SeSaMe: A Data Set of Semantically Similar Java Methods

Marius Kamp, Patrick Kreutzer, Michael Philippsen
Programming Systems Group, Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Erlangen, Germany
marius.kamp@fau.de, patrick.kreutzer@fau.de, michael.philippsen@fau.de

Abstract—In the past, techniques for detecting similarly behaving code fragments were often only evaluated with small, artificial oracles or with code originating from programming competitions. Such code fragments differ largely from production codes.

To enable more realistic evaluations, this paper presents SeSaMe, a data set of method pairs that are classified according to their semantic similarity. We applied text similarity measures on JavaDoc comments mined from 11 open source repositories and manually classified a selection of 857 pairs.

I. INTRODUCTION

Functionality is often implemented more than once [1]–[4]. Even across software projects there are semantically similar methods that are equivalent, perform the same computation, or achieve the same goal, albeit in different ways. Developers may choose to merge such methods to reduce the maintenance effort, they may reuse test cases for them, or they can fix similar bugs that occur in all the similar methods.

We argue that work on detecting semantically similar code could benefit from more realistic evaluations. Two recent papers [5], [6] use a tiny, artificial data set [7], which comprises only 16 positives, some of which even differ semantically. Others [8]–[10] rely on some submissions for the Google CodeJam programming competition [11]. All submissions for an exercise are expected to be (nearly) equivalent; submissions for different exercises are semantically distinct. These many submissions are, however, still ill-suited as a benchmark for semantic similarity for three reasons: First, as the problems to solve are only algorithmic and mainly use basic data types, they only cover a fraction of the code present in real-world software. Second, the binary classification in (nearly) equivalent and semantically distinct is too strict because two similar real-world code fragments are rarely written against the same specification and hence may behave differently in subtle or systematic ways. Third, the categories of the exercises (e.g., number theory or graph algorithms) represent a feature that may suffice to determine the (dis-)similarity of the submissions without even reasoning about the semantics of the programs. Other data sets from educational contexts (like the one used by Li et al. [12]) show similar shortcomings.

As an alternative, this paper presents SeSaMe, a data set of 857 Java method pairs, automatically mined from real-world code repositories, that were then manually classified according to their semantic similarities. These methods have a variety of goals (like mathematical calculations, graphical user interface management, or data organization), heavily use project-specific data types, and are often neither equivalent nor completely distinct in functionality. The key idea of the automatic mining is to exploit the similarity of the JavaDoc comments. The data set holds similar as well as dissimilar pairs that are labeled w.r.t. three flavors of similarity.

Section II presents our methodology for data collection. Section III details the data set and how to use it. Section IV discusses some challenges and limitations. We cover related work in Section V before we conclude.

II. DATA COLLECTION

Today’s software projects in general provide an API documentation to assist developers with finding methods that implement a desired behavior. Because the documentation of individual methods often describes their semantics [13], we consider it much more likely that methods with similar JavaDoc comments are semantically similar than arbitrary pairs of methods. Hence, we first extracted JavaDoc comments from popular open-source Java repositories. Then, we trained a language model to compute the pairwise similarity of these comments. Next, we manually inspected and classified a sample of these method pairs w.r.t. their semantic similarity.

We searched for semantically similar methods in 11 Java projects from a wide range of application domains; see Table I for the 5 general-purpose utility libraries, 4 specialized libraries, and 2 applications. There is a chance to find semantically similar methods across projects because there are subsets with some overlapping functionality.

Below we outline the three steps to create SeSaMe.

A. Mining JavaDoc Comments

When mining the JavaDoc comments of these projects into a database, we skipped 5 kinds of methods: (1) Abstract methods

