a sophisticated Python module for splitting identifiers, to split variable names.

3.2 Confusing Variable Pair

A compound variable name seems to be a phrase describing the role of the variable. Thus, it is readable for many programmers and code reviewers without additional descriptions, such as the comments in the program. However, when we have two or more well-described compound names in our code fragment, they may cause a side effect on the code readability. If those compound variable names are similar to each other, they can be *confusing* variable pairs (Tashima et al. 2018; Aman et al. 2019). Some programmers or code reviewers mix up those variables during the programming or reviewing activities, even though each variable's name is easy to read.

Figure 2 shows an example of a programmer encountering confusing variable pairs while programming a Java code on Visual Studio Code, an integrated development environment (IDE). In this example, the programmer is about to write "`shippingWeight`," and the IDE suggests candidates including the correct name and other highly similar ones. Suppose the programmer wrongly chose "`shippingHeight`" or "`shippingMaxWeight`" for it. In those cases, the programmer and the code reviewers might not quickly find those mistakes because "`shippingHeight`" and "`shippmingMaxWeight`" are highly similar to "`shippingHeight`" in terms of string similarity or semantic similarity. Although the above case shown in Fig. 2 is just an example, it illustrates the risk of variable mix-up caused by confusing variable pairs. It is better to focus on confusing variable pairs toward successfully managing the code readability.

[3] We leverage PyEnchant 3.2.2, a spellchecking library, to check words.

[4] https://github.com/casics/spiral

Here, we define a *confusing variable pair* as a pair of the following two variables v_1 and v_2 in a program:

1. The scope of v_1 overlaps with that of v_2, and
2. The name of v_1 is highly similar to that of v_2,

where the scope of a variable is the line number interval in which the variable is available. When two variables are available simultaneously within a block, and their names are highly similar, we consider they become a confusing variable pair that may adversely affect an aspect of the code readability—the ease of identifying each variable. In other words, such confusing variables can be harder to distinguish and raise a risk of variable misreading or mixing up during the programming or code review activities.

However, the above second condition remains to be clearly defined. To overcome it, we have to define the similarity between variables clearly. There can be two perspectives: string similarity and semantic similarity. We describe these similarities in the next section.

4 Evaluation of Name Similarity

To detect confusing variable pairs in programs, we need to quantify the similarity of variable names. In this section, we describe how to evaluate name similarities in terms of both string and semantic similarity.

4.1 Evaluation of String Similarity

When two variable names have a common part (substring), they look similar. The larger the common part is, the more similar they look. For example, a variable name "`shippingHeight`" looks more similar to "`shippingWeight`" than "`productHeight`." We can quantify such a string similarity by focusing on how many characters we should edit to convert one name to another. A character edit is one of character addition, deletion, and substitution. Then, the least number of character edits to convert can be an index of string dissimilarity, referred to as the *Levenshtein distance* (Gusfield 1997).

We can quantitatively compare the similarities of the above example names as follows. For the sake of convenience, we denote the Levenshtein distance between $name_1$ and $name_2$ by $d_L(name_1, name_2)$. Then, we obtain

$$d_L(\text{“shippingHeight”}, \text{“shippingWeight”}) = 1\ ,$$
$$d_L(\text{“shippingHeight”}, \text{“productHeight”}) = 8\ .$$

For the first pair ($name_1$ ="`shippingHeight`", $name_2$ ="`shippingWeight`"), we can convert $name_1$ to $name_2$ by only substituting "`H`" with "`W`," i.e., the least number of required character edits is one. On the other hand, for the second pair, ($name_1$ ="`shippingHeight`", $name_2$ ="`productHeight`"), we need eight character edits to convert $name_1$ to $name_2$ as: $\texttt{s} \rightarrow \emptyset$ (delete "`s`"), $\texttt{h} \rightarrow \texttt{p}$ (substitute "`h`" with "`p`"), $\texttt{i} \rightarrow \texttt{r}$, $\texttt{p} \rightarrow \texttt{o}$, $\texttt{p} \rightarrow \texttt{d}$, $\texttt{i} \rightarrow \texttt{u}$, $\texttt{n} \rightarrow \texttt{c}$, and $\texttt{g} \rightarrow \texttt{t}$.

We can compute $d_L(name_1, name_2)$ by the following recurrence formula (Gusfield 1997):

$$d_L(name_1, name_2) = \begin{cases} \text{length}(name_1) & (\text{length}(name_2) = 0), \\ \text{length}(name_2) & (\text{length}(name_1) = 0), \\ d_L(name_{1,2-*}, name_{2,2-*}) & (name_{1,1} = name_{2,1}), \\ 1 + \min\{\, d_L(name_{1,2-*}, name_2), \\ \qquad d_L(name_1, name_{2,2-*}), \\ \qquad d_L(name_{1,2-*}, name_{2,2-*}) \,\} & (otherwise), \end{cases}$$

where "length($name_k$)" indicates the character count of $name_k$, "$name_{k,1}$" is the first character of $name_k$, and "$name_{k,2-*}$" is the substring of $name_k$ made by erasing its first character (for $k = 1, 2$).

Although d_L reasonably measures how two names look different, the length of the name may also affect the similarity evaluation. For example, while the following two pairs have the same Levenshtein distance ($d_L(\cdot, \cdot) = 1$), the similarity level does not look identical:

$$d_L(\text{“fileA”}, \text{“fileB”}) = 1 \,,$$
$$d_L(\text{“distanceBetweenAandB”}, \text{“distanceBetweenAandC”}) = 1 \,.$$

To consider such a difference, we introduce the following normalized Levenshtein distance, nd_L:

$$nd_L(name_1, name_2) = \frac{d_L(name_1, name_2)}{\max\{\, \text{length}(name_1), \text{length}(name_2) \,\}} \,.$$

Then, the normalized Levenshtein distance evaluates the above example pairs successfully:

$$nd_L(\text{“fileA”}, \text{“fileB”}) = \frac{1}{\max\{5, 5\}} = 0.2 \,,$$
$$nd_L(\text{“distanceBetweenAandB”}, \text{“distanceBetweenAandC”}) = \frac{1}{\max\{20, 20\}} = 0.05 \,.$$

Because the distance is an inverse similarity measure, we use the following Levenshtein similarity, sim_L, in this paper:

$$sim_L(name_1, name_2) = 1 - nd_L(name_1, name_2) \,.$$

The range of sim_L value is [0, 1]. The higher value the pair has, the more similar they are.

4.2 Evaluation of Semantic Similarity

Next, we focus on the semantic similarity of variable names and describe how to evaluate them. To evaluate the semantic similarity, we need to express those names numerically while considering their semantics. To this end, we leverage the tools developed in the natural language processing study world.

There have been techniques for quantitatively representing the semantics of words in natural language processing studies: *word embeddings*. An embedding is a numerical vector representation of a word. Word2Vec (Mikolov et al. 2013)s is one of the most popular models for producing embeddings using the neural network. It learns the associations of words in the training data (texts), and Word2Vec uses the trained network to produce the corresponding word vectors. Because semantically similar words are likely to appear in a similar

context, the corresponding vectors can also become close in the resulting vector space. Thus we can evaluate the semantic similarity between words using the closeness between the corresponding vectors. To produce the document (sequence of words) embedding rather than the word embedding, an extended version of Word2Vec has been developed: Doc2Vec (Le and Mikolov 2014). We can also evaluate the semantic similarity between sentences using Doc2Vec.

We can regard a compound variable name as a short sentence by splitting the variable name into its element words. Hence, we can also measure the semantic similarity between compound variable names using Doc2Vec through such a name splitting. Notice that Doc2Vec is not the state-of-the-art model in natural language processing studies. There have recently been advanced natural language models, and "Bidirectional Encoder Representations from Transformers (BERT)" (Devlin et al. 2019) is a well-known promising one. Hence, we tried leveraging some of the state-of-the-art "Sentence-BERT" models[5]. However, they did not seem to be suited for our application. Most sequences of words made by splitting compound variable names consist of a few (two or three) words and form incomplete short sentences. Because those variable-specific sentences differ substantially from general English sentences used for training Sentence-BERT models, the above state-of-the-art models could not adequately evaluate the similarity of such incomplete short sentences. Although we might be able to fit those models to our application through a fine-tuning process, we would like to perform it as future work because such a tuning requires much richer computing resources.

We present the steps to produce sentence vectors for compound variable names using Doc2Vec below.

1. Name splitting: We split a compound variable name by the sophisticated identifier splitter, Spiral 1.1.0 (Hucka 2018). The tool can split variable names into element tokens by the camelCase or the snake_case. For example, we split "`numberOfLetter`" and "`number_of_letter`" into "`number`, `Of`, `Letter`" and "`number`, `of`, `letter`," respectively. Moreover, Spiral can split a simply-concatenated name like "`testfile`" into "`test`" and "`file`". When we can obtain two or more English words[6] or abbreviations through the splitting by Spiral, we regard the original variable name as a compound name. We leverage PyEnchant 3.2.2, a spellchecking library, to check whether or not a word is an English dictionary word. To cover abbreviations that are not ordinary English words, we prepared an additional private dictionary[7] and used it in the PyEnchant checking.
2. Preprocessing: We decapitalize all words to avoid the difference in letter case affecting the vectorization. After that, we perform the stemming of the word to uniform the word styles because a word can appear in different styles like "`list`," "`lists`," "`listed`," and "`listing`." We utilize PorterStemmer in Python nltk 3.7 to stem the words.
   Furthermore, we sometimes encounter a variable name using a number such as "`inputBuffer2`." Although the number is a part of the name, it would not be essential for considering the meaning of the variable name. To avoid any impact caused by such a number, we replace all numbers appearing in a compound variable name with the special token "`<num>`."

[5] We can obtain some pre-trained language models at https://huggingface.co/models. We tried using "sentence-transformers/all-MiniLM-L6-v2," "sentence-transformers/all-mpnet-base-v2," "sentevnce-transformers/sentence-t5-large," "sentence-transformers/gtr-t5-large," "flax-sentence-embeddings/all_datasets_v4_MiniLM-L6," "digio/Twitter4SSE," and "flax-sentence-embeddings/reddit_single-context_mpnet-base," as a trial.

