2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

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 either 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 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 Code-Jam 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, the problems to solve are only algorithmic and mainly use basic data types. Hence, 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. Instead, they 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 structure manipulation), 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 I
Summary of the projects used to create SeSaMe

Project	**URL**	**Revision**
commons-collections	commons.apache.org/proper/commons-collections	74ad2114
commons-lang	commons.apache.org/proper/commons-lang	f110da945
guava	github.com/google/guava	581ba1436
openjdk11	openjdk.java.net/projects/jdk/11	51151
trove	bitbucket.org/trove4j/trove	3237c62
caffeine	github.com/ben-manes/caffeine	f107fbc8
commons-math	commons.apache.org/proper/commons-math	34bd17077
deeplearning4j	deeplearning4j.org	19f269cab0
eclipse.jdt.core	www.eclipse.org/jdt/core/index.php	ea5aa5d8a1
checkstyle	checkstyle.sourceforge.net	b2f8bca79
freemind	freemind.sourceforge.net	39f6673b

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00079

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 since 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.

There does not seem to be a best transformation into a vector [15], [16]. Therefore, we chose several vector space models [14], [16]–[20] and word embedding algorithms [21], [22] and applied a particle swarm optimization (PSO) [23], [24] to find a vector representation that performs well. We labeled the method pairs in our training set in a binary way, indicating whether the JavaDoc comments of the methods are similar or not. We followed the suggestion from Turney and Pantel [15] and used the cosine of the angle between the vectors to compute their similarity. It is more important that we find *some* method pairs with actually similar JavaDoc comments than to find *all* those pairs and risk to get many false positives. Hence, the PSO optimized solely for precision (i.e., the ratio of method pairs correctly detected as similar to all pairs reported to be similar). We checked that the vector representation returned by the PSO is not overfitted on our training set by sampling random method pairs from the database and comparing the results of the similarity computation with other techniques.[1] A method pair for which the result returned by the current candidate felt wrong was labeled and added to the training set, before we repeated the PSO.

The described process eventually returned a pipeline for textual similarity computation. This pipeline works on trigrams of words (i.e., the previous and the next word are "attached" to a word) instead of working on single words. To transform the JavaDoc comments into a vector, the pipeline uses Positive Pointwise Mutual Information (PPMI) [15] with the Context Distribution Smoothing enhancement presented by Levy *et al.* [16]. Given a comment-trigram frequency matrix $\mathbf{F}$ with n_d rows and n_w columns (i.e., comment i contains trigram j $\mathbf{F}_{ij}$ times), standard PPMI is defined as follows:

$$\mathrm{p}_{ij} = \frac{\mathbf{F}_{ij}}{\sum_{x=1}^{n_d}\sum_{y=1}^{n_w}\mathbf{F}_{xy}} \qquad \mathrm{p}_{i*} = \frac{\sum_{y=1}^{n_w}\mathbf{F}_{iy}}{\sum_{x=1}^{n_d}\sum_{y=1}^{n_w}\mathbf{F}_{xy}}$$

$$\mathrm{p}_{*j} = \frac{\sum_{x=1}^{n_d}\mathbf{F}_{xj}}{\sum_{x=1}^{n_d}\sum_{y=1}^{n_w}\mathbf{F}_{xy}} \quad \mathrm{ppmi}_{ij} = \max\left(0, \log\frac{\mathrm{p}_{ij}}{\mathrm{p}_{i*}\cdot\mathrm{p}_{*j}}\right).$$

Here, p_{ij} is the probability that drawing a random comment-trigram pair yields comment i and trigram j (i.e., p_{ij} is their joint probability). p_{i*} and p_{*j} describe the probabilities that the trigram i occurs in any comment and that any trigram occurs in comment j, respectively [17]. In this way, PPMI is a measure of the co-occurrence of comment i and trigram j.

Levy *et al.* [16] adapted a technique called Context Distribution Smoothing to PPMI to reduce the impact of rare words. Their modified variant replaces p_{*j} with:

$$\mathrm{p}_{*j}^{(\alpha)} = \frac{\left(\sum_{x=1}^{n_d}\mathbf{F}_{xj}\right)^{\alpha}}{\sum_{x=1}^{n_d}\left(\sum_{y=1}^{n_w}\mathbf{F}_{xy}\right)^{\alpha}}.$$

As suggested in their work, we set α to 0.75.

The described pipeline stores 37,930,253 method pairs with a JavaDoc similarity > 0 (2% of all pairs) in the database.