<table>
<thead>
<tr>
<th>Project</th>
<th>URL</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-collections</td>
<td>commons.apache.org/proper/commons-collectio ns</td>
<td>7d4d2114</td>
</tr>
<tr>
<td>commons-lang</td>
<td>commons.apache.org/proper/commons-lang</td>
<td>f10ld4a945</td>
</tr>
<tr>
<td>guava</td>
<td>github.com/google/guava</td>
<td>581b1436</td>
</tr>
<tr>
<td>openjdk11</td>
<td>openjdk.java.net/projects/jdk11</td>
<td>51151</td>
</tr>
<tr>
<td>trove</td>
<td>bitbucket.org/trove/trove</td>
<td>32317</td>
</tr>
<tr>
<td>caffeine</td>
<td>github.com/ben-manes/caffeine</td>
<td>f107b1c8</td>
</tr>
<tr>
<td>commons-math</td>
<td>commons.apache.org/proper/commons-math</td>
<td>3d4b17077</td>
</tr>
<tr>
<td>deeplearning4j</td>
<td>deeplearning4j.org</td>
<td>19f20acab0</td>
</tr>
<tr>
<td>eclipse.jdt.core</td>
<td><a href="http://www.eclipse.org/jdt/core/index.php">www.eclipse.org/jdt/core/index.php</a></td>
<td>ea5a58a1</td>
</tr>
<tr>
<td>checkstyle</td>
<td>checkstyle.sourceforge.net</td>
<td>b2f8bca79</td>
</tr>
<tr>
<td>freemind</td>
<td>freemind.sourceforge.net</td>
<td>39f6673b</td>
</tr>
</tbody>
</table>


Copyright 2019 IEEE
and non-default, non-static methods declared in interfaces as they do not hold any code. (2) Small methods with less than 10 abstract syntax tree nodes (according to Eclipse JDT) unless there is an infix operator or a conditional expression. This skips the ubiquitous trivial getters/setters, which are uninteresting for a semantic comparison. (3) Methods that only throw an exception as they usually signal unsupported functionality. (4) Methods with a signature of equals(Object), hashCode(), or toString() as well as constructors as they appear frequently, are likely to have similar JavaDoc comments, and are specific to the underlying data representation. (5) Methods that match our heuristics for test methods (e.g., methods in classes whose names start or end with “Test[s,]” and have a parent directory named “test{s,}”). Skipping test methods is a concession to techniques for semantic similarity that rely on test cases. The resulting database holds 60,160 methods and their JavaDoc comments.

B. Similarity Computation

Calculating the similarity between text documents with markup annotation (and this is what JavaDoc is) is a common task in information retrieval [14]. According to Manning et al. [14], we first normalized the comments by removing markup annotations and expanding contractions like “e.g.” or “don’t”. We then converted the comments into a vector representation that effectively captures the textual similarities.

As there does not seem to be a best transformation into a vector [15, 16], we chose several vector space models [14, 16]–[20] and word embedding algorithms [21, 22] and applied a particle swarm optimization (PSO) [23] to find a vector representation that performs well. In our training set, we labeled the method pairs in a binary way, indicating whether the JavaDoc comments of the methods are similar or not. We then defined the unified similarity 

\[ \text{msim}(m_1, m_2) = \mu \cdot \text{docsim}(m_1, m_2) + (1-\mu) \cdot (1-\text{toksim}(m_1, m_2)) \]

In this way, method pairs with high JavaDoc similarity and low token similarity have a high unified similarity msim. For the creation of our data set we used \( \mu = 0.75 \) to give more weight to JavaDoc similarity than to token dissimilarity.

C. Crafting a Diverse Data Set

As JavaDoc similarity does not imply semantic similarity, we need human intervention. Since manually examining all found pairs with a positive msim value is not an option, \(^3\) we need to pick a subset. Inspecting only those method pairs with the highest msim values may lead to a uniform data set because the mined projects may contain large clusters of similar methods. Hence, the selected pairs should (a) contain

\(^1\) We started with a tf-idf vector space model (see Manning et al. [14, section 6.2]), but added other techniques that showed different or better performance.

\(^2\) We retrieved the token streams with the tools from Eclipse JDT and omitted identifier and type names to only capture the structural commonalities.

\(^3\) If inspecting a single pair only took an unrealistic 5 seconds, a person would need 6 years full-time.
a sufficient amount of semantically similar pairs and (b) be diverse to support a fair evaluation of different techniques for determining semantic similarity. To fulfill (a) and (b), we use the Maximal Marginal Relevance (MMR) [25] approach, which finds and ranks documents that are both as similar as possible to a query and also as different from each other as possible. MMR iteratively selects documents \( d \) from a set \( R \) for inclusion in set \( S \) using a query \( Q \) as follows:

\[
d = \arg \max_{d_i \in R \setminus S} \left( \lambda \cdot \text{Sim}_1(d_i, Q) - (1 - \lambda) \cdot \max_{d_j \in S} \text{Sim}_2(d_i, d_j) \right).
\]