[6] To avoid wrong extractions such as "`a`" and "`an`" from "`aan`," we exclude too short tokens, shorter than three characters, from our English word checking.

[7] It is available from out support tool site https://github.com/amanhirohisa/cvpfinder.

3. Vectorizing: After processing the variable names above, we train the Doc2Vec model and vectorize those names. By computing the similarity between vectors, we can obtain the degree to which two names are semantically similar. In this study, our similarity metric between vectors is the cosine similarity. The range of similarity is $[-1, 1]$; the higher value the variable pair has, the more similar they are.

□

Through the above steps, for instance, we could judge that "`sizeOfBlocks`" is relatively similar to "`blockSize`" automatically.

5 Support Tools

The similarity evaluation of variable names described in the previous section forms the foundation for detecting the confusing variable pair. In general, a software product includes many source files, and each source file has various variables. Hence, it is not easy to extract all variable pairs and evaluate their similarities, even if we focus only on compound names. To support programmers with the detection of confusing variable pairs, we developed tool support for Java and Python programs. We describe the support tools in this section.

5.1 Outline

Both support tools for Java and Python process a source file and output the detection report in the following steps (Fig. 3).

1. The tool parses the source file and obtains the abstract syntax tree (AST) data.
2. It analyzes the AST and extracts the variables with compound names. The tool produces the variable table providing the variable's name, type, kind (local variable, class field, etc.) and scope information.
3. The tool extracts the variable pairs whose scopes overlap, i.e., the candidate set of confusing variable pairs, from the variable table.
4. For each variable pair in the candidate set, the tool computes the similarity between their names from both string and semantic perspectives.
5. Finally, the tool warns the variable pairs using the pre-defined similarity thresholds and sorts them to output the confusing variable pair report.

For the convenience of developing parsers, we separately developed the analysis program corresponding to steps 1 and 2 for Java and Python, respectively. The programs corresponding to the remaining steps are common to Java and Python but use different Doc2Vec models and thresholds for these languages. The support tools are available from our public repository https://github.com/amanhirohisa/cvpfinder.

5.2 Support Tool for Java

To parse Java source files, we utilized JavaParser 3.42.2[8]. JavaParser is an open-source library for analyzing Java programs. We developed an analysis program in Java: the program parses the source file using JavaParser and traverses the AST to obtain variable declarations

[8] http://javaparser.org/

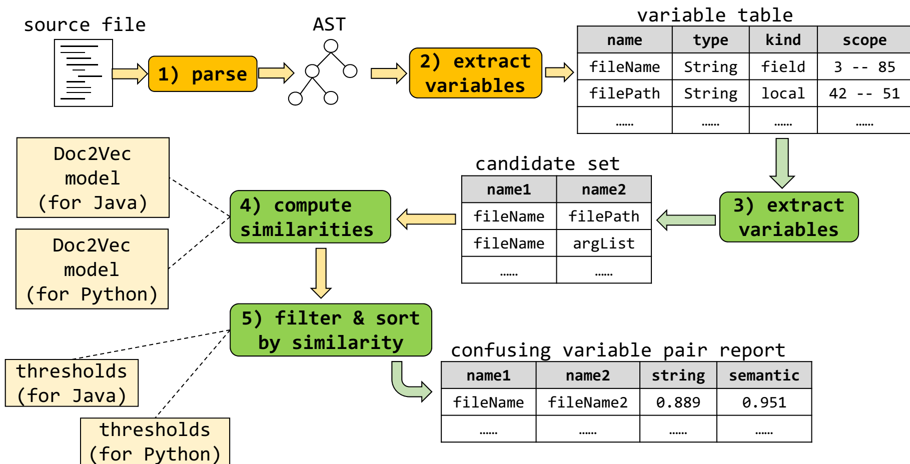


Fig. 3 Outline of processing steps in the support tool

(corresponding to steps 1 and 2). In this tool, a "variable" corresponds to a local variable declared in a method, a method parameter, or a class field. We extract the variable's name and type from the declaration's AST node. Moreover, we examine the scope range where the variable is available by checking the corresponding ancestor AST nodes. We define the beginning and the ending of a variable's scope as the first and the last lines of the closest block containing the variable's declaration, respectively. Notice that our tool does not support lambda expressions because they can declare different variables with the same name on the same line of code, and we cannot discriminate those variables.

After extracting variables from Java programs, we single out the variables with compound names. As described in Sections 3.1 and 4.2, we consider a variable's name to be a compound name if it is written in camelCase, snake_case, or a simple concatenation of dictionary words (or well-known abbreviations). We developed a Python program to pick compound-named variables using Spiral 1.1.0 and PyEnchant 3.2.2. To cover well-known abbreviations, we prepared an additional private dictionary as well.

For the next step (Step 3), we developed another Python program to find all variable pairs such that their scopes overlap. For Step 4, we developed Python programs to compute the string similarity using the Levenshtein similarity (see Section 4.1) and the semantic similarity using the Doc2Vec model (see Section 4.2), respectively. Since there may be a difference in naming variables between Java and Python trends, we prepared different models that we trained using different variable data. We used the variable data collected in our investigation described in the next section.

For the final step (Step 5), we developed a Python program to rank the candidate variable pairs in terms of the "confusion" level. Because there are two criteria of name similarity, we use two thresholds to filter confusing variable pairs. Then, we sort them in descending order of the number of similarity scores surpassing the predefined thresholds. We design the thresholds as user-adjustable parameters in this tool. We will explore thresholds through an evaluation study with human participants presented in the next section.

5.3 Support Tool for Python

We leveraged the `ast` module provided in Python to parse Python source files and obtained the corresponding ASTs. Python grammar has no explicit statement for variable declaration (except for the "`global`" declaration). The environment allocates a variable at which the programmer assigns a value to the variable first. In other words, such a first assignment to a variable corresponds to the variable's declaration. Thus, we traverse the AST to find `ast.Name` nodes and regard them as variable declaration points if they are used in the "`Store`" context. Furthermore, we identify the variable's scope by checking the corresponding ancestor AST nodes. This tool supports the following three kinds of variables:

- global variables, which are available anywhere in the module,
- class attributes, which are available anywhere within the class, and
- local variables, which are available only within the function or the method.

Notice that we can also use a narrower scope variable that are available only within a list like "`x`" in "`[x for x in list]`"; it is referred to as "list comprehension." However, this tool does not support variables in the context of list comprehension because they are limited to list construction and would not become a part of confusing variable pairs. Similarly, the tool omits variables used in the context of "set comprehension" or "dictionary comprehension."

After extracting variables from Python program as described above, our tool performs the remaining steps, i.e., making the variable table, making the candidate set, computing similarities, and reporting the confusing variable pairs. Because the remaining steps are the same as the ones described in Section 5.2, we will omit the details.

5.4 Example

We present a simple example of confusing variable pair detection by our tools. Our tools are shell scripts calling the related Java programs and Python programs to extract variable pairs and compute their similarities. The script names are "`cvpfinder4j`" and "`cvpfinder4p`" for Java and Python, respectively.

We can give a single source file or a directory containing source files as the input for our tool. Figure 4 illustrates the execution example where we gave directory "`storm/external/storm-jdbc`" as the input. The directory contains 20 Java source files, and our tool detected 18 confusing variable pairs. Figure 5 presents report file (`report/report.csv`) produced by that execution, and Fig. 6 zooms in on the part of confusing variable names. In the report CSV file, the path of the analyzed source file, the mark representing if it is a confusing variable pair, the variable names with their scopes, and the computed string and semantic similarities. The report is sorted to raise the confusing variable pairs to the top of the list. The mark "`**`" indicates the pair's similarity is higher

```
$ cvpfinder4j  storm/external/storm-jdbc
[target dir] storm/external/storm-jdbc
(20 java files)
Computing string similarity ... done.
Computing semantic similarity ... done.

18 confusing variable pairs found !
see report/report.csv for the details.
```

Fig. 4 An example of processing Java source files by the support tool

1	path	confusing	name1	name2	string_similarity	semantic_similarity
2	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcLookupBolt.java	**	tableName @(35--100)	tableName @(49--55)	1	1
3	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcLookupBolt.java	**	insertQuery @(35--100)	insertQuery @(57--63)	1	1
4	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/common/Column.java	**	selectQuery @(29--76)	selectQuery @(36--44)	1	1
5	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/common/Column.java	**	jdbcLookupMapper @(29--76)	jdbcLookupMapper @(36--44)	1	1
6	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/common/HikariCPConnectionProvider.java	**	columnName @(40--111)	columnName @(51--55)	1	1
7	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/common/HikariCPConnectionProvider.java	**	columnName @(40--111)	columnName @(57--60)	1	1
8	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/mapper/SimpleJdbcLookupMapper.java	**	connectionProvider @(37--266)	connectionProvider @(43--46)	1	1
9	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/mapper/SimpleJdbcLookupMapper.java	**	queryTimeoutSecs @(37--266)	queryTimeoutSecs @(43--46)	1	1
10	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/mapper/SimpleJdbcMapper.java	**	outputFields @(24--59)	outputFields @(28--33)	1	1
11	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcQuery.java	**	schemaColumns @(27--92)	schemaColumns @(41--44)	1	1
12	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcStateFactory.java	**	jdbcLookupMapper @(112--155)	jdbcLookupMapper @(141--144)	1	1
13	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcStateFactory.java	**	connectionProvider @(112--155)	connectionProvider @(121--124)	1	1
14	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcStateFactory.java	**	tableName @(112--155)	tableName @(126--129)	1	1
15	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcStateFactory.java	**	insertQuery @(112--155)	insertQuery @(131--134)	1	1
16	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcStateFactory.java	**	selectQuery @(112--155)	selectQuery @(146--149)	1	1
17	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/trident/state/JdbcStateFactory.java	**	queryTimeoutSecs @(112--155)	queryTimeoutSecs @(151--154)	1	1
18	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcInsertBolt.java	*	connectionProvider @(27--96)	connectionProviderParam @(59--62)	0.782608696	0.853873
19	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/mapper/SimpleJdbcLookupMapper.java	*	columnList @(70--73)	columnLists @(53--94)	0.909090909	0.9384209
20	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcInsertBolt.java		queryTimeoutSecs @(27--96)	connectionProviderParam @(59--62)	0.130434783	0.41746855
21	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcLookupBolt.java		insertQuery @(35--100)	queryTimeoutSecs @(65--68)	0.1875	0.6560837
22	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcLookupBolt.java		insertQuery @(35--100)	topologyContext @(70--76)	0.066666667	0.42882505
23	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcLookupBolt.java		insertQuery @(35--100)	columnLists @(82--90)	0	0.3725243
24	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcLookupBolt.java		insertQuery @(35--100)	outputFieldsDeclarer @(96--99)	0.2	0.26932004
25	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/common/Column.java		selectQuery @(29--76)	jdbcLookupMapper @(29--76)	0.125	0.34974438
26	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/common/Column.java		selectQuery @(29--76)	connectionProvider @(36--44)	0.222222222	0.33027032
27	storm/external/storm-jdbc/src/main/java/org/apache/storm/jdbc/bolt/JdbcInsertBolt.java		queryTimeoutSecs @(27--96)	topologyContext @(70--84)	0.1875	0.3516874

Fig. 5 An example of detection report produced by the support tool

than both the string and semantic similarity thresholds; the mark "*" indicates the pair's similarity is higher than one of the string similarity or semantic similarity thresholds. In the example, 16 pairs are identical named pairs, i.e., they are pairs of class fields and local variables with the same names; two pairs are highly similar names: "`connectionProvider`" vs. "`connectionProviderParam`," and "`columnList`" vs. "`columnLists`." They represent typical examples of confusing variable pairs.

