IdBench: Evaluating Semantic Representations of Identifier Names in Source Code

Yaza Wainakh
Department of Computer Science
TU Darmstadt
Darmstadt, Germany
yaza.wainakh@gmail.com

Moiz Rauf
Department of Computer Science
University of Stuttgart
Stuttgart, Germany
moiz.rauf@iste.uni-stuttgart.de

Michael Pradel
Department of Computer Science
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

***Abstract*—Identifier names convey useful information about the intended semantics of code. Name-based program analyses use this information, e.g., to detect bugs, to predict types, and to improve the readability of code. At the core of name-based analyses are semantic representations of identifiers, e.g., in the form of learned embeddings. The high-level goal of such a representation is to encode whether two identifiers, e.g., `len` and `size`, are semantically similar. Unfortunately, it is currently unclear to what extent semantic representations match the semantic relatedness and similarity perceived by developers. This paper presents IdBench, the first benchmark for evaluating semantic representations against a ground truth created from thousands of ratings by 500 software developers. We use IdBench to study state-of-the-art embedding techniques proposed for natural language, an embedding technique specifically designed for source code, and lexical string distance functions. Our results show that the effectiveness of semantic representations varies significantly and that the best available embeddings successfully represent semantic relatedness. On the downside, no existing technique provides a satisfactory representation of semantic similarities, among other reasons because identifiers with opposing meanings are incorrectly considered to be similar, which may lead to fatal mistakes, e.g., in a refactoring tool. Studying the strengths and weaknesses of the different techniques shows that they complement each other. As a first step toward exploiting this complementarity, we present an ensemble model that combines existing techniques and that clearly outperforms the best available semantic representation.**

***Index Terms*—source code, neural networks, embeddings, identifiers, benchmark**

I. Introduction

Identifier names play an important role in writing, understanding, and maintaining high-quality source code [1]. Because they convey information about the meaning of variables, functions, classes, and other program elements, developers often rely on identifiers to understand code written by themselves and others. Beyond developers, various automated techniques analyze, use, and improve identifier names. For example, identifiers have been used to find programming errors [2]–[5], to mine specifications [6], to infer types [7], [8], to predict the name of a method [9], or to complete partial code using a learned language model [10]. Techniques for improving identifier names pinpoint inappropriate names [11] and suggest more suitable names [12]. The basic idea of all these approaches is to infer the intended meaning of a piece of code from the natural language information in identifiers, possibly along with other information, such as the structure of code, data flow, and control flow. We here refer to program analyses that rely on identifier names as a primary source of information as *name-based analyses*.

Most name-based analyses reason about names in one of two ways. First, some approaches build upon *string distance functions*, such as the Levenshtein distance, sometimes in combination with algorithms for tokenizing names, e.g., based on underscore or camel-case notation [13]. Given a pair of identifiers, e.g. `len` and `length`, a string distance function yields a real-valued number that indicates to what extent the character sequences in the identifiers resemble each other. String distance functions are at the core of name-based analyses to detect name-related bugs [2], [3], to predict types [7], to improve identifier names [14], or to suggest appropriate names [15]. Second, another approach, which has become popular more recently, are neural network-learned *embeddings of identifiers*. An embedding maps each identifier into a continuous vector representation, so that similar identifiers are mapped to similar vectors. Embeddings implicitly define a similarity function via the cosine similarity of embedding vectors. For example, embeddings of identifiers are at the core of neural program analyses [16] to predict types [8], to detect bugs [4], to de-obfuscate code [17], to complete partial code [10], and to map API elements across programming languages [18].

The common aim of both string distance functions and embeddings of identifiers is to reason about the semantics of identifiers, and we hence call both of them *semantic representations of identifiers*, or short *semantic representations*. The overall effectiveness of a name-based analysis relies on the assumption that the underlying semantic representation encodes some kind of semantic relationship between identifiers. For example, two semantically similar identifiers, such as `len` and `length`, should be closer to each other than two unrelated identifiers, such as `length` and `click`.

Despite the importance of semantic representations for name-based analyses, it is currently unclear how well existing

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

approaches actually represent semantic relationships. Specifically, we are interested in the following questions:

a) RQ 1: How accurately do state-of-the-art semantic representations match the semantic relatedness *of identifiers as perceived by software developers?:* "Relatedness" here means the degree of association between two identifiers, which covers various possible relations between them, e.g., being used in the same application domain or being opposites of each other. For example, `top` and `bottom` are related because they are opposites, `click` and `dblclick` are related because they belong to the same general concept, and `getBorderWidth` and `getPadding` are related because they belong to the same application domain. The relatedness of identifiers is relevant for tools that reason about the broad meaning of code elements, e.g., to predict the types of functions [8], [19].

b) RQ 2: How accurately do state-of-the-art semantic representations match the semantic similarity *of identifiers as perceived by software developers?:* "Similarity" here means the degree to which two identifiers have the same meaning, in the sense that one could substitute the other without changing the overall meaning [20]. For example, `length` and `size`, as well as `username` and `userid`, are similar to each other. The similarity of identifiers is, e.g., relevant for name-based bug detection tools [4], [21].

c) RQ 3: What are the strengths and weaknesses of the existing semantic representations?: Better understanding why particular techniques sometimes succeed or fail to accurately represent identifiers will enable improving the current semantic representations.

d) RQ 4: Do the existing semantic representations complement each other?: If current techniques are complementary, it may be possible to combine them in a way that outperforms the individual techniques.

Addressing these questions relies on a way to measure and compare the effectiveness of semantic representations of identifiers in source code. This paper presents IdBench, the first benchmark for this task, which is based on a dataset of developer assessments about the relatedness and similarity of pairs of identifiers. We gather this dataset through surveys that show real-world identifiers and code snippets to hundreds of developers, asking them to rate their semantic relationship. Taking the developer assessments as a gold standard, IdBench allows for evaluating semantic representations in a systematic way by measuring to what extent a semantic representation agrees with ratings given by developers. Moreover, inspecting pairs of identifiers for which a representation strongly agrees or disagrees with the benchmark helps understand the strengths and weaknesses of the representation.

Applying our methodology to seven widely used semantic representations leads to various novel insights. We find that different techniques differ heavily in their ability to accurately represent identifier relatedness and similarity. The best among the studied techniques, the CBOW variant of FastText [22], accurately represents the relatedness of identifiers (RQ 1), but none of the available techniques accurately represents the similarity of identifiers (RQ 2). Studying the strengths and weaknesses of each technique (RQ 3) shows that some embeddings are confused about identifiers with opposite meaning, e.g., `rows` and `cols`, about identifiers that belong to the same application domain but are not similar, and about synonyms, e.g., `file` and `record`. Furthermore, practically all techniques struggle with identifiers that use abbreviations, which are very common in software. We also find that simple string distance functions, which measure the similarity of identifiers without any learning, are surprisingly effective, and even outperform some learned embeddings for the similarity task.