The parameter \( \lambda \) tunes the results between relevance and diversity. \( \text{Sim}_1 \) is the similarity of a document to the query \( Q \) and \( \text{Sim}_2 \) is the similarity of two documents. MMR selects these documents \( d \) one by one until the resulting set \( S \) reaches the desired cardinality. To adapt this framework to our task, we regard a pair of methods \( (m_1, m_2) \) as a document. \( \text{Sim}_1 \) is the similarity \( \text{msim}(m_1, m_2) \) of the given pair. For two method pairs \( d_i = (m_1^{(i)}, m_2^{(i)}) \) and \( d_j = (m_1^{(j)}, m_2^{(j)}) \), \( \text{Sim}_2 \) is:

\[
\text{Sim}_2(d_i, d_j) = \frac{2}{1 + \exp \left( - \sum_{k \in \{1, 2\}} \text{msim}(m_1^{(k)}, m_1^{(k)}) - \text{pfsim}(d_i, S) \right)} - 1.
\]

Intuitively, \( \text{Sim}_2 \) adds together all possible combinations of \( \text{msim} \) for the methods in \( d_i \) and \( d_j \). Additionally, it includes the project/file similarity \( \text{pfsim}(d_i, S) \), which is the sum of the relative frequency of the projects and the relative frequency of the files in which the given methods are declared. This avoids that MMR repeatedly samples methods from the same project/file. MMR computes \( \text{pfsim} \) based on the previously selected method pairs. Hence, it is not stored in the database. As a single high similarity value should result in a high \( \text{Sim}_2 \), a sigmoid-like function then constrains the result to \([0, 1]\).

We used MMR with \( \lambda = 0.6 \) (i.e., similarity is slightly more important than diversity) to sample 900 method pairs from the database.\(^4\) Preliminary experiments showed that \( \lambda = 0.6 \) yields a good compromise between quality and runtime. When there was more than one pair that maximized the MMR formula, we randomly picked one. MMR only selected pairs with methods from different projects. This reduces the chances of picking method pairs that have similar JavaDoc comments just because they are related by inheritance.

\[
\text{msim}(d_i, d_j) = \text{Sim}_1(d_i, Q) - (1 - \lambda) \cdot \text{Sim}_2(d_i, d_j).
\]

\[
\text{pfsim}(d_i, S) = \sum_{k \in S} \text{pfsim}(d_i, k).
\]

D. Labeling the Method Pairs

These 900 method pairs were then inspected and classified by 8 expert programmers (called voters below) who have at least a Master’s degree in Computer Science. As preliminary experiments showed that a single scale to classify semantic similarity is insufficient, the voters rated the similarity w.r.t. goals (the purpose of the methods and the context in which they are called), operations (the abstract steps that the method performs to achieve its goal), and effects (the state change on a technical level that the method causes).

The voters rated whether they agree that two presented methods are similar with respect to each of these categories. For this purpose, they could choose between 3 options (“agree”, “conditionally agree”, and “disagree”) and specify their confidence (“high”, “medium”, and “low”).

We distributed the method pairs among the voters so that each pair got three votes. The number of pairs that a voter classified ranges from 150 to 900 pairs; most voters inspected 300 pairs. Each voter received a random permutation of method pairs to mitigate order effects. To classify the pairs, the voters looked at the two methods side-by-side. To further aid comprehension, the pairs were supplemented by the accessed fields and by the called methods as long as these were defined in the same source code file (to avoid an information overload). The voters spent a total of about 70 hours inspecting a total of about 80,000 lines of source code. The numbers of lines of the pairs range from 12 to 1,271 with a median of 49 lines. Most voters classified their pairs over a period of less than 5 days, interspersed with their other job duties.

Since the classification is a complex, repetitive task, the voters may have suffered from a phenomenon called qualitative overload, which induces boredom and results in a reduced attention [26]. Hence, it is likely that they made some mistakes. Therefore, after the initial classification was done we discussed the pairs on which they disagreed. In total, the voters altered 215 of the 2,700 classifications after the discussion, but they still do not agree on all method pairs. This is expected because our definitions of the three flavors of similarity are informal and allow for interpretation. To make our data set usable for evaluations, we excluded the cases where at least two voters chose conflicting options (“agree” and “disagree”) for at least one of the flavors of similarity or where at least two voters reported “low” confidence in their answers. The resulting SeSaMe data set comprises 857 method pairs.