We omit to show examples of using our tool for Python (`cvpfinder4p`) because the usage is the same as the tool for Java.

6 Large-Scale Investigation of Confusing Variable Pairs

We conducted a large-scale investigation of compound variable names in Java and Python programs. In this section, we report and discuss the results.

confusing	name1	name2	string_similarity	semantic_similarity
**	tableName @(35--100)	tableName @(49--55)	1	1
**	insertQuery @(35--100)	insertQuery @(57--63)	1	1
**	selectQuery @(29--76)	selectQuery @(36--44)	1	1
**	jdbcLookupMapper @(29--76)	jdbcLookupMapper @(36--44)	1	1
**	columnName @(40--111)	columnName @(51--55)	1	1
**	columnName @(40--111)	columnName @(57--60)	1	1
**	connectionProvider @(37--266)	connectionProvider @(43--46)	1	1
**	queryTimeoutSecs @(37--266)	queryTimeoutSecs @(43--46)	1	1
**	outputFields @(24--59)	outputFields @(28--33)	1	1
**	schemaColumns @(27--92)	schemaColumns @(41--44)	1	1
**	jdbcLookupMapper @(112--155)	jdbcLookupMapper @(141--144)	1	1
**	connectionProvider @(112--155)	connectionProvider @(121--124)	1	1
**	tableName @(112--155)	tableName @(126--129)	1	1
**	insertQuery @(112--155)	insertQuery @(131--134)	1	1
**	selectQuery @(112--155)	selectQuery @(146--149)	1	1
**	queryTimeoutSecs @(112--155)	queryTimeoutSecs @(151--154)	1	1
*	connectionProvider @(27--96)	connectionProviderParam @(59--62)	0.782608696	0.853873
*	columnList @(70--73)	columnLists @(53--94)	0.909090909	0.9384209
	queryTimeoutSecs @(27--96)	connectionProviderParam @(59--62)	0.130434783	0.41746855
	insertQuery @(35--100)	queryTimeoutSecs @(65--68)	0.1875	0.6560837

Fig. 6 A zoom-in view of the part of confusing variable names in Fig. 5

Table 1 Data collection environment and hyperparameters

Item	Description
Processor	Apple M1
Memory	16GB
OS	macOS Ventura 13.0.1
Python	Python 3.9.15
repository collector	radon-repositories-collector 0.0.5
identifier splitter	Spiral 1.1.0
stemmer	PorterStemmer on nltk 3.7
spell checker	enchant 2.3.3
	pyenchant 3.2.2
Doc2Vec	Doc2Vec on gensim 4.2.0
	(vector_size=100, dm=0, min_count=1, epochs=400)

6.1 Aim

Although we have developed support tools for detecting confusing variable pairs automatically in the previous section, we still need to examine if the name similarities can contribute to detecting confusing variable pairs. Through an evaluation study with human participants regarding a variable pair's confusion level, we show the usefulness of the string and semantic similarity in distilling confusing variable pairs. Furthermore, we discuss the characteristics specific to confusing variable pairs.

In this study, we tackle the following two research questions (RQs).

- **RQ1: Are the string and semantic similarity scores helpful in detecting confusing variable pairs?**
  This is a fundamental question for our similarity quantification. We need to examine if our similarity evaluations are valid. We will answer this question through an evaluation study with human participants.
- **RQ2: What kind of characteristics do the detected confusing variable pairs have?**
  It is also helpful to understand the naming trends of the confusing variable names to prevent deterioration of the code readability caused by those variables. We will examine the names detected in the studied projects.

6.2 Data Collection and Evaluation Study

We performed our data collection under the computational environment shown in Table 1. The data collection procedure includes the following steps.

1. **Clone Git repositories.** To collect as many Java and Python Git repositories as possible, we utilized radon-repositories-collector 0.0.5 (Dalla Palma et al. 2021)[9], an open-source repository collector. The tool can explore public repositories in GitHub and collect metadata of those repositories, such as repository URLs. We searched repositories under the following conditions[10]:

[9] https://github.com/radon-h2020/radon-repositories-collector

[10] We performed that search on November 25th, 2022.

(1) Its primary development language is Java or Python.
(2) It has been pushed at least once after January 1st, 2020.
(3) Its "stars" score is greater than or equal to 1,000.

We set conditions (2) and (3) to filter out inactive and too-minor projects because the results derived from inactive/minor projects might not be attractive to practitioners. We made local copies of the found Git repositories from GitHub.

2. **Extract candidate sets of confusing variable pairs from Java projects and Python ones.** For Java source programs and Python ones in the collected projects, we parsed them and extracted variables with compound names. Then, we prepared the candidate set of confusing variable pairs described in Section 5.
3. **Train Doc2Vec models for each programming language.** We trained our Doc2Vec models of compound variable names. Because there may be a difference in naming trends between Java and Python, we built the models separately. The training dataset consists of the token sequences made by splitting and stemming the collected compound variable names[11] (see Section 4.2).
4. **Compute the string and semantic similarities of candidate variable pairs.** For each candidate pair of confusing variables, we computed the string similarity using the Levenshtein similarity (see Section 4.1) and the semantic similarity using the trained Doc2Vec model (see Section 4.2).

To answer the above RQs, we conducted an evaluation study with human participants. We asked 30 people (one colleague at Ehime University, 11 graduate students, and 18 undergraduate students at Ehime University or Okayama Prefectural University) to perform a qualitative evaluation regarding the perceived confusion of a given pair of variable names using a 5-point ordinal scale: –2, –1, 0, +1, and +2.

First, we e-mailed the participants to explain the aim of the study and the protection of personal information. In the mail, we conveyed the following points: (1) We had been studying the impact of confusing variable pairs on code readability and wanted to collect data regarding the perceived confusion of variable pairs. (2) A participant would take about 10–15 minutes to complete the task. (3) It is an unpaid task. (4) Although we collect their answers with their names (personal information), we anonymize and do not reveal them. (5) It is free to accept or reject the invitation for the study.

Next, the participants who accepted our invitation (the above 30 people) entered their names and gave their answers on our Web site. We did not give them any limit regarding the answer time and environment. All participants could answer anywhere and anytime without a time limit. The study site displayed variable pairs and pull-down options for answering their perceived confusion levels, as shown in Fig. 7. When the participants look at a pair of variable names, they answer "+2" if they feel a risk of mixing up those variables because their names are confusing. On the other hand, if they feel that a pair of names is easy to discriminate, they rank it as "–2." The remaining "–1," "0," and "+1" correspond to the intermediate levels from "easier" to "harder" to discriminate.

To pick samples to be evaluated by the participants, we performed stratified random sampling. We classified the variable pairs into 100 categories by their string and semantic similarities, as shown in Fig. 8. These 100 categories are from the combinations of a string similarity interval and a semantic similarity interval, where we consider ten string similarity intervals—(0, 0.1], (0.1, 0.2], . . . , (0.9, 1.0]—and ten semantic similarity

[11] We excluded duplicated names from our training dataset because they have different document ids with the identical content and may adversely affect the training process.

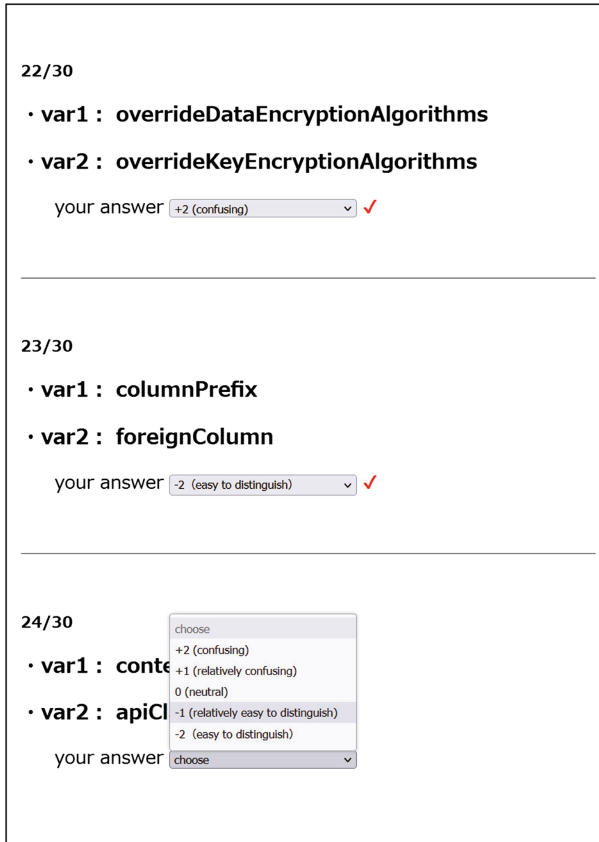


Fig. 7 A snapshot of the evaluation study Web page

intervals—$(-1, 0.1]$, $(0.1, 0.2]$, ..., $(0.9, 1.0]$. Although the first semantic similarity interval, $(-1, 0.1]$, has a broader range than the others, we combined the intervals of negative values with $(0, 0.1]$ because of the few instances. Then, for each participant, we randomly selected 30 categories out of 100 ones and singled out a random sample from each category to form the set of 30 variable pairs to be evaluated[12]. We asked the participants to evaluate two collections of variable pairs, i.e., one collection is 30 Java variable pair data, and another is 30 Python variable pair data. For a fair evaluation, we hid the category information of the sample variable pairs and shuffled the order of samples in the study.