A close inspection of the results shows that different techniques complement each other (RQ 4). To benefit from the strengths of multiple techniques, we present a new semantic representation that combines the available techniques into an ensemble model based on features of identifiers, such as the number of characters or whether an identifier contains non-dictionary words. The ensemble model clearly outperforms each of the existing semantic representations, improving agreement with developers by 6% and 19% for relatedness and similarity, respectively.

In summary, this paper makes the following contributions.

- *A reusable benchmark*. We make available a benchmark of hundreds of pairs of identifiers, providing a way to systematically evaluate existing and future embeddings.[1] To the best of our knowledge, this is the first benchmark to systematically evaluate semantic representations of identifiers.
- *Novel insights*. Our study reveals both strengths and limitations of current semantic representations, along with concrete examples to illustrate them. These insights provide a basis for future work on better semantic representations.
- *A technique that outperforms the state-of-the-art*. Combining the currently available techniques based on a few simple features yields a semantic representation that clearly outperforms all individual techniques.

II. Methodology

To measure and compare the accuracy of semantic representations, we gather thousands of ratings from 500 developers (Section II-A). Cleaning and compiling this raw dataset into a benchmark yields several hundreds of pairs of identifiers with gold standard similarities (Section II-B). We then measure the agreement between the gold standard and state-of-the-art semantic representations (Section II-C), where we study two string distance functions and five learned embeddings (Section II-D). We apply our methodology to JavaScript code, because recent work on identifier names and code embeddings focuses on this language [4], [8], [17], [23], but our methodology can also be applied to other languages.

A. Developer Surveys

IdBench includes three benchmark tasks: A *relatedness task* and two tasks to measure how well an embedding

[1] https://github.com/sola-st/IdBench

Identifiers: `radians`, `angle`

1) *How related are the identifiers?*

Unrelated ◯ ◯ ◯ ◯ ◯ Related

2) *Could one substitute the other?*

Not substitutable ◯ ◯ ◯ ◯ ◯ Substitutable

(a) Direct survey.

Which identifier fits best into the blanks?

◯ `positions` ◯ `indices`

```
Opentip.____ = [ "top", "topRight",
    "right", "bottomRight", "bottom",
    "bottomLeft", "left", "topLeft"];
Opentip.position = {};
_ref = Opentip.____;
```

(b) Indirect survey.

Fig. 1: Examples of the developer surveys.

reflects the similarity of identifiers: a *similarity task* and a *contextual similarity task*. The following describes how we gather developer assessments that provide data for these tasks. The supplementary material provides additional examples and details of the survey setup.

a) Direct Survey of Developer Assessments: This survey shows two identifiers to a developer and then directly asks how related and how similar the identifiers are. Figure 1a shows an example question from the survey. The developer is shown pairs of identifiers and is then asked to rate on a five-point Likert scale how related and how similar these identifiers are to each other. In total, each developer is shown 18 pairs of identifiers, which we randomly sample from a larger pool of pairs. Before showing the questions, we provide a brief description of what the developers are supposed to do, including an explanation of the terms "related" and "substitutable". The ratings gathered in the direct survey are the basis for the relatedness task and the similarity task of IdBench.

b) Indirect Survey of Developer Assessments: This survey asks developers to pick an identifier that best fits a given code context, which indirectly asks about the similarity of identifiers. The motivation is that identifier names alone may not provide enough information to fully judge how similar they are [20]. For example, without any context, identifiers `idx` and `hl` may cause confusion for developers who are trying to judge their similarity. The survey addresses this challenge by showing the code context in which an identifier occurs, and by asking the developers to decide which of two given identifiers best fits this context. If, for a specific pair of identifiers, the developers choose both identifiers equally often, then the identifiers are likely to be similar to each other, since one can substitute the other. Figure 1b shows a question asked during the indirect survey. As shown in the example, for code contexts where the identifier occurs multiple times, we show multiple blanks that all refer to the same identifier. In total, we show 15 such questions to each participant of the survey, where the 15 identifier pairs are randomly selected from the set of all studied pairs. The ratings gathered in the indirect survey are the basis for the contextual similarity tasks of IdBench.

TABLE I: Occurrences of IdBench identifiers in code corpora of different languages.

	Total occurrences		Occurrences of individual identifiers		
Language	Number	Perc.	Min	Mean	Max
JavaScript	3,697,498	12.5%	62	7,639	629,413
Python	2,279,866	14.8%	0	4,710	1,367,832
Java	757,064	6.3%	0	1,564	119424

c) Selection of Identifiers and Code Examples: We select identifiers and code contexts from a corpus of 50,000 JavaScript files [24]. We select 300 pairs, made out of 488 identifiers, through a combination of automated and manual selection, aimed at a diverse set that covers different degrees of similarity and relatedness. At first, we extract from the code corpus all identifier names that appear more than 50 times, including method names, variable names, property names, and other types of identifiers. A naive approach would be to randomly sample pairs among those identifiers. However, this naive approach would result almost only in unrelated and dissimilar identifier pairs. Instead, we follow a methodology proposed for natural language [25], which ranks all pairs based on the cosine similarity according to a given embedding, and then selects pairs from different ranges in the ranking. We select pairs using two embeddings [17], [26]. The fact that these embeddings are later also evaluated with the benchmark does not introduce bias because the ground truth of the benchmark is constructed only from the human ratings, not from the embeddings. In addition to pairs selected as suggested in [25], we manually select some synonym pairs, which we observed to lack otherwise, and add randomly selected pairs, which are likely to be unrelated. The manual selection was done before evaluating any semantic representations to avoid biasing the benchmark.

To gather the code contexts for the indirect survey, we search the code corpus for occurrences of the selected identifiers. As the size of the context, we choose five lines, aiming to provide sufficient context to pick the best identifier without overwhelming the study participants with large amounts of code. For each identifier, we randomly select five different contexts. When showing a specific pair of identifiers to a developer, we randomly select one of the gathered contexts for one of the two identifiers.

Table I shows how often the selected identifiers occur in the JavaScript corpus. Overall, the identifiers in IdBench occur 3.7 million times, which covers 12.5% of all identifier occurrences. Even though this was not a criterion when selecting the identifiers, the benchmarks covers a non-negligible portion

of real-world code. The table also shows how often individual identifiers occur, which is 7,639 times, on average. To assess whether IdBench could also be used for other languages, we also measure the occurrences in Python [27] and Java code corpora [28] with 50,000 files each. As shown in Table I, the identifiers are also frequent in code beyond JavaScript, with an average number of occurrences of 4,710 and 1,564 in the Python and Java corpora, respectively. A manual analysis shows that identifiers that occur across languages cover general programming terminology, whereas identifiers that appears in JavaScript only are mostly specific to the web domain, e.g., `tag_h4` or `DomRange`.

