Leveraging a Corpus of Natural Language Descriptions for Program Similarity

Meital Zilberstein
Technion, Israel
mbs@cs.technion.ac.il

Eran Yahav
Technion, Israel
yahave@cs.technion.ac.il

Abstract

Program similarity is a central challenge in many programming-related applications, such as code search, clone detection, automatic translation, and programming education.

We present a novel approach for establishing the similarity of code fragments by: (i) obtaining textual descriptions of code fragments captured in millions of posts on question-answering sites, blogs and other sources, and (ii) using natural language processing techniques to establish similarity between textual descriptions, and thus between their corresponding code fragments. To improve precision, we use a simple static analysis that extracts type signatures, and combine the results of textual similarity with similarity of the signatures. Because our notion of code similarity is based on similarity of textual descriptions, our approach can determine semantic relatedness and similarity of code across different libraries and even across different programming languages, a task considered extremely difficult using traditional approaches. To evaluate our approach, we use data obtained from the popular question-answering site, STACKOVERFLOW. To obtain a ground-truth to compare against, we developed a crowdsourcing system, LIKE2DROPS, that allows users to label the similarity of code fragments. We used the system to collect similarity classifications for a massive corpus of $6{,}500$ program pairs. Our results show that our technique is effective in determining similarity, and achieves more than 85% precision, recall and accuracy.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Code Similarity, Natural Language, Program Analysis, Semantics

Onward!'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4076-2/16/11...$15.00
http://dx.doi.org/10.1145/2986012.2986013

1. Introduction

We address the problem of similarity between code fragments. Code similarity and equivalence are classic problems in programming languages and software engineering [5, 10]. There has been a lot of work on syntactic code similarity and clone detection [3, 4, 23]. However, most approaches cannot capture similarity across programs using different APIs or algorithms, let alone programming languages. There has also been a lot of work on semantic equivalence and differencing of programs [15, 36]. However, most of these approaches require that the programs being compared not be too different from one another (e.g., different versions of the same program with small patches [36], or the same program at different abstraction levels [39]).

Furthermore, in some cases, equivalence is not what we are looking for. Our goal is to capture connections between snippets, such as semantic similarity or relatedness, which are more relaxed notions than strict equivalence. We further aim to capture these connections for code fragments using different libraries and languages.

***Our Approach: Combining* Big Code *and* Natural Language Processing.** Inspired by Hindle et al. [14], we explore how *"big code"* (large scale code repositories) and *natural language processing* (NLP) can help us address the problem of semantic relatedness of code fragments. The challenge is to establish such semantic similarity (more generally, semantic relatedness) automatically.

The main idea of this paper is to tackle the problem of semantic relatedness between code fragments by considering the semantic relatedness of their corresponding textual descriptions. Doing so requires: (i) establishing a relationship between a code fragment and its textual description(s), (ii) measuring semantic similarity between textual descriptions, and (iii) creating comparable snippet representations

To address the first challenge, we rely on relationships between code and descriptions as created, for example, on common Q&A sites such as STACKOVERFLOW (SO), in documentation, on blog posts, and in comments or method and variable names. We also match fragments that are not in the database to existing syntactically similar fragments.

To address the second challenge, we utilize state-of-the-art text similarity methods. The third challenge is dealt by a dataflow analysis that extracts type information and improves the precision of relatedness results. This approach might seem surprising because NL seems to be less constrained space than code; however, we target SOF posts that use only a small lexicon and contain code fragments that can serve as anchors. Hence they are semi-structured.

We draw inspiration from previous research in the area of image retrieval [50, 52], where image similarity can be determined by comparing tags associated with images and not only by visual similarity. Our approach represents a radical departure from standard techniques that only use the code, and takes an "open world" approach that leverages collective knowledge.