6.3 Results

We report the results of the data collection and the evaluation study with human participants below.

[12] To minimize the participants' fatigue while collecting as many sample evaluations as possible, we limited the sample size to 30 in this study.

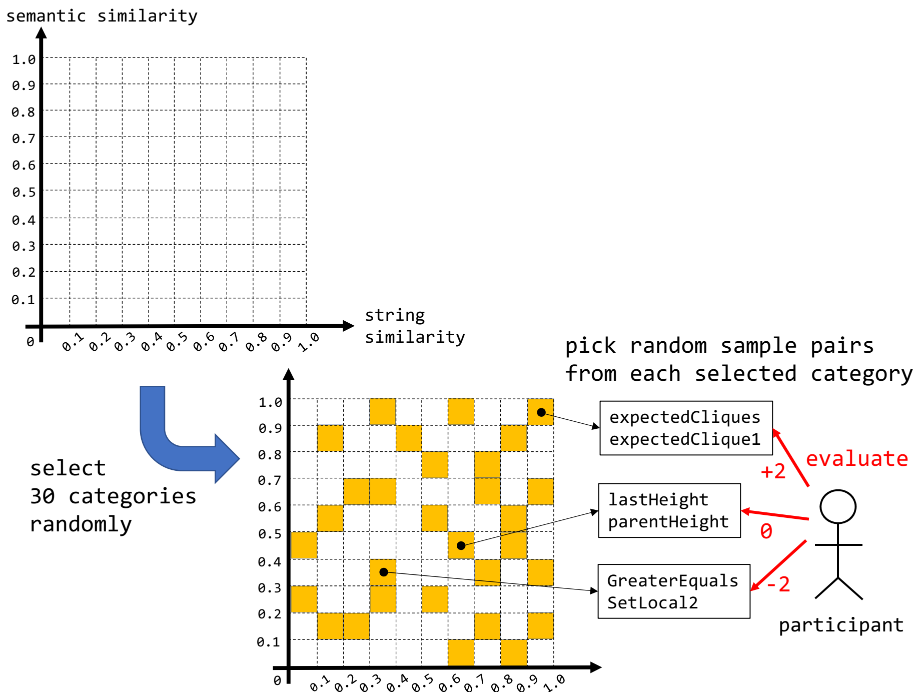


Fig. 8 Outline of random sampling in our evaluation study

6.3.1 Results of Data Collection

We obtained meta-data of 1,890 Java projects and 2,481 Python ones from GitHub using radon-repositories-collector 0.0.5 and analyzed their source files. As a result, we successfully analyzed 859,674 Java source files from 1,876 projects and 283,513 Python source files from 2,427 projects[13]. The remaining 14 Java and 54 Python projects corresponded to the following ones: (a) they have no Java/Python source files, or (b) our analysis program failed to parse the source files because multi-byte characters appeared in the variable names.

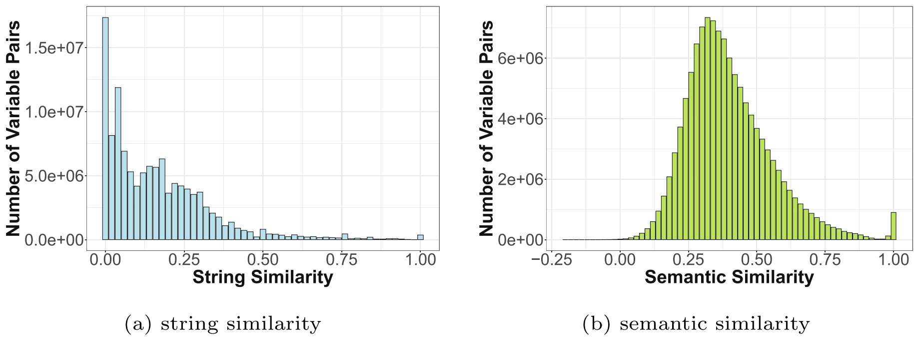
From the above Java source files, we collected 20,873,226 variables, and 8,333,176 (40%) were variables with compound names. Similarly, from the Python source files, we obtained 9,613,564 variables, and 3,256,524 out of them (34%) were compound-named ones. By checking their scopes, we found 116,921,126 pairs (in Java) and 106,943,523 pairs (in Python) as candidates for confusing variable pairs in this study.

We measured the string similarity and the semantic similarity for each pair. Figures 9, 10 and Tables 2, 3 show the distributions of similarity scores and the descriptive statistics. Because there are variable pairs where two names are the same[14], the maximum similarity is 1.0.

From the results of string similarity measurement, we can see that most variable pairs have relatively low similarity scores: the medians in Java and Python are 0.136 and 0.185, respectively. Both of them have right-skewed distributions. The results above say that not

[13] Our dataset is available from https://zenodo.org/record/7493554.

[14] For instance, one variable is a class's field, and another is a local variable in a method.



(a) string similarity (b) semantic similarity

Fig. 9 Histograms of similarity scores in Java

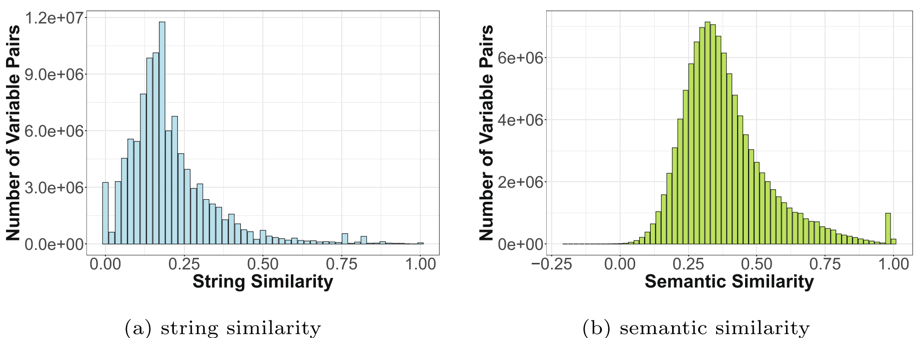
many variable pairs look confusing (highly similar) in both the Java and Python programs. We believe such trends are natural and consistent with our programming practice, i.e., we are likely to avoid using confusing variable pairs in our programs.

Although the semantic similarity seems to be higher than the string similarity in our collected dataset, we do not consider that there are many semantically similar variable pairs because their medians and 75 percentiles are around 0.4 (see Tables 2, 3). As shown in Figs. 9 and 10, the higher the semantic similarity gets, the fewer the number of variable pairs. Notice that although there are more variable pairs with high semantic similarity scores around 1.0 than the string similarity results (see Figs. 9 and 10), they include not only the same name pairs but the different numbered pairs, such as "`valuesToArray1`" and "`valuesToArray2`."

6.3.2 Results of Evaluation Study with Human Participants

As a result of the evaluation study with human participants, we obtained 839 evaluation samples of Java variable pairs and 869 evaluation samples of Python ones from 30 participants. Because three participants answered only Java or Python data, and two participants made two missing values (blank answers) by mistake, the number of collected samples was not 900 (=30×30) for each language.

Tables 4 and 5 show the counts of variable pair samples by their confusion level evaluations. Due to the stratified random sampling, we could get results dispersed from –2 to +2, but their distributions were not uniform. The samples with evaluation "0" (judged as "neutral") were



(a) string similarity (b) semantic similarity

Fig. 10 Histograms of similarity scores in Python

Table 2 Descriptive statistics of similarity scores in Java

	Min	25%	50%	Mean	75%	Max
String	0.000	0.045	0.136	0.172	0.250	1.000
Semantic	–0.129	0.300	0.382	0.407	0.491	1.000

Table 3 Descriptive statistics of similarity scores in Python

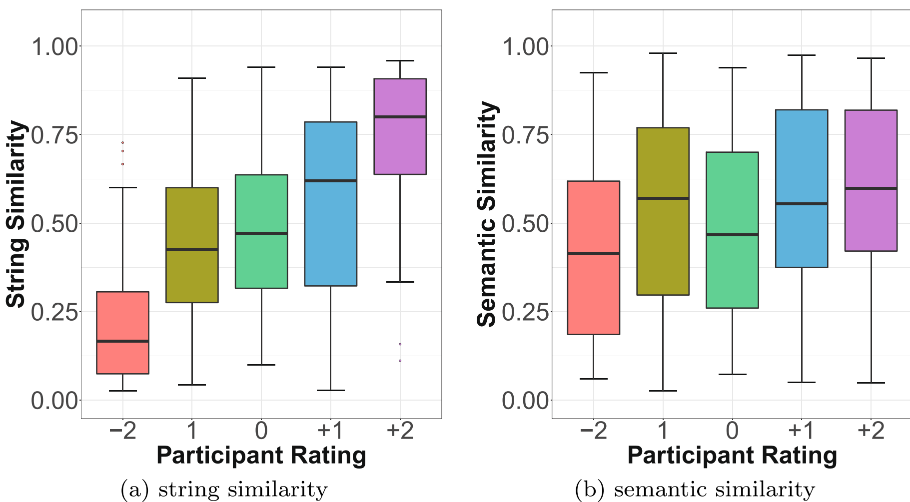
	Min	25%	50%	Mean	75%	Max
String	0.000	0.130	0.185	0.213	0.261	1.000
Semantic	–0.147	0.288	0.364	0.394	0.466	1.000

Table 4 Counts of variable pair samples by their confusion level evaluations (Java)