To better understand whether IdBench covers identifiers that appear in different syntactic roles, we measure for each identifier how often it used as a function name, variable name, or property name. We then assign each identifier to one of these roles based on whether the majority of its occurrences is in a specific role. The measurements show that 17% of the identifiers are primarily function names, 18% are primarily variables names, 34% are primarily property names, and the rest is commonly used in multiple roles.

d) Participants: We recruit developers to participate in the survey in several ways. About half of the participants are volunteers recruited via personal contacts, posts in public developer forums, and a post in an internal forum within a major software company. The other half of the participants were recruited via Amazon Mechanical Turk, where we offered a compensation of one US dollar for completing both surveys. On average, participants took around 15 minutes to complete both surveys. That is, the offered compensation matches the average salary of software developers in some countries of the world.[2] In total, 500 developers participate in the survey. Most participants live in North America and in India, and they have at least five years of experience in software development.

B. Data Cleaning

Crowd-sourced surveys may contain noise, e.g., due to lack of expertise or involvement by the participants [29]. To address this challenge, we gather at least ten ratings per pair of identifiers and then clean the data based on the inter-rater agreement, which has been found effective in other crowd-sourced surveys [30].

a) Removing Outlier Participants: As a first filter, we remove outlier participants based on the inter-rater agreement, which measures the degree of agreement between participants. We use Krippendorf's alpha coefficient, because it handles unequal sample sizes, which fits our data, as not all participants rate the same pairs and not all pairs have the same number of ratings. The coefficient ranges between zero and one, where zero represents complete disagreement and one represents perfect agreement. For each participant, we calculate the difference between her rating and the average of all the other ratings for each pair. Then, we average these differences for each rater, and discard participants with a difference above a threshold τ (values given in Table II). We perform this computation both for the relatedness and similarity ratings from the direct survey, and then remove outliers based on the average difference across both ratings.

b) Removing Downer Participants: As a second filter, we eliminate participants that decrease the overall inter-rater agreement (IRA). We call such participants *downers* [31], because they bring the agreement level between all participants down. For each participant p, we compute IRA_{sim} and IRA_{rel} before and after removing p from the data. If IRA_{sim} or IRA_{rel} increases by at least 10%, then we discard that participant's ratings.

c) Removing Outlier Pairs: As a third filter, we eliminate some pairs of identifiers used in the indirect survey. Since our random selection of code contexts may include contexts that are not helpful in deciding about the most suitable identifier, the ratings for some pairs may be misleading. For example, this is the case for code contexts that contain short and meaningless identifiers or that mostly consist of comments unrelated to the missing identifier. To mitigate this problem, we remove a pair if the difference in similarity as rated in the direct and indirect surveys exceeds some threshold θ (values given in Table II).

Table II shows the number of identifier pairs that remain in the benchmark after data cleaning. For each of the three tasks, we provide a *small*, *medium*, and *large* benchmark, which differ in the thresholds used during data cleaning. The smaller benchmarks use stricter thresholds and hence provide higher agreements between the participants, whereas the larger benchmarks offer more pairs. The thresholds are selected to strike a balance between increasing the overall inter-rater

[2]https://www.payscale.com/research/IN/Job=Software_Developer/Salary

TABLE II: Benchmark sizes and inter-rater agreement (IRA).

Size	Thresholds		Task					
	τ	θ	Relatedness		Similarity		Contextual simil.	
			Pairs	IRA	Pairs	IRA	Pairs	
Small	0.215	0.4	167	0.67	167	0.62	115	
Medium	0.23	0.5	247	0.61	247	0.57	145	
Large	0.25	0.6	291	0.56	291	0.51	176	

TABLE III: Pairs of identifiers with their gold standard similarities.

Identifier 1	Identifier 2	Score		
		Related-ness	Similar-ity	Contextual similarity
`substr`	`substring`	0.94	1.00	0.89
`setMinutes`	`setSeconds`	0.91	0.22	0.06
`reset`	`clear`	0.90	0.89	0.94
`rows`	`columns`	0.88	0.08	0.22
`setInterval`	`clearInterval`	0.86	0.09	0.34
`count`	`total`	0.83	0.81	0.79
`item`	`entry`	0.78	0.77	0.92
`miny`	`ypos`	0.68	0.37	0.02
`events`	`rchecked`	0.16	0.14	0.18
`re`	`destruct`	0.06	0.02	0.02

agreement while keeping enough pairs and ratings to form a representative benchmark.

C. Measuring Agreement with the Benchmark

Given the ground truth similarities and a semantic representation technique, we want to measure to what extent both agree with each other.

a) Converting Ratings to Scores: As a first step of measuring the agreement with the benchmark, we convert the ratings gathered for a specific pair during the developer surveys into a similarity score in the $[0, 1]$ range. For the direct survey, we scale the 5-point Likert-scale ratings into the $[0, 1]$ range and average all ratings for a specific pair of identifiers. For the indirect survey, we use a signal detection theory-based approach for converting the collected ratings into numeric values, which has been previously used to create a similarity benchmark for natural languages [20]. This conversion yields an unbounded distance measure d for each pair, which we convert into a similarity score s by normalizing and inverting the distance: $s = 1 - \frac{d - min_d}{max_d - min_d}$ where min_d and max_d are the minimum and maximum distances across all pairs.

b) Examples: Table III shows representative examples of identifier pairs and their scores for the three benchmark tasks.[3] The examples illustrate that the scores match human intuition and that the gold standard clearly distinguishes relatedness from similarity. Some of the highly related and highly similar pairs, e.g., `substr` and `substring`, are lexically similar, while others are synonyms, e.g., `count` and `total`. The identifiers `rows` and `columns` are strongly related, but one cannot substitute the other, and they hence have low similarity. Similarly `miny`, `ypos` represent distinct properties of the variable `y`, which is why they are related but not similar. Finally, some pairs are either weakly or not at all related, e.g., `re` and `destruct`.

c) Correlation with benchmark: We measure the magnitude of agreement of a semantic representation with IdBench by computing Spearman's rank correlation between the similarities of pairs of identifier vectors according to the semantic representation and our gold standard of similarity scores.

Definition 1 (Correlation with benchmark): Given n pairs (s_i, g_i) of similarity scores, where s_i is computed by a semantic representation and g_i is the gold standard, let $rank(s_i)$ and $rank(g_i)$ be the ranks of s_i and g_i, respectively. The correlation of the semantic representation with the benchmark is $r = \frac{cov(rank(s_i), rank(g_i))}{\sigma_{rank(s_i)} \sigma_{rank(g_i)}}$ where cov and σ are covariance and standard deviation of the rank variables, respectively.