Preliminary experiments with these method pairs showed that some of them with the highest JavaDoc similarity are also syntactically very similar. If we only selected syntactically similar pairs, SeSaMe would add little to existing data sets (see Section V). Hence, we also computed the similarity of the tokens present in the bodies of the method pairs by means of the Levenshtein distance [25].[2] We then defined the unified similarity $\mathrm{msim}(m_1, m_2)$ to interpolate between the JavaDoc similarity $\mathrm{docsim}(m_1, m_2)$ and the token dissimilarity $1 - \mathrm{toksim}(m_1, m_2)$ using the parameter μ:

$$\begin{aligned}\mathrm{msim}(m_1, m_2) = {} & \mu \cdot \mathrm{docsim}(m_1, m_2) \\ & + (1-\mu)\cdot(1 - \mathrm{toksim}(m_1, m_2)).\end{aligned}$$

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) [26] 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:

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

The parameter λ tunes the results between relevance and diversity. Sim_1 is the similarity of a document to the query Q and 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 MMR to our task, we regard a pair of methods (m_1, m_2) as a document.[4] Sim_1 is the similarity $\mathrm{msim}(m_1, m_2)$ of the given pair. For two method pairs $d_i = \left(m_1^{(i)}, m_2^{(i)}\right)$ and $d_j = \left(m_1^{(j)}, m_2^{(j)}\right)$, Sim_2 is:

$$\mathrm{Sim}_2(d_i, d_j) = \frac{2}{1+\exp\left(-\sum_{k,l \in \{1,2\}} \mathrm{msim}\left(m_k^{(i)}, m_l^{(j)}\right) - \mathrm{pfsim}(d_i, S)\right)} - 1.$$

Intuitively, Sim_2 adds together all possible combinations of msim for the methods in d_i and d_j. Additionally, it includes the project/file similarity $\mathrm{pfsim}(d_i, S)$, 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 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 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.[5] 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.

D. *Labeling the Method Pairs*

These 900 method pairs were then inspected and classified by 8 expert programmers (called voters below) who have graduated at least with a Master's degree in Computer Science. Preliminary experiments showed that a single scale to classify semantic similarity is insufficient. Hence, 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).

[4]We omit the query Q because users do not need to control the sampling.
[5]This is the largest number that the voters were willing to manually classify and that conforms to our methodology described in Section II-D.

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 [27]. 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 did 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 with conflicting votes ("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 shows the distribution of the classifications for all voters. There are 66 method pairs that the majority rated as similar in all three flavors of similarity. In contrast, most of the voters considered 543 method pairs as dissimilar in all three flavors. Fig. 2 summarizes the pairs that were rated similar/distinct w.r.t. one flavor but distinct/similar w.r.t. the others. This suggests that similarity of operations and effects is often tied to a similarity of goals whereas similar

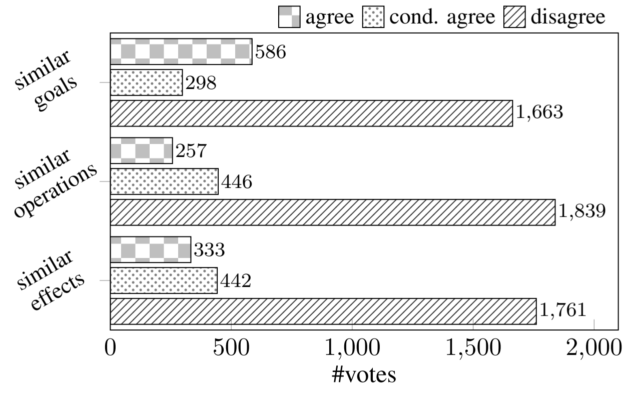

Fig. 1. Summary of all votes.

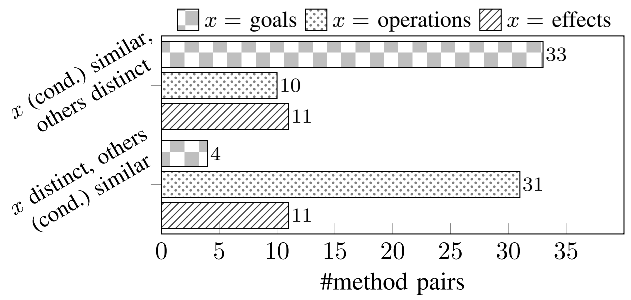

Fig. 2. (Conditional) similarity vs. dissimilarity for the three flavors.

methods with different operations are quite common. The latter case covers different approaches to the same problem and might be the most challenging to detect.

III. Using the Data Set

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 `pairid` 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 semantic similarity can, for example, assist the development of program obfuscators that transform 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 [28] 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. Yet, a similarity detection tool would need a way to gain the expert knowledge that would be needed to draw a different conclusion. We argue that this is far beyond the current state of the art.

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 [29]–[32]. Although syntactically similar code *might* also be semantically similar, syntactically *dis*similar code can well be semantically similar.

Svajlenko *et al.* [30] 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 [31] only comprises 9 Type-4 clones—too few to reliably evaluate tools that compute semantic similarities.

Wagner *et al.* [33] 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 [34] 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 compileable. Their code fragments cannot be compiled, which limits applicability in tools for detecting semantic similarity.

For their evaluation, Tajima *et al.* [35] 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.

References

[1] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *ISSTA'09: Intl. Symp. Softw. Testing and Analysis*, Chicago, IL, Jul. 2009, pp. 81–92.

[2] E. Juergens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy & paste," in *CSMR'10: European Conf. Softw. Maintenance and Reengineering*, Madrid, Spain, Mar. 2010, pp. 78–87.