	Confusion level evaluation					
	–2	–1	0	+1	+2	Total
#Samples	197 (23.5%)	197 (23.5%)	80 (9.5%)	226 (26.9%)	139 (16.6%)	839

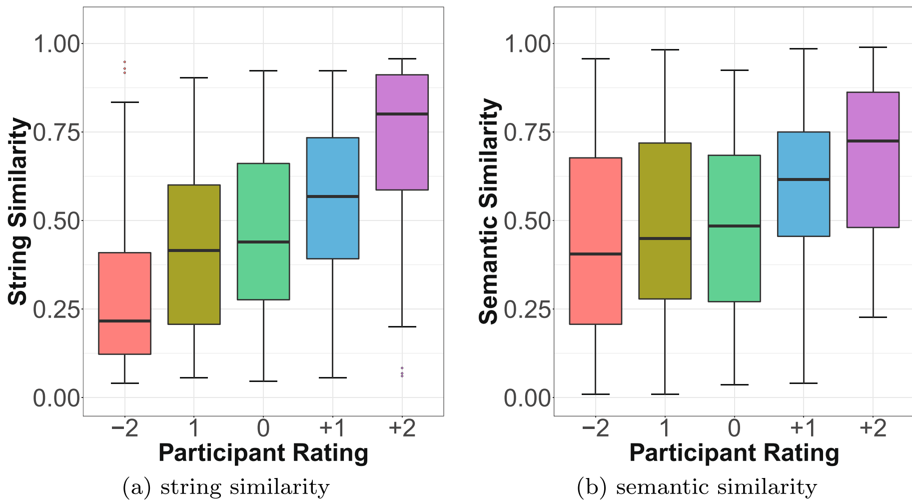
Table 5 Counts of variable pair samples by their confusion level evaluations (Python)

	Confusion level evaluation					
	–2	–1	0	+1	+2	Total
#Samples	215 (24.7%)	241 (27.7%)	98 (11.3%)	190 (21.9%)	125 (14.4%)	869



(a) string similarity (b) semantic similarity

Fig. 11 Boxplots of similarity scores vs. participant ratings in Java



(a) string similarity (b) semantic similarity

Fig. 12 Boxplots of similarity scores vs. participant ratings in Python

the minorities (9.5% and 16.3%) in the results of both languages. Hence, it seems not too hard to discriminate "confusing"(+2) or "relatively confusing"(+1) variable pairs from "non-confusing (easy to distinguish)"(–1 or –2) pairs for the participants in our study.

Figures 11 and 12 show the boxplots of similarity scores by the participant ratings. For both languages, the higher string similarity tends to be observed in the more confusing variable pair samples (see Figs. 11(a) and 12(a)). Thus, the string similarity scores seem to correspond to the human evaluations regarding the confusion levels. On the other hand, the semantic similarity scores are less likely to link to the participant ratings. Although the results in both languages have relatively increasing trends in Figs. 11(b) and 12(b), their trends are weaker than the results of string similarity scores. Nonetheless, we could observe the common trend that the set of variable pair samples with the evaluation "+2" tend to have high similarity scores in terms of both string and semantic similarity.

6.4 Discussion

We discuss the results of our data collection and evaluation study from the perspectives of our RQs mentioned above.

6.4.1 RQ1: Are the String and Semantic Similarity Scores Helpful in Detecting Confusing Variable Pairs?

We conducted an evaluation study of the variable pair's confusion level with 30 human participants to examine if our name similarity metrics are practical. Figures 11(a) and 12(a) show positive correlative relationships between the variable name pair's string similarity score and the participant's evaluation regarding the confusion level of the pair—the Spearman's ρ between the string similarity score and the participant's rating are 0.565 ($p < 2.2 \times 10^{-16}$) in Java and 0.482 ($p < 2.2 \times 10^{-16}$) in Python. Although such increasing trends in the semantic similarity scores are weaker than the string ones—$\rho = 0.212$ ($p = 1.258 \times 10^{-5}$) in Java and $\rho = 0.299$ ($p = 4.298 \times 10^{-10}$) in Python, we also see that the name pairs with

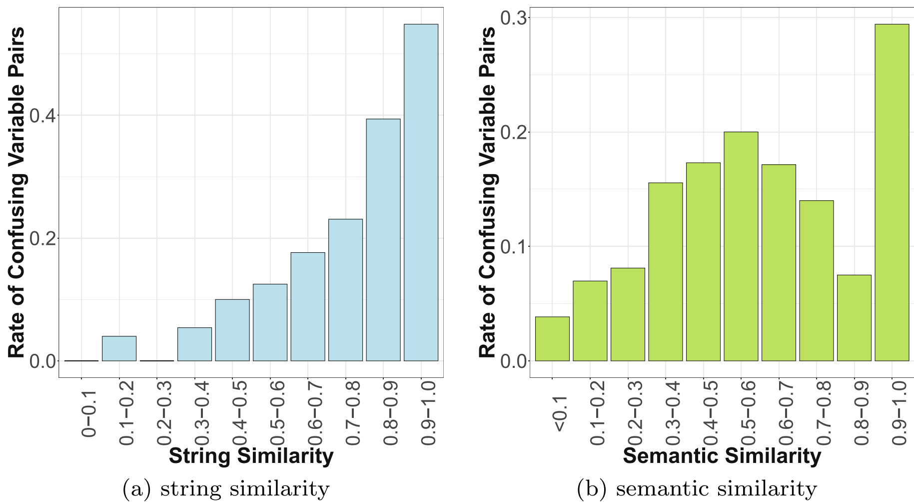


Fig. 13 Rates of confusing variable pairs over the similarity scores in Java

human evaluation "+2" tend to have relatively high semantic similarity scores (Figs. 11(b) and 12(b)).

We analyze the evaluation data from another perspective: the rate of "+2" evaluations (judged as "confusing variable pair"). For instance, through the evaluation study, we obtained 31 evaluations for Java variable name pairs whose string similarity scores are within (0.9, 1.0], and 17 out of 31 pairs were evaluated as "+2". Then, the rate is approximately 0.55 (=17/31). The higher the rate is, the more noteworthy the similarity scores corresponding to the range become, in detecting confusing variable pairs. Figures 13 and 14 show the rates of "+2" evaluations for both string and semantic similarity in both languages. From the figures, we see increasing trends in the rate of confusing variable pairs over the string similarity in Java

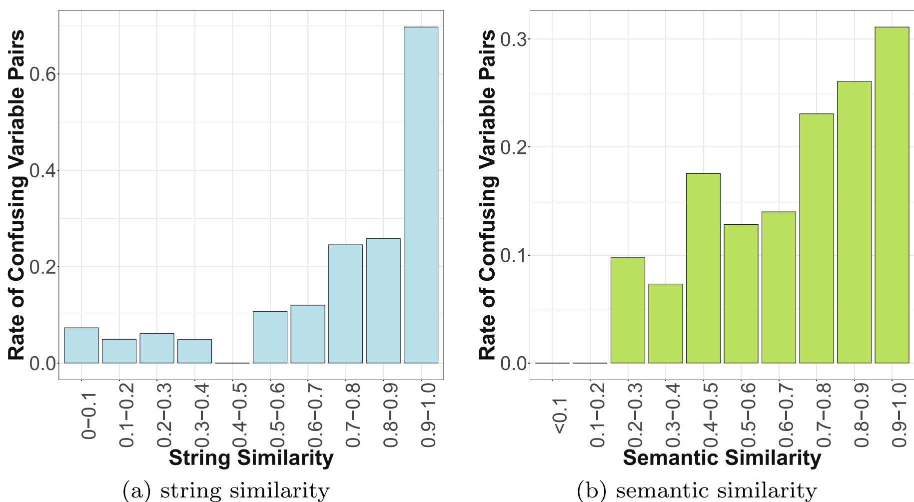


Fig. 14 Rates of confusing variable pairs over the similarity scores in Python

and both string and semantic similarity in Python. Although we did not observe such an increasing trend over the semantic similarity in Java (see Fig. 14(b)), it might be affected by the participants' programming backgrounds. Because the majority of the study participants have learned programming skills in C and Python, they might not be familiar with camelCase-styled compound names in Java. Nonetheless, the rate of confusing variable pairs in Java gets the highest when the pairs' semantic similarity scores are within (0.9, 1.0], so we consider both similarity scores to help detect confusing variable pairs in both languages.

Answer to RQ1

We give our answer to RQ1—*Are the string and semantic similarity scores helpful in detecting confusing variable pairs?*—as:

> Both string and semantic similarity scores helpfully work for detecting confusing variable pairs in Java and Python programs. When a variable name pair has a relatively high string or semantic similarity, human programmers are likely to consider it to be confusing and feel a risk of mixing up the variable names.

6.4.2 RQ2: What Kind of Characteristics do the Detected Confusing Variable Pairs Have?

To answer our RQ2, we need to obtain samples of "confusing variable pairs." Thus, we begin by exploring name similarity thresholds to filter confusing variable pairs. As we have seen the relationships between the similarity scores and the rates of confusing variable pairs in Figs. 13 and 14, we can expect helpful thresholds to be around 0.9 for both string and semantic similarity scores. We will conduct preliminary work on the thresholds below.

First, we check the correlation between two similarity measures—string and semantic—to reasonably treat them in finding thresholds. We examined the correlation between the similarity scores using Pearson's r and confirmed moderate (relatively strong) positive correlations between the two similarity scores: $r = 0.688$ ($p < 2.2 \times 10^{-16}$) in Java and $r = 0.651$ ($p < 2.2 \times 10^{-16}$) in Python. Such a positive correlation relationship would be natural because:

- if two compound names have high string similarity, they may share some words in their names, resulting in high semantic similarity;
- however, even if two compound names have high semantic similarity, they might not use identical words.

Although the two similarity scores have a positive correlation, they have different points of view regarding the name similarity, and we cannot say one of them is redundant. Hence, we will use both similarities to explore the criteria to distill confusing variable pairs below. We denote the thresholds for string and semantic similarity by τ_{str} and τ_{sem}, then consider our criteria as follows.

- Is the string similarity higher than τ_{str}?, or
- Is the semantic similarity higher than τ_{sem}?