The correlation ranges between 1 (perfect agreement) and -1 (complete disagreement). For string distance functions, we compute the similarity score $s_i = 1 - d_{norm}$ for each pair based on a normalized version d_{norm} of the distance returned by the string distance function. We use Spearman's rank correlation because directly comparing absolute similarities across different embeddings may be misleading [32]. The reason is that, depending on how "wide" or "narrow" an embedding space is, a cosine similarity of 0.3 may mean a rather high or a rather low similarity. A rank-based comparison, as provided by Spearman's rank correlation, is more robust to different ways of populating the embedding space with identifiers than computing the correlation of absolute similarities.

[3]The full list of identifiers pairs is available for download as part of our benchmark.

D. Embeddings and String Distance Functions

To assess how accurately existing semantic representations encode the relatedness and similarity of identifiers, we evaluate seven semantic representations against IdBench: Two string distance functions and five learned embeddings.

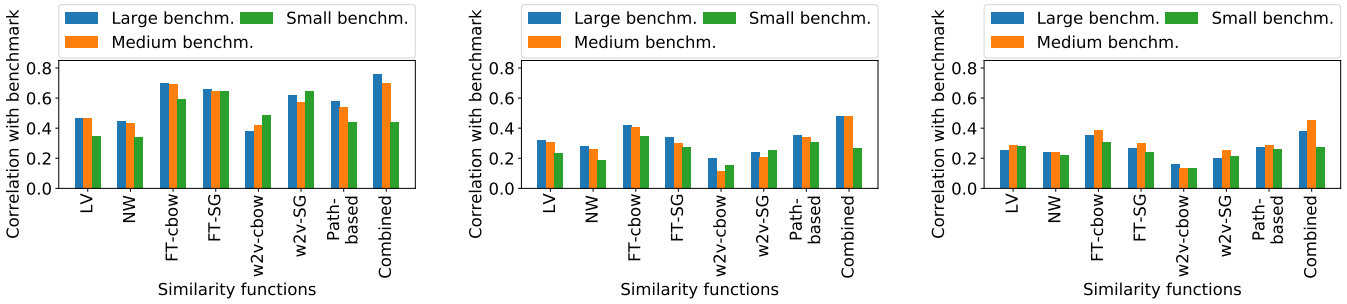
String distance functions use lexical similarity as a proxy for the semantic relatedness of identifiers. We consider these functions because they are used in name-based bug detection tools [2], including a bug detection tool deployed at Google [3], to improve identifier names [14], and to suggest appropriate names [15]. The two string distance functions we evaluate are:

- "LV": *Levenshtein's* edit distance, which is the number of character insertions, deletions, and substitutions required to transform one identifier into another.
- "NW": *Needleman-Wunsch* distance [33], which generalizes the Levenshtein distance by computing global alignments of two strings.

Learned embeddings are popular in recent name-based analyses, e.g., for bug detection [4], type prediction [8], and for predicting names and types of program elements [17]. The five learned embeddings we evaluate are:

- "w2v-cbow": The continuous bag of words variant of *Word2vec* [26], [34].
- "w2v-sg": The skip-gram variant of *Word2vec*.
- "FT-cbow": The continuous bag of words variant of *FastText* [22], a sub-word extension of Word2vec that represents words as character n-grams.
- "FT-sg": The skip-gram variant of *FastText*.
- "path-based": An embedding technique specifically designed for code, which learns from paths through a structural, tree-based representation of code [17].

We train all embeddings on the same code corpus of 50,000 JavaScript files [24]. For each embedding, we experiment with various hyper-parameters (e.g., dimension, number of context words) and report results only for the best performing models.[4] We provide all identifiers as they are to the semantic representations, without pre-processing or tokenizing identifiers. The rationale is that such pre-processing should be part of the semantic representation. For example, the NW string distance function aligns the characters of identifiers, and the FastText embeddings split identifiers into character n-grams, which may enable these techniques to reason about subtokens of an identifier.



(a) Relatedness. (b) Similarity. (c) Contextual similarity.

Fig. 2: Correlations of embeddings and string distance functions with the small, medium, and large variants of the benchmark.

III. Results

A. RQ 1: Accuracy of Representing Semantic Relatedness

The following addresses the question how accurately the studied techniques represent the relatedness of identifiers, i.e., the degree of association between the two identifiers. Figure 2a shows the agreement of the evaluated semantic representations with the small, medium, and large variants of the relatedness benchmark in IdBench. All techniques achieve relatively high levels of agreement, with correlations between 41% and 74%. The neurally learned embeddings clearly outperform the string distance-based similarity functions (41-74% vs. 46-49%), showing that the effort of learning a semantic representation is worthwhile. In particular, the learned embeddings match or even slightly exceed the inter-rater agreement, which is considered an upper bound of how strongly an embedding may correlate with a similarity-based benchmark [31]. Comparing different embedding techniques with each other, we find that both FastText variants achieve higher scores than all other embeddings. In contrast, despite using additional structural information of source code, path-based embeddings score only comparably to Word2vec.

A likely reason for the effectiveness of FastText is that it generalizes across lexically similar names by computing embeddings based on character n-grams of an identifier. E.g., given the identifier `getIndex`, FastText computes its embedding based on embeddings for its various characters n-grams, such as `Index` and `Ind`, allowing the approach to generalize across lexically similar identifiers, such as `setIndex` or `ind`.

B. RQ 2: Accuracy of Representing Semantic Similarity

This research question is about the semantic similarity, i.e., the degree to which two identifiers have the same meaning. Figure 2b shows how much the studied semantic representations agree with the similarity benchmark in IdBench. Overall, the figure shows a much lower agreement with the gold standard than for relatedness. One explanation is that encoding semantic similarity is a harder task than encoding the less strict concept of relatedness. Similar to relatedness, FT-cbow shows the strongest agreement, ranging between 35% and 38%.

The results of the contextual similarity task (Figure 2c) confirm the findings from the similarity task. All studied techniques are less effective than for relatedness, and FT-cbow achieves the highest agreement with IdBench.

A perhaps surprising result is that string distance functions are roughly as effective as some of the learned embeddings and sometimes even outperform them. The reason is that some semantically similar identifiers are also lexically similar, e.g., `len` and `length`. One downside of string distance functions is that they miss synonymous identifiers, e.g., `count` and `total`.

C. RQ 3: Strengths and Weaknesses of Existing Techniques

To better understand the strengths and weaknesses of the studied semantic representations, we inspect various examples (Section III-C1) and study interesting subsets of all identifier pairs in isolation (Section III-C2).

1) Examples: To better understand why current embeddings sometimes fail to accurately represent similarities, Table IV shows the most similar identifiers of selected identifiers according to the FT-cbow and path-based embeddings. The examples illustrate two observations. First, FastText, due to its use of n-grams [22], tends to cluster identifiers based on lexical commonalities. While many lexically similar identifiers are also semantically similar, e.g., `substr` and `substring`, this approach misses other synonyms, e.g., `item` and `entry`. Another downside is that lexical similarity may also establish wrong relationships. For example, `substring` and `substrCount` represent different concepts, but FastText finds them to be highly similar.