![Fig. 1. Summary of all votes.](image-url)

\(^4\)This is the largest number that the voters were willing to manually classify and that conforms to our methodology described in Section II-D.
Our data set SeSaMe and the tools used to create it are publicly available at https://github.com/FAU-Inf2/sesame. The relevant data are stored in a single JSON file (dataset.json) that contains an object describing the used Java projects and a list holding the classified method pairs. Each list element consists of four components: The pair id that identifies the method pair, information regarding the first and second method that this pair consists of, and the goals, operations, and effects; rating and confidence assigned by the participants of the manual classification. A method is identified by its project, the file it is defined in, and the method signature. We also supply the database that contains a table holding all 60,160 mined methods and another table for the 37,930,253 computed docsim and toksim values.

We envision several use cases for SeSaMe: For instance, our data sets supports the evaluation of techniques that compute the semantic similarity of methods (see Section I). Instead of a binary notion of semantic similarity, our data set provides different flavors of similarity. This could, for example, help to detect that a technique performs very well in detecting the (more technical) similarity of methods with similar goals and effects, but only achieves modest success when only the goals on a higher level of abstraction are similar.

Furthermore, SeSaMe may be used in studies that explore the features humans perceive as semantically similar. A better understanding of the concept of semantic similarity can, for example, assist the development of code obfuscation techniques that transform source code so that humans no longer associate it with its actual purpose.

IV. LIMITATIONS

Our approach to create the data set bears some limitations that may affect its applicability: First, we tried to avoid that the selection of projects influences the results by the way we selected the software projects (see the reasoning in Section II).

Second, we tried to avoid the Google CodeJam problem where methods have non-semantic features that can be exploited for classifying similarity. Although we dealt with syntactic similarities (see Section II-B), there may still be other biases that we are not aware of.

The third limitation is in the manual classification. None of the voters plays an active role in the selected projects. As there is strong agreement among the voters (Krippendorff’s α [27] values of 0.87, 0.80, and 0.81 for goals, operations, and effects; 1 = perfect, 0 = chance agreement), lack of familiarity did not cause random ratings. We cannot rule out a systematically wrong classification but question the ability of any similarity detection tool to gain the expert knowledge that would be needed to draw a different conclusion.

Despite these limitations, we argue that our data set is a first step towards comprehensive real-world benchmarks for semantic similarity, like those that are now commonly used to evaluate syntactic code clone detectors. We encourage other researchers to use and investigate our work to elicit a discussion on the requirements for such data sets.

V. RELATED WORK

Research on code clone detection resulted in several data sets of syntactically similar code [28]–[31]. Although syntactically similar code might also be semantically similar, syntactically dissimilar code can well be semantically similar.

Svajlenko et al. [29] claim that their data set for code clone detection also contains Type-4 clones, i.e., “code fragments that perform the same computation but are implemented by different syntactic variants” [7]. They obtained the clones by searching for syntactic characteristics of 10 functionalities. It needs much more to conclude that a technique for semantic similarity works for general code fragments.

The code clone detection data set by Krutz and Le [30] only comprises 9 Type-4 clones—too few to reliably evaluate tools that compute semantic similarities.

Wagner et al. [32] took a closer look at Google CodeJam submissions, selected 58 pairs of functionally similar code, and classified them according to their degree of difference regarding five measures. As we discussed in Section I, their data set lacks real-world code and therefore may at most be used in addition to SeSaMe in an evaluation.

Zilberstein and Yahav’s technique [33] to create a data set is quite similar to ours. They took a collection of code fragments from a popular Q&A website and applied a crowd-sourcing approach to classify these w.r.t. their semantic similarities, whereas we take code directly from open source repositories, which is very likely to be compilable. Their code fragments cannot be compiled, which limits applicability in tools for detecting semantic similarity.

For their evaluation, Tajima et al. [34] created a data set of 49 semantically similar intra-project method pairs but, to the best of our knowledge, it is not available online.

VI. CONCLUSION

This paper presents SeSaMe, a data set of 857 pairs of real-world Java methods and a manual classification of their semantic similarity w.r.t. goals, operations, and effects. It allows for a more fine-grained and realistic evaluation of techniques that compute semantic similarity.