Next, we consider the binary classification of variable pairs—*confusing or not*—using the above criteria while changing the thresholds. In the classification, the positive instances are the ones with human evaluation "+2", and the negative instances are the others. We computed the recall, precision, and F1 value (the harmonic mean of recall and precision) for each set

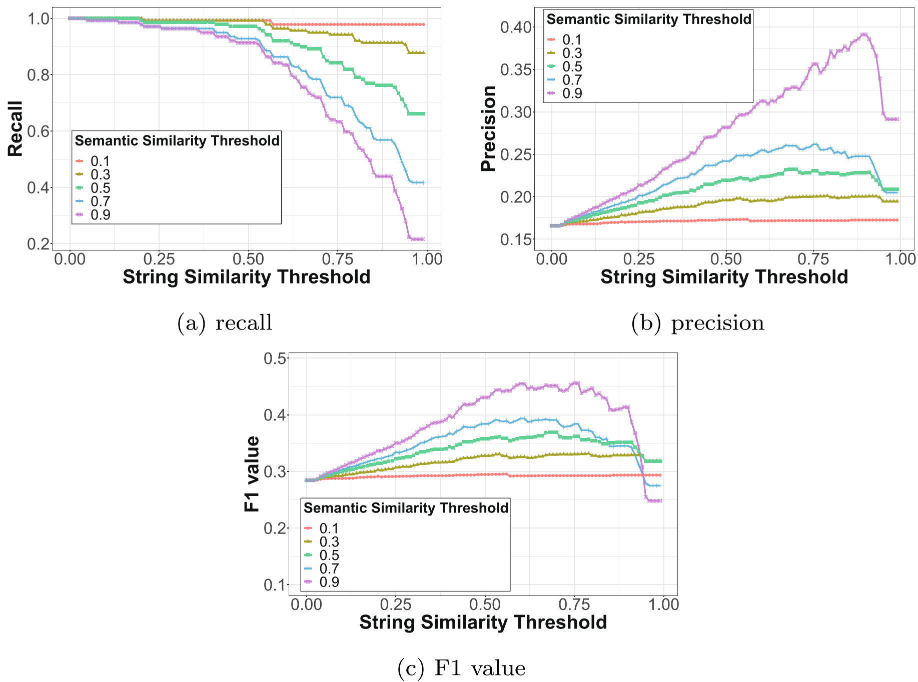


Fig. 15 Recall, precision, and F1 values of the binary classification of confusing variable pairs using various thresholds in Java

of thresholds while changing the thresholds from 0.00 to 0.99 in 0.01 increments. Figures 15 and 16 present parts of the results: we show only the computational results when we set the semantic similarity thresholds to 0.1, 0.3, 0.5, 0.7, and 0.9 to express the curves' trends in those figures because all results form 100 curves in the charts.

From the figures, we see the trends that the recall values decrease monotonically as both string and semantic similarity thresholds get higher in both languages. Although high values of both similarity measures correlate with the rate of confusing variable pairs, the risk of overlooking those pairs (positive instances) seems to rise when we set around 0.9 or higher thresholds to the string similarity (see Figs. 15(a) and 16(a)). The precision values show the tendencies that they increase as both string and semantic similarity thresholds become higher and higher. Thus, we may detect confusing variable pairs with higher confidence when we set the thresholds to higher values. However, those precision values also drop when we set the string similarity threshold to around 0.9 or higher (see Figs. 15(b) and 16(b)). To examine the reason, we explored the evaluation results. Then, we found that some participants evaluated the following samples as not confusing pairs:

- "`csObservationMethod`" vs. "`vsObservationMethod`" (string similarity = 0.947), and
- "`mRedditDataRoomDatabase`" vs. "`redditDataRoomDatabase`" (string similarity = 0.913).

Although their string similarity scores were high, the participants did not regard them as confusing pairs because their initial letters differed. In our quantification context, two long names with common tokens may have a high string similarity score because we normalize

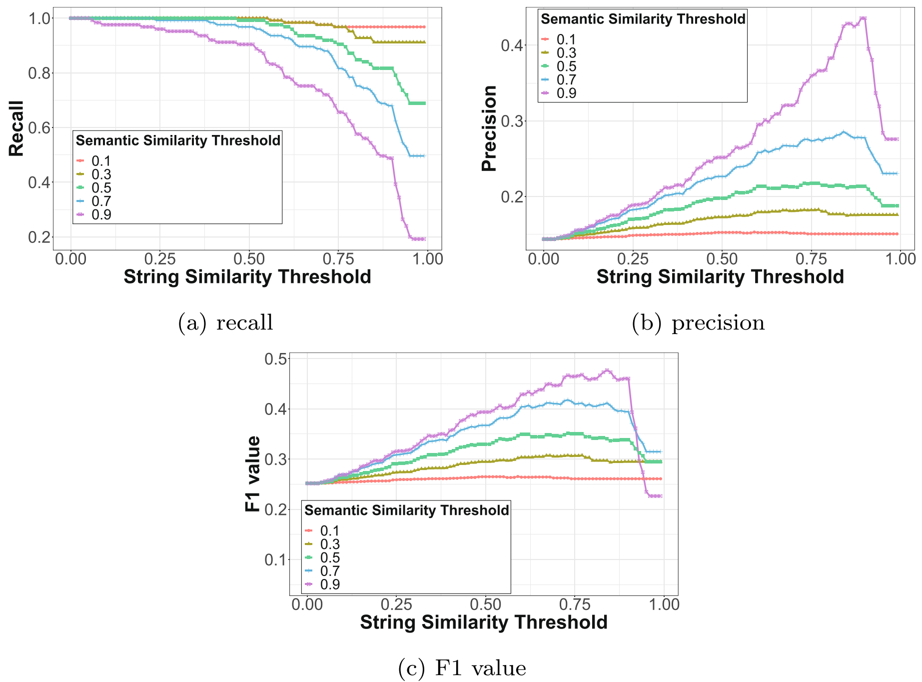


Fig. 16 Recall, precision, and F1 values of the binary classification of confusing variable pairs using various thresholds in Python

Table 6 Proposed thresholds

Language	String similarity (τ_{str})	Semantic similarity (τ_{sem})
Java	0.75	0.96
Python	0.84	0.95

Table 7 Number of variable pairs by similarity category in Java

		String similarity	
		$> \tau_{str}$	$\leq \tau_{str}$
Semantic Similarity	$> \tau_{sem}$	1,007,963 (0.86%)	32,991 (0.03%)
	$\leq \tau_{sem}$	583,287 (0.50%)	115,296,885 (98.61%)

Table 8 Number of variable pairs by similarity category in Python

		String similarity	
		$> \tau_{str}$	$\leq \tau_{str}$
Semantic Similarity	$> \tau_{sem}$	300,935 (0.28%)	917,025 (0.86%)
	$\leq \tau_{sem}$	120,169 (0.11%)	105,605,394 (98.75%)

Table 9 Examples of confusing variable pairs in Java

No.	Variable name pair	Similarity String	Similarity Semantic
1	`expectedFormattedResultsList` `expectedFormattedResultsPColl`	0.828	0.340
2	`githubComKubernetesSigsServiceCatalogPkgApis Servicecatalog V1beta1ServiceClass` `githubComKubernetesSigsServiceCatalogPkgApis Servicecatalog V1beta1ServiceInstance`	0.913	0.966
3	`someString722` `someString872`	0.846	0.583
4	`currentP1Stages` `currentP3Stages`	0.933	0.810
5	`byteCodeAppenders` `byteCodeAppender`	0.941	0.220
6	`typeVariableAnnotationTokens` `stypeVariableBoundAnnotationTokens`	0.848	0.922
7	`variablePattern` `patternVariable`	0.067	0.972
8	`FIELD_TARGET_FIELD` `targetField`	0.056	0.989

the Levenshtein distance by the name length. Hence, such cases would cause a decrease in precision values at around 0.9 thresholds in Figs. 15(b) and 16(b).

By comparing the F1 values—the harmonic mean of the recall and the precision values—(see Figs. 15(c) and 16(c)), we derived the best combinations of thresholds from our evaluation results (Table 6): τ_{str} = 0.75 and τ_{sem} = 0.96 for Java, and τ_{str} = 0.84 and τ_{sem} = 0.95 for Python.[15]

Using the above criteria, we can classify the pairs of compound-named variables into four categories shown in the Tables 7 and 8. As a result, 1,007,963 (0.86%) pairs and 300,935 (0.28%) pairs have higher similarities than both string and semantic thresholds in Java and Python programs; 1,624,241 (1.39% = 0.86 + 0.03 + 0.50%) pairs and 1,338,129 (1.25% = 0.28 + 0.86 + 0.11%) pairs have higher similarities than either the string or semantic thresholds in Java and Python programs. Our support tools can detect those confusing variable pairs and warn the former variable pairs by mark "**" and the latter ones by mark "*" (see Fig. 5 in Section 5.4).

Next, to find clues for answering RQ2, we thoroughly examine the evaluation study samples. Tables 9 and 10 show instances from our data of Java and Python variable pairs.

From these examples, we surmise that confusing variable pairs have one of the following five characteristics: in a variable name pair, (1) only their tailing tokens differ (e.g., No.1, 2, 9, 10 in Tables 9 and 10), (2) only their numbered parts differ (e.g., No.3, 4, 11, 12), (3) a name is a substring of the other's name (e.g., No.5, 13, 14), (4) a name's token set is a subset of the other name's token set (e.g., No.6, 8), or (5) a name can be converted into the other name by changing the order of tokens (e.g., No.7, 15, 16), where we treat the tokens in a case-insensitive manner after the stemming preprocess.

[15] Notice that Figs. 15(c) and 16(c) present the results of τ_{sem} = 0.9 but not τ_{sem} = 0.95 and 0.96 to avoid the charts becoming too complicated.

Table 10 Examples of confusing variable pairs in Python

No.	Variable name pair	Similarity	
		String	Semantic
9	content_ii content_iii	0.909	0.579
10	RELATIVE_POSITION_II_LG RELATIVE_POSITION_II_BI	0.913	0.763
11	res4b20_branch2a res5b_branch2a	0.813	0.984
12	source_options_site_coll_model_json source_options_site_coll_model_json2	0.972	0.922
13	add_one_udfs add_one_udf	0.929	0.804
14	eigBlockVector eigBlockVectorX	0.933	0.912
15	frame_num num_frames	0.200	0.988
16	TASK_TYPES_TO_STRING STRING_TO_TASK_TYPES	0.200	0.987

We will examine the above surmises using our variable pair dataset (not limited to the evaluation study samples) below. Here, we define a "confusing variable pair" as a pair whose string and semantic similarity scores are higher than the thresholds shown in Table 6.