Second, in contrast to FastText, path-based embeddings tend to cluster words based on the structural and syntactical contexts they occur in. This approach helps the embeddings to identify synonyms despite their lexical differences, e.g., `count` and `total`, or `files` and `records`. The downside is that it also clusters various related but not similar identifiers, e.g., `minText` and `maxText`, or `substr` and `getPadding`. Some of these identifiers even have opposing

[4]Details on the hyperparameters and how we tuned them are available in the supplementary material.

TABLE IV: Top-5 most similar identifiers by the FT-cbow and path-based models.

Identifier	Embedding	Nearest neighbors				
substr	FT-cbow	substring	substrs	subst	substring1	substrCount
	Path-based	substring	getInstanceProp	getPadding	getMinutes	floor
item	FT-cbow	itemNr	itemJ	itemL	itemI	itemAt
	Path-based	entry	child	record	targ	nextElement
count	FT-cbow	countTbl	countInt	countRTO	countsAsNum	countOne
	Path-based	total	limit	minVal	exponent	rate
rows	FT-cbow	rowOrRows	rowXs	rows_l	rowsAr	rowIDs
	Path-based	cols	cells	columns	tiles	items
setInterval	FT-cbow	resetInterval	setTimeoutInterval	clearInterval	getInterval	retInterval
	Path-based	clearInterval	assume	alert	nextTick	ReactTextComponent
minText	FT-cbow	maxText	minLengthText	microsecText	maxLengthText	minuteText
	Path-based	maxText	displayMsg	blankText	disableText	emptyText
files	FT-cbow	filesObjs	filesGen	fileSets	extFiles	libFiles
	Path-based	records	tasks	names	tiles	todos
miny	FT-cbow	min_y	minBy	minx	minPt	min_z
	Path-based	minx	ymin	dataMax	dataMin	ymax

meanings, e.g., `rows` and `cols`, which can mislead code analysis tools when reasoning about the semantics of code.

2) Interesting Subsets of All Identifier Pairs: To better understand the strengths and weaknesses of semantic representations for specific kinds of identifiers, we analyze some interesting subsets of all identifier pairs in more detail. We focus on four subsets:

- *Abbreviations*. Pairs where at least one identifier is an abbreviation and where both identifiers refer to the same concept, e.g., `substr` and `substring`, or `cfg` and `conf`. Since abbreviations are commonly used for concise source code, accurately reasoning about them is important.
- *Opposites*. Pairs where one identifier is the opposite of the other identifier, e.g., `xMin` and `xMax`, or `setInstanceProp` and `getInstanceProp`. Since opposite identifiers often occur in similar contexts, they may be difficult to distinguish.
- *Synonyms*. Pairs that refer to the same concepts, e.g., `reset` and `clear`, or `emptyText` and `blankText`. These identifiers often are lexically different but should be represented in a similar way.
- *Added subtoken*. Pairs where both identifiers are identical, except that one adds a subtoken to the other, e.g., `id` and `sessionid`, or `maxLine` and `maxLineLength`.
- *Tricky tokenization*. Pairs where at least one of the identifiers is composed of multiple subtokens but uses neither camel case nor snail case to combine subtokens, e.g., `touchmove` and `touchend`, or `newtext` and `content`. This and the above subset are interesting because some semantic representations reason about subtokens of identifiers.

To extract pairs into these subsets, we inspect all 167 pairs from the small benchmark, which yields between 7 and 22 pairs per set.

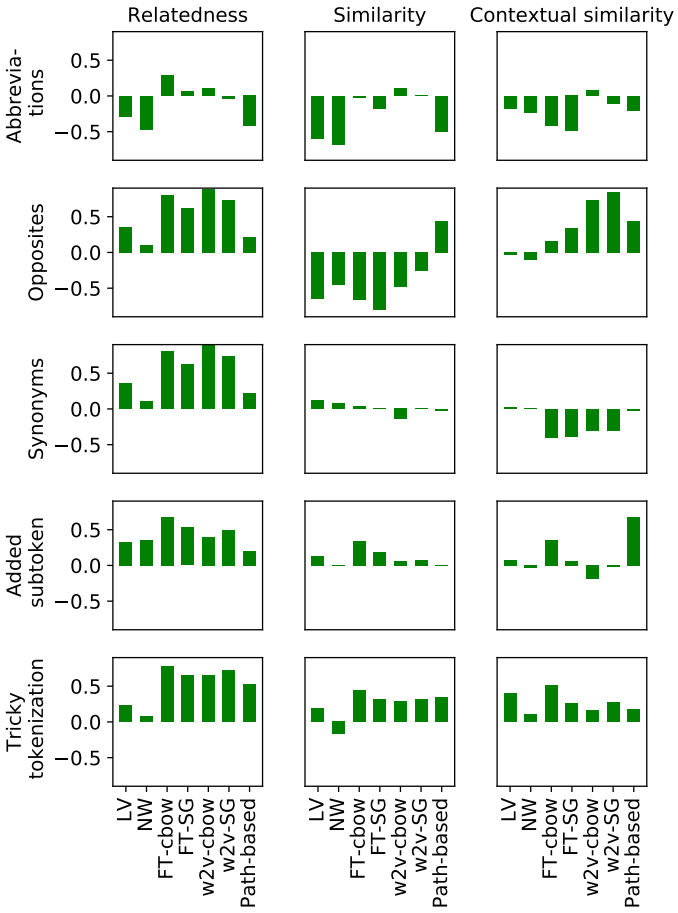


Fig. 3: Agreement and disagreement with the benchmark for different kinds of identifiers.

Figure 3 shows how much the different techniques agree or disagree with the benchmark for selected subsets. As in Figure 2, each bar shows the Spearman rank correlation between the predicted similarities and the ground truth. That is, higher values are better and negative values indicate a clear disagreement with the ground truth.

The results shows that all techniques are challenged by abbreviations, with more than half of the correlations being negative. The poor performance for abbreviations can be attributed to the fact that fewer characters provide less information and that there may be many variants of the same name. For opposites and synonyms, we find that most techniques, and in particular the learned embeddings, successfully represent the relatedness of these identifiers. However, almost all techniques clearly fail to capture that opposite identifiers are not similar, as one cannot replace the other, and to capture that synonyms are similar.

For the subtoken-related subsets, we find that most techniques are challenged by pairs where one identifier adds a subtoken to the other, in particular, when reasoning about similarity. One explanation is that identifiers with an added subtoken tend to be rather specialized, and hence, occur less frequent, which gives less training data to the learning-based techniques. When being faced with identifiers that use non-obvious tokenization, most techniques, with the exception of Needleman-Wunsch, perform relatively well. We attribute this result to the fact that techniques that reason about substrings of an identifier, such as FastText [22], do not rely on a specific tokenization approach, such as camel case or snail case, but instead consider character n-grams of the given identifier.