[3] V. Käfer, S. Wagner, and R. Koschke, "Are there functionally similar code clones in practice?" In *IWSC'18: Intl. Workshop Softw. Clones*, Campobasso, Italy, Mar. 2018, pp. 2–8.

[4] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *MSR'16: Mining Softw. Repositories*, Austin, TX, May 2016, pp. 362–373.

[5] D. E. Krutz and E. Shihab, "CCCD: Concolic code clone detection," in *WCRE'13: Working Conf. Reverse Eng.*, Koblenz, Germany, Oct. 2013, pp. 489–490.

[6] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: Memory comparison-based clone detector," in *ICSE'11: Intl. Conf. Softw. Eng.*, Waikiki, Honolulu, HI, May 2011, pp. 301–310.

[7] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, May 2009.

[8] W. Li, H. Saidi, H. Sanchez, M. Schäf, and P. Schweitzer, "Detecting similar programs via the Weisfeiler-Leman graph kernel," in *ICSR'16: Intl. Conf. Softw. Reuse*, vol. 9679, Limassol, Cyprus, Jan. 2016, pp. 315–330.

[9] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," in *ICPC'16: Intl. Conf. Program Comprehension*, Austin, TX, May 2016, pp. 1–10.

[10] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: Detecting similarly behaving software," in *FSE'16: Foundations of Softw. Eng.*, Seattle, WA, Nov. 2016, pp. 702–714.

[11] (Jan. 2019). Google CodeJam, [Online]. Available: https://code.google.com/codejam/.

[12] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *ICSE'16: Intl. Conf. Softw. Eng.*, Austin, TX, May 2016, pp. 501–510.

[13] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *ASE'01: Automated Softw. Eng.*, San Diego, CA, Nov. 2001, pp. 107–114.

[14] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, England: Cambridge University Press, 2008.

[15] P. D. Turney and P. Pantel, "From frequency to meaning: Vector space models of semantics," *J. of Artificial Intelligence Research*, vol. 37, pp. 141–188, Feb. 2010.

[16] O. Levy, Y. Goldberg, and I. Dagan, "Improving distributional similarity with lessons learned from word embeddings," *Trans. of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015.

[17] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Computational Linguistics*, vol. 16, no. 1, pp. 22–29, 1990.

[18] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, Sep. 1990.

[19] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, Oct. 1999.

[20] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS'13: Neural Information Processing Sys.*, Lake Tahoe, NV, Dec. 2013, pp. 3111–3119.

[22] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML'14: Intl. Conf. Machine Learning*, Beijing, China, Jun. 2014, pp. 1188–1196.

[23] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *ICNN'95: Intl. Conf. Neural Networks*, Perth, Australia, Nov. 1995, pp. 1942–1948.

[24] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *SIS'07: Swarm Intelligence Symp.*, Honolulu, HI, Apr. 2007, pp. 120–127.

[25] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[26] J. Carbonell and J. Goldstein, "The use of MMR, diversity-based reranking for reordering documents and producing summaries," in *SIGIR'98: Research and Development in Information Retrieval*, Melbourne, Australia, Aug. 1998, pp. 335–336.

[27] C. D. Fisher, "Boredom at work: A neglected concept," *Human Relations*, vol. 46, no. 3, pp. 395–417, 1993.

[28] R. Artstein and M. Poesio, "Inter-coder agreement for computational linguistics," *Computational Linguistics*, vol. 34, no. 4, pp. 555–596, Dec. 2008.

[29] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. on Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[30] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. Mamun Mia, "Towards a big data curated benchmark of inter-project code clones," in *ICSME'14: Intl. Conf. Softw. Maintenance and Evolution*, Victoria, BC, Canada, Sep.–Oct. 2014, pp. 476–480.

[31] D. E. Krutz and W. Le, "A code clone oracle," in *MSR'14: Mining Softw. Repositories*, Hyderabad, India, May–Jun. 2014, pp. 388–391.

[32] Y. Yuki, Y. Higo, K. Hotta, and S. Kusumoto, "Generating clone references with less human subjectivity," in *ICPC'16: Intl. Conf. Program Comprehension*, Austin, TX, May 2016, pp. 1–4.

[33] S. Wagner, A. Abdulkhaleq, I. Bogicevic, J.-P. Ostberg, and J. Ramadani, "How are functionally similar code clones syntactically different? an empirical study and a benchmark," *PeerJ Computer Science*, vol. 2, no. e49, Mar. 2016.

[34] M. Zilberstein and E. Yahav, "Leveraging a corpus of natural language descriptions for program similarity," in *Onward'16: Intl. Symp. New Ideas, New Paradigms, and Reflections on Programming and Softw.*, Amsterdam, The Netherlands, Nov. 2016, pp. 197–211.

[35] R. Tajima, M. Nagura, and S. Takada, "Detecting functionally similar code within the same project," in *IWSC'18: Intl. Workshop Softw. Clones*, Campobasso, Italy, Mar. 2018, pp. 51–57.