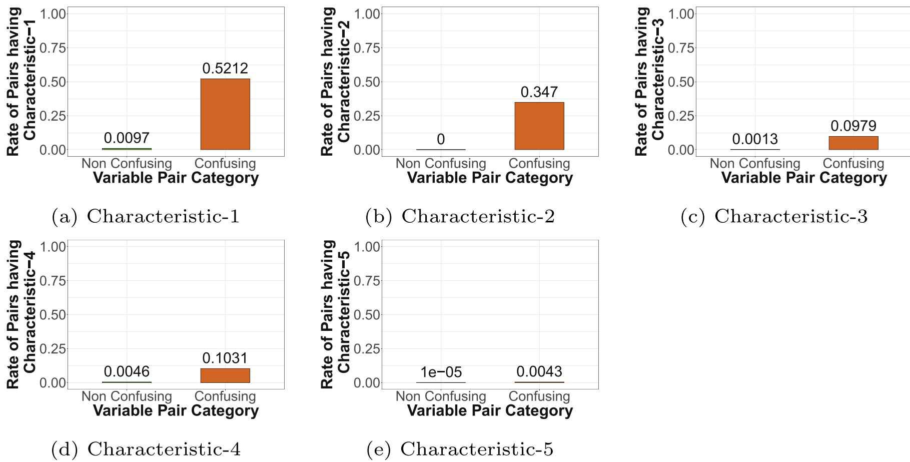
Characteristic-1: Only the Tailing Tokens Differ

We checked all compound variable pairs in our collected dataset and compared their token sequences. Then, we counted the pairs having Characteristic-1. For example, the pair of "`textFieldMap`" and "`textFieldList`" has that characteristic because only the tailing tokens—"`map`" and "`list`"—differ.

As a result, about 52.1% (=846,641/1,624,241) of confusing variable pairs in Java and about 86.4% (=1,156,308/1,338,129) in Python had Characteristic-1. Although we also found not confusing variable pairs having the characteristic, their percentages were only about 1.0% (=1,114,098/115,296,885) in Java and about 0.7% (=715,151/105,605,394) in Python (see Figs. 17(a) and 18(a)). Hence, we can say that confusing variable pairs tend to have Characteristic-1.

Characteristic-2: Only Their Numbered Parts Differ

To check if many confusing variable pairs have Characteristic-2, we counted those pairs in our dataset. A pair of "`asyncResult1`" and "`asyncResult2`" is a typical example. As a result, about 34.7% (563,582 confusing variable pairs) in Java and about 77.1% (1,031,958 confusing variable pairs) in Python had Characteristic-2 (see Figs. 17(b) and 18(b)). Notice that some variable pairs also had Characteristic-1 when their tailing tokens were numbers, so some variable pairs were counted for both Characteristics 1 and 2. On the other hand, we did not find non-confusing variable pairs having Characteristic-2 because the name pairs with Characteristic-2 have high string similarity scores, and all of them fall into the con-



(a) Characteristic-1 (b) Characteristic-2 (c) Characteristic-3

(d) Characteristic-4 (e) Characteristic-5

Fig. 17 Rates of variable pairs having characteristics 1–5 in Java

fusing variable pair category.[16] Consequently, we can say that confusing varable pairs have Characteristic-2.

Characteristic-3: a Name is a Substring of the Other Name

This characteristic also looks typical for confusing variable pairs because such names tend to have more common parts and be harder to distinguish. For example, the pair of "`fileNameList`" and "`fileNameLists`" corresponds to this characteristic. As a result of our counting, we found about 9.8% (159,006) of confusing variable pairs in Java and about 3.8% (50,806) of confusing variable pairs having Characteristic-3. About 0.1% of non-confusing variable pairs in Java and about 0.2% in Python were also found in our dataset. Because the percentages of confusing variable pairs with Characteristic-3 are significantly fewer than Characteristics 1 and 2 (see Figs. 17(c) and 18(c)), we may not say that confusing variable pairs have Characteristic-3 confidently. However, the percentages of non-confusing variable pairs are low (0.1–0.2%), so we do not need to neglect the characteristic.

Characteristic-4: a Name's Token Set is a Subset of the Other Name's Token Set

This characteristic is an extension of Characteristic-3 to token level rather than character level. An example pair having Characteristic-4 is "`callsReceivedRow`" and "`callsBytesReceivedRow`" because all of the tokens in the former name, "`calls`," "`received`," and "`row`" are included in the latter name. As a result, we discovered that about 10.3% (167,503) of confusing variable pairs in Java and about 6.8% (90,687) of them in Python had the characteristic (see Figs. 17(d) and 18(d)). Only about 0.5% and 0.3% of non-confusing variable pairs in Java and Python had that characteristic. Similar to

[16] Notice that we regarded a variable name as a compound name when we could split the original name into two or more English dictionary words or abbreviations consisting of three or more characters (see Section 4.2). Thus the variables with short and numbered names like "`dx1`" and "`dx2`" (their string similarity is about 0.667) were excluded from our dataset.

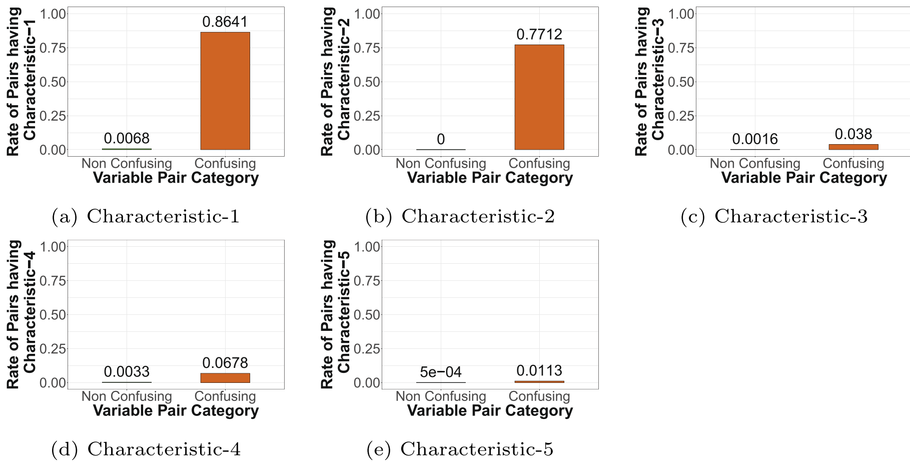


Fig. 18 Rates of variable pairs having characteristics 1–5 in Python

Characteristic-3, although this characteristic was also not observed in the majority of confusing variable pairs, the percentages of non-confusing ones are still low. Thus, we may also consider Characteristic-4 as a part of confusing variable pairs' characteristics.

Characteristic-5: a Name can be Converted into the Other Name by Changing the Order of Tokens

Finally, we checked the cases in which two compound names consist of the same token set, but their orders differ. The pair of "`variablePattern`" and "`patternVariable`" is an example. Given such a pair, programmers and code reviewers might mix them up during the programming and code review activities. We checked our dataset and found that such pairs rarely appear in real: the percentages of confusing variable pairs with Characteristic-5 were about 0.4% (6,992) in Java and about 1.1% (15,073) in Python (see Figs. 17(e) and 18(e)). Hence, although the variable pairs having the characteristic look confusing, we cannot say it is a common characteristic of confusing variable pairs.

From the above results, we consider the confusing variable pairs would mainly have Characteristics 1–4. By eliminating the duplicated counts, we confirmed that about 57.4% of confusing variable pairs in Java and about 89.4% of them in Python had one of the above four characteristics. Notice that we did not count the variable pairs consisting of two identical names (e.g., a pair of a local variable and a class field) for the above characteristics, and their percentages in confusing variable pairs were about 22.3% and 4.9% in Java and Python. Thus, we can cover the majority of confusing variable pairs by the pairs with the four characteristics and the identical name pairs: 79.7% (=57.4% + 22.3%) in Java and 94.3% (=89.4% + 4.9%) in Python.

Answer to RQ2

We answer to RQ2—*What kind of characteristics do the detected confusing variable pairs have?*–as:

Through examining the results of the evaluation study with human participants and collected variable pair data, we found four characteristics specific to the detected confusing variable pairs: (1) they share most tokens, and only their tailing tokens differ, (2) they share most parts, and only their numbered parts differ, (3) a name is a substring of the other name, or (4) a name's token set is a subset of the other name's token set. Those characteristics would be common to both Java and Python programs. When we encounter such variables with similar names, we should reconsider the names to avoid making a confusing variable pair.

6.5 Threats to Validity

6.5.1 Internal Validity

Although we conducted an evaluation study with human participants, most of them were Japanese students. Because they are not native English speakers, their evaluation of semantic aspects of variable names might be insufficient. Moreover, their programming background would threaten our study's internal validity: most students have learned programming skills in C and Python. A further evaluation study with various backgrounded people is our future work.

There are other threats to internal validity in the design of our evaluation study with human participants. In the study, each variable pair was assessed by a single participant independently. The results might highly depend on their background and preference regarding the variable naming. Different assessment systems involving two or more participants, such as discussion, voting, or averaging, may lead to different study results. Moreover, although we asked a participant to assess the confusion level of the pair of variable names displayed on the study Web site, it did not present the surrounding code. If the participant has deep knowledge of the program domain and can see not just the variable names but the surrounding code, he/she may assess the confusion level differently. We need to conduct a further study with skilled developers while showing the context in which the variables are used as our significant future work.

We measured the semantic similarity using the Doc2Vec model. Because its performance depends on the hyperparameters, their setting can be a threat to internal validity. To mitigate this threat, we preliminary experimented with various combinations of two major hyperparameters, `vector_size` and `epochs`. Because the Doc2Vec model steadily produced almost the same vectors for the same names when we set `vector_size=100` and `epochs=400`, we used them in our study. However, there might be a better setting, and we might miss a more proper model for evaluating semantic similarity. Thus, we also tried analyzing the semantic similarity of compound variable names using the state-of-the-art natural language model, Sentence-BERT (Reimers and Gurevych 2019), described in Section 4.2. Although Sentence-BERT models did not work well in our settings, we plan to perform an additional fine-tuning process in our future work.