D. RQ 4: Complementarity of Existing Techniques

Our inspection of examples and of specific subsets of identifier pairs shows that different semantic representation techniques work well for different kinds of identifiers. For example, some techniques work better for abbreviations than others. Based on this observation, we hypothesize that the existing semantic representations complement each other. If this hypothesis is correct, combining techniques in such a way that the most suitable set of techniques is used for a given pair of identifiers could represent similarities more accurately than any of the individual techniques.

To validate this hypothesis, we present an ensemble model that combines existing semantic representations. The key idea is to train a model that predicts the similarity of two identifiers based on the similarity scores provided by the existing semantic representations. To this end, the approach queries each of the seven techniques studied in this paper for a similarity score and provides these scores to the model.

To help the model decide what representations to favor for a given pair of identifiers, we also provide to the model a set of features that describe some properties of identifiers. Given two identifiers, the features we consider are:

- The length of these identifiers.
- The number of subtokens in each of the identifiers, based on snail case and camel case conventions.
- The number of words among the subtokens that are not in an English dictionary. The rationale for this feature is to identify abbreviations, which usually are not dictionary words.

Given the seven similarity scores and the features, we train a model that takes the scores and features of a pair as an input, and then that predicts a similarity score for the pair. We train the model in a supervised way, using the ground truth provided in IdBench as the labels for learning. We use an off-the-shelf support vector machine model with the default hyperparameters provided by the underlying library[5].

In practice, one would train the model with all pairs in our benchmark and then apply the trained model to new pairs. To enable us to measure the effectiveness of the model, we here train it with all but one pair, and then apply the trained model to the left-out pair. We repeat this step for each pair and use the score predicted by the model as the score of the combined technique.

Figure 2 shows the results of the combined approach. Combining different semantic representations clearly outperforms all existing techniques. For example, for the large benchmark, the combined approach increases the relatedness, similarity, and contextual similarity of the best individual technique by 6%, 19%, and 5%, respectively. This result confirms our hypothesis that the existing techniques complement each other and shows the benefits of combining them.

IV. Discussion

This section discusses some lessons learned from our study of semantic representations, along with ideas for addressing the current limitations in future work.

a) Neurally learned embeddings accurately represent the relatedness of identifiers: Overall, all neural embeddings considered in our evaluation provide a high agreement with the ground truth provided by the relatedness scores in IdBench. This result shows that embeddings are effective at assigning similar vector representations to identifiers that occur in the same application domain or that are associated in some other way.

b) No existing technique accurately represents the similarity of identifiers: While the best available embeddings are highly effective at representing relatedness, none of the studied techniques reaches the same level of agreement for similarity. In fact, even the best results in Figures 2b and 2c (38%) clearly stay beyond the inter-rater agreement of our benchmark (62%), showing a huge potential for improvement. For many applications of embeddings of identifiers, semantic similarity is crucial. For example, techniques that suggest suitable variable or method names [9], [17] aim for the name that is most similar, not only most related, to the concept represented by the variable or method. Likewise, name-based analyses for finding programming errors [4] or variable misuses [21] aim at identifying situations where the developer uses a wrong but perhaps related variable. Improving the ability of

[5]Class sklearn.svm.SVR from scikit-learn.

semantic representations to accurately represent the similarity of identifiers will benefit these name-based analysis.

c) Neural embeddings generally outperform string distance functions: Our results for both relatedness and similarity show that the best available neural embeddings outperform classical string distance functions. For example, for the relatedness benchmark, the string distance functions achieve up to 49% correlation, whereas embeddings achieve up to 74% correlation. For the similarity and contextual similarity benchmarks, the differences are smaller (32% vs. 38% for similarity, and 29% vs. 36% for contextual similarity), but still clearly visible. These results suggest that name-based analyses are likely to benefit from using embeddings instead of string distance functions.

d) Opposite are challenging: Inspecting examples of (in)accurately represented pairs of identifiers shows that identifiers that describe opposing concepts are particularly challenging for current semantic representations. For example, both the FT-cbow and path-based embeddings assign similar vectors to `minText` and `maxText`, even though these identifiers are clearly not similar but only related. Another example are the `setInterval` and `clearInterval` function names. Table IV shows these and other examples of this phenomenon. Improving semantic representations to better distinguish identifiers with opposing meaning will benefit name-based analyses that, e.g., suggest method names [9] or refactorings of identifiers [12].

e) Distinguishing singular and plural identifiers is particularly challenging: Another challenge we observe while inspecting pairs of inaccurately represented pairs of identifiers is to distinguish identifiers of individual items from identifiers of collections of items. For example, FT-cbow assigns very similar vectors to `substr` and `substrs` (Table IV). Such a conflation of singular and plural concepts may be misleading, e.g., for name-based analyses that predicts types [7], [8], [19].

f) Shared subword information may be misleading: String distance functions and, to some extent, also subword-based embeddings, such as FastText, rely on the assumption that substrings shared by two identifiers increase the chance that the identifiers are semantically similar. While a subword-based approach helps deal with the out-of-vocabulary problem [35], it may also mislead the semantic representation. For example, the FT-cbow embedding assigns similar vectors to `minText` and `minuteText`, as well as to `setInterval` and `clearInterval`, as these identifiers share subwords, even though the identifiers refer to clearly different concepts.

g) Expanding abbreviations may improve semantic representations: The finding that practically all existing semantic representations have difficulties with abbreviations raises the question how to address this limitation. One promising direction is to expand abbreviations into longer identifiers before querying for their relatedness or similarity to another identifier. Several techniques for expanding identifiers have been proposed [36]–[40], which could possibly be used as a preprocessing step within semantic representations.

h) Different semantic representations complement each other: The availability of different techniques for reasoning about the similarity of identifiers can be exploited by combining multiple such techniques. Our ensemble model (Section III-D) shows the potential of combined approaches.

V. Threats to Validity

A. Threats to Internal Validity

Threats to internal validity are about factors that may influence our results. The identifiers and the code examples associated with them may not be representative of other code. To mitigate this threat, we gather data from a large and diverse code corpus, and we select identifiers that cover semantically similar and dissimilar pairs of identifiers (Section II-A0c). The decision to perform our work with code written in a dynamically typed programming language, JavaScript, biases our results toward such languages. The reason for focusing on a dynamically typed language is that such languages are the target of various name-based analyses [4], [7], [8], [19], [41], [42] and embedding techniques [17], [23].

Some ratings gathered our surveys may be inaccurate, e.g., because participants may have misunderstood the instructions. To mitigate this threat, we gather at least ten ratings per pair of identifiers and then carefully clean the ratings gathered by developers to remove noise and outliers (Section II-B).