Main Contributions. The contributions of this paper are:

- A framework for semantic relatedness of code, based on similarity of corresponding natural language descriptions and type signatures. Our approach effectively and efficiently determines quantitative similarity between code fragments using different libraries and code fragments in different programming languages.
- An implementation of our approach in a tool called SIMON, and an evaluation over 100,000 pairs of code fragments in multiple programming languages, showing that it can handle a wide variety of code fragments.
- A crowdsourcing platform in which users can classify the relatedness of pairs; results collected from 50 users are used to classify a sample of 6,500 pairs. The labeled data is of independent interest for researchers in this area.
- When compared against labeled data, our approach obtains more than 85% precision, recall and accuracy. This accuracy makes our approach a valuable complement to techniques based purely on semantic similarity of code.

2. Overview

In this section, we provide an informal, high-level overview of SIMON using an example.

2.1 Motivating Example

Consider the two code fragments in Figure 1. Both generate permutations of a given string. The code fragment in Figure 1(a) is written in Python and the one in Figure 1(b) is written in Java. Despite considerable syntactic difference and the different programming languages, we would like to say that the two are similar: both fragments generate permutations with the slight difference that (a) performs printing and (b) returns the result.

Efforts to capture this similarity via syntactic approaches, such as comparison of strings or *abstract syntax trees* (AST), will fail due to the large differences in the syntax of these languages and their use of two different computation structures. Even semantic approaches that are based on input-output relations will have difficulty finding the similarity be-

```python
def p (head, tail=''):
  if len(head) == 0:
    print tail
  else:
    for i in range(len(head)):
      p(head[0:i] + head[i+1:], tail + head[i])
p("abc")
```

(a)

```java
public static Set<String> generateP(String input) {
  Set<String> set = new HashSet<String>();
  if (input == "") return set;
  Character a = input.charAt(0);
  if (input.length() > 1) {
    input = input.substring(1);
    Set<String> permSet = generateP(input);
    for (String x : permSet)
      for (int i = 0; i <= x.length(); i++)
        set.add(x.substring(0, i) + a + x.substring(i));
  } else {
    set.add(a + "");
  }
  return set;
}
```

(b)

Figure 1. Semantically related fragments for generating permutations of a string: (a) written in Python and (b) in Java.

cause Figure 1(a) holds concrete values (line 7) and Figure 1(b) expects to get them as an input (line 1). Moreover, the use of language-specific operations (e.g., `range` in Python, `charAt` in Java) adds another layer of difficulty. We present an approach that sidesteps these challenges.

Intuition. Figure 2 shows a simplified view of our approach (the actual system architecture is shown in Figure 3). Our goal is to determine semantic similarity between the code snippets s and s'. We do so by mapping the snippets to corresponding textual descriptions and computing the semantic similarity between them. To map s and s' to textual descriptions, we first use syntactic similarity to find representative snippets $\hat{s_1}$,$\hat{s_2}$ and $\hat{s_1}'$,$\hat{s_2}'$,$\hat{s_3}'$ in a pre-computed database of code fragments with corresponding textual descriptions. We utilize the similarity between the textual descriptions to determine the semantic similarity of code fragments. Finding representatives in the description mapping allows us to compare code fragments even when they are not in our pre computed description mapping.

The database is queried for similarity using the steps:
Step 1 - Linking Code to Textual Descriptions: The first step is to link the code fragments to their corresponding textual descriptions. Table 1 shows the STACKOVERFLOW questions which correspond to the code fragments of Figure 1. We use a syntactic code similarity technique to link a given code fragment to one that has a corresponding textual description. Section 6.1.1 elaborates on the retrieval method we use to find these descriptions, and Section 5.4 specifies the text processing steps used to create meaningful descriptions. This step is based on the observation that pieces of knowledge close together in cyberspace tend to be related [2].
Step 2 - Extracting Type Signatures: We use static analysis to extract type signature (inputs $\rightarrow$ outputs) from a code frag-

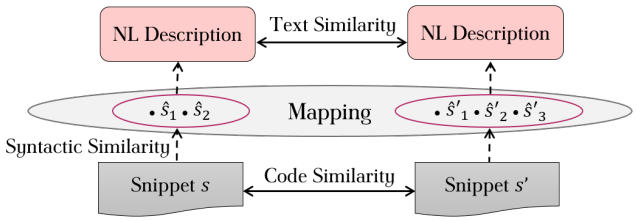

Figure 2. SIMON's core: snippets are mapped to corresponding natural language elements and the mapping is used to determine the snippets' semantic similarity.