6.5.2 Construct Validity

We adopted the Levenshtein similarity as our measure of the string similarity. Moreover, we used the cosine similarity between the document vectors obtained by Doc2Vec, to evaluate the semantic similarity. Although these measures have been widely used, there might be other better ways of measuring those similarities.

6.5.3 External Validity

Although we conducted a large-scale investigation and analyzed many variable names in various programs, it is limited to open-source Java and Python projects. Other projects using other languages or commercial projects may lead to different results. Because there are language-specific features or practices in naming variables, we plan to perform other investigations on other language projects in the future.

We have experimentally presented thresholds of similarity scores to distill confusing variable pairs through the data analysis and used them as the default values in our tool. However, they may not be generalizable to all Java/Python programs because the number of study participants (sample size) was only 30, and most participants were students. A further study with more developers (not students) would be required to mitigate the threat.

7 Conclusion and Future Work

In this paper, we focused on compound variable names appearing in programs. Programmers sometimes declare variables with compound names to express the variables' roles clearly. Although compound names themselves are easy to understand, there is a risk that they can be confusing variable pairs when their names are highly similar. Because the confusing variable pairs may adversely affect the code readability, it is better to detect such pairs automatically. We conducted a large-scale investigation of compound names in Java and Python programs and an evaluation study with human participants to examine if the name similarities contribute to distilling confusing variable pairs.

Our study collected 116,921,127 pairs of compound-named variables from 1,876 open-source Java projects and 106,943,523 pairs from 2,427 open-source Python projects. Then, we measured their name similarity from two perspectives: string similarity and semantic similarity. We used the Levenshtein distance and the Doc2Vec to measure those similarities. The evaluation study with human participants and data analysis have shown that the name similarities help detect confusing variable pairs in Java and Python programs. Moreover, we saw four primary characteristics specific to confusing variable pairs: (a) they share most tokens, and only their tailing tokens differ, (b) they share most parts, and only their numbered parts differ, (c) a name is a substring of the other name, and (d) a name's token set is a subset of the other name's token set. When we encounter such variables with similar names, it is better to reconsider their names to enhance the code readability from the perspective of ease of identifying and distinguishing those variables. Furthermore, we developed support tools for Java and Python programs to detect confusing variable pairs automatically using our data analysis results. Note: Our tools and dataset are available from `https://github.com/amanhirohisa/cvpfinder` and `https://zenodo.org/record/7493554`.

Our significant future work is a further study on the relationship of the variable names' confusion level with program comprehension. Because this paper focuses only on the confusing variable pairs, we have not analyzed the impact of confusing variables on the program comprehension task yet. We plan to perform further studies of program comprehension from the perspective of variable names. Other future work includes (1) further investigations on other software projects using languages other than Java and Python, (2) considering the ambiguity of variable names from the perspective of linguistic antipatterns (Arnaoudova et al. 2016),

and (3) building a predictive model using similarity scores that can predict the variable name confusion level.

Acknowledgements This work was supported by JSPS KAKENHI #20H04184, #21K11831, and #21K11833

Data Availability The dataset used in the present study is available from https://zenodo.org/record/7493554

Declarations

Funding and/or Conflicts of interests/Competing interests The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Aivaloglou E, Hermans F (2016) How kids code and how we know: An exploratory study on the scratch repository. In: Proceedings of the 2016 ACM Conference on International Computing Education Research, ACM, New York, NY, USA, ICER '16, pp 53–61. https://doi.org/10.1145/2960310.2960325, http://doi.acm.org/10.1145/2960310.2960325

Aman H, Amasaki S, Yokogawa T, Kawahara M (2019) Empirical study of fault introduction focusing on the similarity among local variable names. In: Proc. 7th Int. Workshop Quantitative Approaches to Softw. Quality, pp 3–11

Aman H, Amasaki S, Yokogawa T, Kawahara M (2021a) An investigation of compound variable names toward automated detection of confusing variable pairs. In: Proc. 36th IEEE/ACM International Conference on Automated Software Engineering Workshops, pp 133–137. https://doi.org/10.1109/ASEW52652.2021.00036

Aman H, Amasaki S, Yokogawa T, Kawahara M (2021) A large-scale investigation of local variable names in java programs: Is longer name better for broader scope variable? In: Pérez-Castillo R (ed) Paiva ACR, Cavalli AR, entura Martins P. Quality of Information and Communications Technology, Springer International Publishing, Cham, pp 489–500

Arnaoudova V, Di Penta M, Antoniol G (2016) Linguistic antipatterns: what they are and how developers perceive them. Empir Softw Eng 21(1):104–158. https://doi.org/10.1007/s10664-014-9350-8

Beniamini G, Gingichashvili S, Orbach AK, Feitelson DG (2017) Meaningful identifier names: The case of single-letter variables. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp 45–54. https://doi.org/10.1109/ICPC.2017.18

Binkley D, Lawrie D, Maex S, Morrell C (2009) Identifier length and limited programmer memory. Sci Comput Program 74(7):430–445. https://doi.org/10.1016/j.scico.2009.02.006 www.sciencedirect.com/science/article/pii/S0167642309000343

Binkley D, Davis M, Lawrie D, Maletic JI, Morrell C, Sharif B (2013) The impact of identifier style on effort and comprehension. Empir Softw Eng 18(2):219–276. https://doi.org/10.1007/s10664-012-9201-4

Caprile, Tonella (2000) Restructuring program identifier names. In: Proceedings of 2000 International Conference on Software Maintenance, pp 97–107. https://doi.org/10.1109/ICSM.2000.883022

Ceccato M, Di Penta M, Falcarin P, Ricca F, Torchiano M, Tonella P (2014) A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empir Softw Eng 19(4):1040–1074. https://doi.org/10.1007/s10664-013-9248-x

Deissenboeck F, Pizka M (2006) Concise and consistent naming. Softw Q J 14(3):261–282. https://doi.org/10.1007/s11219-006-9219-1

Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proc. 2019 Conf. the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, vol 1, pp 4171–4186

Free Software Foundation (2018) GNU coding standards. https://www.gnu.org/prep/standards/standards.html

Gosling J, Joy B Jr, GLS, Bracha G, Buckley A (2014) The Java Language Specification. Addison-Wesley, Boston, MA

Gusfield D (1997) Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge

Hucka M (2018) Spiral: splitters for identifiers in source code files. J Open Source Software 3(24):653. https://doi.org/10.21105/joss.00653

Kernighan BW, Pike R (1999) The Practice of Programming. Addison-Wesley, Boston, MA

kernel development community T (2016) Linux kernel coding style. https://www.kernel.org/doc/html/v4.10/process/coding-style.html

Knuth DE (2003) Selected Papers on Computer Languages. No. 139 in CSLI Lecture Notes, Center for the Study of Lang. & Inf., Stanford, California

Lacomis J, Yin P, Schwartz EJ, Allamanis M, Goues CL, Neubig G, Vasilescu B (2019) DIRE: A neural approach to decompiled identifier naming. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, pp 628–639. https://doi.org/10.1109/ASE.2019.00064

Lawrie D, Morrell C, Feild H, Binkley D (2007) Effective identifier names for comprehension and memory. Innov Syst Softw Eng 3(4):303–318. https://doi.org/10.1007/s11334-007-0031-2

Le QV, Mikolov T (2014) Distributed representations of sentences and documents. CoRR abs/1405.4053

Liblit B, Begel A, Sweetser E (2006) Cognitive perspectives on the role of naming in computer programs. In: Proc. 18th Annual Psychology of Programming Workshop, pp 53–67

Low D (1998) Protecting java code via code obfuscation. Crossroads 4(3):21–23. https://doi.org/10.1145/332084.332092

Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. CoRR abs/1301.3781

Palma SD, Di Nucci D, Tamburri D (2021) RepoMiner: a language-agnostic python framework to mine software repositories for defect prediction. https://doi.org/10.48550/arXiv.2111.11807 https://arxiv.org/abs/2111.11807

Pigoski TM (1996) Practical Software Maintenance: Best Practices for Managing Your Software Investment, 1st edn. Wiley Publishing, N.J

Reimers N, Gurevych I (2019) Sentence-BERT: Sentence embeddings using siamese BERT-networks. In: Proc. 2019 Conf. Empirical Methods in Natural Language Processing and 9th Int'l Joint Conf. Natural Language Processing, pp 3982–3992, https://doi.org/10.18653/v1/D19-1410

Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2021) Automatically assessing code understandability. IEEE Trans Softw Eng 47(3):595–613. https://doi.org/10.1109/TSE.2019.2901468

Scanniello G, Risi M, Tramontana P, Romano S (2017) Fixing faults in c and java source code: Abbreviated vs. full-word identifier names. ACM Trans Softw Eng Methodol 26(2):6:1–6:43. https://doi.org/10.1145/3104029

Schankin A, Berger A, Holt DV, Hofmeister JC, Riedel T, Beigl M (2018) Descriptive compound identifier names improve source code comprehension. In: Proc. 26th Int. Conf. Program Comprehension, pp 31–40. https://doi.org/10.1145/3196321.3196332

Swidan A, Serebrenik A, Hermans F (2017) How do scratch programmers name variables and procedures? In: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 51–60. https://doi.org/10.1109/SCAM.2017.12

Tashima K, Aman H, Amasaki S, Yokogawa T, Kawahara M (2018) Fault-prone java method analysis focusing on pair of local variables with confusing names. In: Proc. 44th Euromicro Conf. Softw. Eng. & Advanced App., pp 154–158. https://doi.org/10.1109/SEAA.2018.00033

Tran H, Tran N, Nguyen S, Nguyen H, Nguyen TN (2019) Recovering variable names for minified code with usage contexts. In: Proceedings of the 41st International Conference on Software Engineering, IEEE Press, ICSE '19, pp 1165–1175. https://doi.org/10.1109/ICSE.2019.00119

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.