B. Threats to External Validity

Threats to external validity are about factors that may influence the generalizability of our results. One limitation is that IdBench focuses on individual identifiers only. As a result, it is not clear to what extent our evaluation of semantic representations of identifiers allows for conclusions about representations at a larger granularity, e.g., of complex expressions, statements, or sequences of statements. We focus on individual identifiers as they are the basic building blocks of code. Recent work on improving name-based and learning-based bug detection [4] by aggregating identifiers in complex expressions suggests that improving embeddings for individual identifiers also benefits larger-scale code representations [41].

Another limitation is that other string distance functions or other embeddings may perform better or worse than those studied here. We select semantic representations that have been used in past name-based analyses, as well as some recent embedding techniques that are state of the art in natural language processing (NLP). By making IdBench publicly available, we enable others to evaluate future semantic representations. As any benchmark, IdBench consists of a finite set of subjects, which may not be representative for all others. The number of pairs of identifiers in the benchmark (Table II) is in the same order of magnitude as that of word similarity benchmarks used in NLP [20], [31], [43]–[45]. Finally, we focus on JavaScript code, i.e., our findings may not generalize to identifiers in other languages.

Finally, different name-based analyses have different requirements on the semantic representations they build upon. The tasks we present to survey participants may not represent

all these requirements, and hence, a semantic representation may perform better or worse in a specific name-based analysis than IdBench suggests.

VI. Related Work

a) Name-based Program Analysis: Various analyses exploit the rich information provided by identifier names, e.g., to find bugs [2]–[5] and vulnerabilities [47], to mine specifications [6], to infer types based on identifier names as implicit type hints [7], [8], to predict the name of a method [9], to complete partial code using a learned language model [10], to identify inappropriate names [11], to suggest more suitable names [12], to resolve fully qualified type names of methods, variables, etc. in a given code snippet [48], or to map APIs between programming languages based on an embedding of code tokens [18]. A systematic way of evaluating semantic representations of identifiers, as provided in this paper, helps in further exploiting the implicit knowledge encoded in identifiers, and hence will benefit name-based program analyses.

b) Embeddings of Identifiers: Embeddings of identifiers are at the core of several code analysis tools. A popular approach, e.g., for bug detection [4], type prediction [8], or vulnerability detection [47], is applying Word2vec [26], [34] to token sequences, which corresponds to the Word2vec embedding evaluated in Section III. [49] train an RNN-based language model and extract its final hidden layer as an embedding of identifiers. Chen et al. [50] provide a more comprehensive survey of embeddings for source code. Beyond learned embeddings, string distance functions are used in other name-based tools, e.g., for detecting bugs [2], [3] or for inferring specifications [6]. The quality of embeddings is crucial in these and other code analysis tools, and IdBench will help to improve the state of the art.

c) Embeddings of Programs: Beyond embeddings of identifiers, there is work on embedding larger parts of a program. One approach [9] uses a log-bilinear, neural language model [51] to predict the names of methods. Other work embeds code based on graph neural networks [21] or sequence-based neural networks applied to paths through a graph representation of code [23], [52]–[56]. Code2seq embeds code and then generates sequences of NL words [57]. For a broader overview and a detailed survey of learning-based software analysis, we refer the reader to [16] and [58], respectively. To evaluate embeddings of programs, the COSET benchmark provides thousands of programs with semantic labels [59]. Another study measures how effective pre-trained code2vec [23] embeddings are for different downstream tasks [46]. One conclusion from Kang et al.'s work [46] is that evaluating embeddings on a specific downstream task is insufficient, a problem we here address with a task-independent benchmark. Both of the above [46], [59] complement IdBench because the existing work is about entire programs, whereas IdBench is about identifiers. Since identifiers are a basic building block of source code, a benchmark for improving embeddings of identifiers will eventually also benefit learning-based code analysis tools.

d) Benchmarks of Word Embeddings: The NLP community has a long tradition of reasoning about the semantics of words. In particular, that community has addressed the challenge of measuring how well a semantic representation of words matches actual relationships between words through a series of gold standards of words, focusing on either relatedness [25], [43], [44] or similarity [20], [31], [45], [60] of words. These gold standards define how similar two words are based on ratings by human judges, enabling an evaluation that measures how well an embedding reflects the human ratings.

Unfortunately, simply reusing these existing gold standards for identifiers in source code would be misleading. One reason is that the vocabularies of natural languages and source code overlap only partially, because source code contains various terms and abbreviations not found in natural language texts. Moreover, source code has a constantly growing vocabulary, as developers tend to quickly invent new identifiers, e.g., for newly emerging application domains [35]. Finally, even words present in both natural languages and source code may differ in their meaning due to computer science-specific terms, e.g., "float" or "string". This work is the first to address the need for a gold standard for identifiers in code.

e) Data Gathering: Asking human raters how related or similar two words are was first proposed by [45] and then adopted by others [20], [31], [43], [60]. Our direct survey also follows this methodology. [20] propose to gather judgments about contextual similarity by asking participants to choose a word to fill in a blank, an idea we adopt in our indirect survey. To choose words and pairs of words, prior work relies on manual selection [45], pre-existing free association databases [31], [60], e.g., USF [61] or VerbNet [62], [63], or cosine similarities according to pre-existing models [25]. We follow the latter approach, as it minimizes human bias while covering a wide range of degrees of relatedness and similarity.

f) Inter-rater Agreement: Validating and cleaning data gathered via crowd-sourcing based on the inter-rater agreement has been found effective in other crowd-sourced surveys [30]. Gold standards for natural language words reach an inter-rater agreement of 0.61 [43] and 0.67 [31]. Our "small" dataset reaches similar levels of agreement, showing that the rates in IdBench represent a genuine human intuition. As noted by [31], the inter-rater agreement also gives an upper bound of the expected correlation between the tested model and the gold standard. Our results show that current models still leave plenty of room for improvement, especially w.r.t. similarity.

VII. Conclusion

This paper presents the first benchmark for evaluating semantic representations of identifiers names, along with a study of current semantic representation techniques. We compile thousands of ratings gathered from 500 developers into a benchmark that provides gold standard similarity scores representing the relatedness, similarity, and contextual similarity of identifiers. Using IdBench to experimentally compare two string distance functions and five embedding techniques shows that these techniques differ significantly in their agreement

with our gold standard. The best available embeddings are effective at representing how related identifiers are. However, all studied techniques show huge room for improvement in their ability to represent how similar identifiers are. An in-depth study of different subsets of identifiers shows the specific strengths and weaknesses of current semantic representations, e.g., that most techniques are challenged by abbreviations, opposites, and the difference between singular and plural. To exploit the complementarity of current techniques, we present an ensemble model that effectively combines them and clearly outperforms the best individual techniques.

Our work will help addressing the limitations of current semantic representations of identifiers. Such progress will benefit downstream developer tools, in particular, name-based program analyses. More broadly, improving semantic representations of identifiers will also contribute toward better learning-based program testing and analysis techniques.

REFERENCES

[1] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2010, pp. 156–165.

[2] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 232–242.

[3] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[4] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *PACMPL*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018. [Online]. Available: https://doi.org/10.1145/3276517

[5] S. Kate, J. Ore, X. Zhang, S. G. Elbaum, and Z. Xu, "Phys: probabilistic physical unit assignment and inconsistency detection," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 563–573. [Online]. Available: https://doi.org/10.1145/3236024.3236035

[6] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *International Conference on Automated Software Engineering (ASE)*, 2009, pp. 307–318.

[7] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 607–618. [Online]. Available: https://doi.org/10.1145/2950290.2950343

[8] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript function types from natural language information," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 304–315. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00045

[9] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 38–49.

[10] M. R. Parvez, S. Chakraborty, B. Ray, and K. Chang, "Building language models for text with named entities," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, 2018, pp. 2373–2383. [Online]. Available: https://www.aclweb.org/anthology/P18-1221/

[11] E. W. Høst and B. M. Østvold, "Debugging method names," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2009, pp. 294–317.

[12] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 1–12. [Online]. Available: https://dl.acm.org/citation.cfm?id=3339507

[13] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2011, pp. 130–154.

[14] Y. Jiang, H. Liu, J. Q. Zhu, and L. Zhang, "Automatic and accurate expansion of abbreviations in parameters," *IEEE Transactions on Software Engineering*, 2018.

[15] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *International Conference on Software Engineering (ICSE)*, 2016, pp. 1063–1073.

[16] M. Pradel and S. Chandra, "Neural software analysis," *CoRR*, vol. abs/2011.07986, 2020. [Online]. Available: https://arxiv.org/abs/2011.07986

[17] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *ACM SIGPLAN Notices*, vol. 53, no. 4. ACM, 2018, pp. 404–419.

[18] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 438–449.

[19] V. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *FSE*, 2018.

[20] G. A. Miller and W. G. Charles, "Contextual correlates of semantic similarity," *Language and cognitive processes*, vol. 6, no. 1, pp. 1–28, 1991.

[21] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[22] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *TACL*, vol. 5, pp. 135–146, 2017. [Online]. Available: https://transacl.org/ojs/index.php/tacl/article/view/999

[23] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[24] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 761–774.

[25] E. Bruni, N.-K. Tran, and M. Baroni, "Multimodal distributional semantics," *Journal of Artificial Intelligence Research*, vol. 49, pp. 1–47, 2014.

[26] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[27] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *OOPSLA*, 2016.

[28] M. Allamanis and C. A. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 207–216.

[29] A. Kittur, E. H. Chi, and B. Suh, "Crowdsourcing user studies with mechanical turk," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2008, pp. 453–456.

[30] S. Nowak and S. Rüger, "How reliable are annotations via crowdsourcing: a study about inter-annotator agreement for multi-label image annotation," in *Proceedings of the international conference on Multimedia information retrieval*, 2010, pp. 557–566.

[31] F. Hill, R. Reichart, and A. Korhonen, "Simlex-999: Evaluating semantic models with (genuine) similarity estimation," *Computational Linguistics*, vol. 41, no. 4, pp. 665–695, 2015.

[32] V. Zhelezniak, A. Savkov, A. Shen, and N. Y. Hammerla, "Correlation coefficients and semantic textual similarity," *arXiv preprint arXiv:1905.07790*, 2019.

[33] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[34] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their composi-

tionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[35] H. Babii, A. Janes, and R. Robbes, "Modeling vocabulary for big code machine learning," *CoRR*, 2019. [Online]. Available: https://arxiv.org/abs/1904.01873

[36] A. Corazza, S. D. Martino, and V. Maggio, "LINSEN: an efficient approach to split identifiers and expand abbreviations," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSM.2012.6405277

[37] Y. Jiang, H. Liu, and L. Zhang, "Semantic relation based expansion of abbreviations," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 131–141. [Online]. Available: https://doi.org/10.1145/3338906.3338929

[38] C. D. Newman, M. J. Decker, R. S. Alsuhaibani, A. Peruma, D. Kaushik, and E. Hill, "An empirical study of abbreviations and expansions in software artifacts," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 269–279. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00040

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

[40] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," in *Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2007, pp. 213–222.

[41] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," 2020. [Online]. Available: https://openreview.net/pdf?id=ryxnJlSKvr

[42] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: Neural type prediction with search-based validation," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, 2020, pp. 209–220. [Online]. Available: https://doi.org/10.1145/3368089.3409715

[43] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin, "Placing search in context: The concept revisited," *ACM Transactions on information systems*, vol. 20, no. 1, pp. 116–131, 2002.

[44] T. Schnabel, I. Labutov, D. M. Mimno, and T. Joachims, "Evaluation methods for unsupervised word embeddings," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, 2015, pp. 298–307. [Online]. Available: http://aclweb.org/anthology/D/D15/D15-1036.pdf

[45] H. Rubenstein and J. B. Goodenough, "Contextual correlates of synonymy," *Communications of the ACM*, vol. 8, no. 10, pp. 627–633, 1965.

[46] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *ASE*, 2019.

[47] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *CoRR*, vol. abs/1803.04497, 2018. [Online]. Available: http://arxiv.org/abs/1803.04497

[48] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen, "Statistical learning of API fully qualified names in code snippets of online forums," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 632–642. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180230

[49] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE*, 2016, pp. 87–98.

[50] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *arXiv preprint arXiv:1904.03061*, 2019.

[51] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[52] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *CoRR*, vol. abs/1806.07336, 2018. [Online]. Available: http://arxiv.org/abs/1806.07336

[53] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *CoRR*, vol. abs/1710.11054, 2017. [Online]. Available: http://arxiv.org/abs/1710.11054

[54] J. Henkel, S. K. Lahiri, B. Liblit, and T. W. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 163–174.

[55] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," *CoRR*, vol. abs/1802.07779, 2018.

[56] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017, pp. 363–376.

[57] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[58] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[59] K. Wang and M. Christodorescu, "Coset: A benchmark for evaluating neural program embeddings," *CoRR*, 2019. [Online]. Available: https://arxiv.org/abs/1905.11445

[60] D. Gerz, I. Vulić, F. Hill, R. Reichart, and A. Korhonen, "Simverb-3500: A large-scale evaluation set of verb similarity," *arXiv preprint arXiv:1608.00869*, 2016.

[61] D. L. Nelson, C. L. McEvoy, and T. A. Schreiber, "The university of south florida free association, rhyme, and word fragment norms," *Behavior Research Methods, Instruments, & Computers*, vol. 36, no. 3, pp. 402–407, 2004.

[62] K. Kipper, B. Snyder, and M. Palmer, "Extending a verb-lexicon using a semantically annotated corpus." in *LREC*, 2004.

[63] K. Kipper, A. Korhonen, N. Ryant, and M. Palmer, "A large-scale classification of english verbs," *Language Resources and Evaluation*, vol. 42, no. 1, pp. 21–40, 2008.