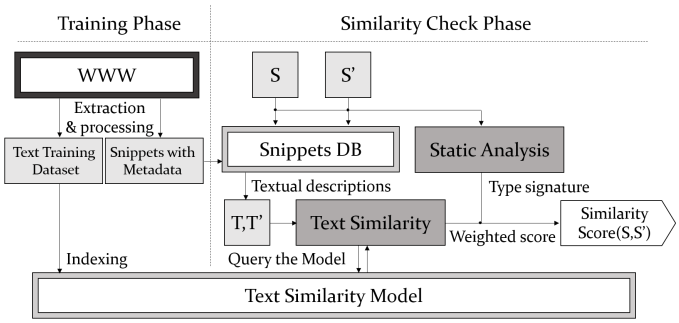

Figure 3. The architecture of SIMON.

	Snippet (a)
Title	How to generate all permutations of a list in Python
Content	How do you generate all the permutations of a list in Python, independently of the type of elements in that list? For example: ⟨ *some code* ⟩
Tags	python, algorithm, permutation, combinatorics, python-2.5
Votes	171
	Snippet (b)
Title	Generating all permutations of a given string
Content	What is an elegant way to find all the permutations of a string. E.g. *ba*, would be *ba* and *ab*, but what about *abcdefgh*? Is there any example Java implementation?
Tags	java, algorithm
Votes	124

Table 1. Information about the code fragments in Figure 1.

ment. Section 5.5 provides more details about our definition of type signature. This step is used to improve the precision of the text-based similarity method.

In our example, for the snippet of Figure 1(a), we extract the type signature String × Iterable → String, and for the snippet of Figure 1(b) we get String → Set⟨String⟩.

Step 3 - Comparison: The third step is the comparison between the new entities. For simplicity, we consider the title as the only code description; however, SIMON uses the content of the entire question. The title texts are closely related, and indeed, their similarity score is higher than 0.79, indicating strong similarity. This value was computed using a state-of-the art text similarity approach, specialized for the programming domain, as explained in Section 5.4

The strong affinity between the type signatures allows us to increase the confidence of the text similarity score.

Key Aspects. This example demonstrates:

- **Similarity across languages:** SIMON can determine similarity of programs written in different programming languages with different syntax.
- **Similarity across libraries:** SIMON determines similarity when using different libraries.
- **Similarity between partial programs:** SIMON determines similarity of programs where input and output are not explicitly defined (no return or explicit parameters).

Applications. The ability to capture similarities between code fragments written in different or similar programming languages has numerous applications: (i) code search using other semantically similar code fragments or a natural language description, (ii) automatic translation between code fragments in different programming languages, (iii) clone detection, (iv) learning of different implementation techniques, (v) automatic tagging of programs with the words that best describe them, and (vi) NLP enrichment based on the similarity classifications obtained from LIKE2DROPS.

3. Background

In this section, we provide the background for the text similarity techniques that are used throughout this paper.

3.1 Text Similarity

Determining the similarity between two texts is one of the main problems in many NLP and *information retrieval* (IR) applications [53]. The *vector space model* (VSM) [42] based methods are the most widely used today. In this approach, any text is represented by a n-dimensional vector, where n is the number of different terms that were used to index the training set (the sub-group of texts that is used to train the model). Each cell is associated with a weight that indicates its importance and can be calculated in certain ways.

Example. We use two semantically similar sentences to illustrate the use of different text similarity methods.

(a) How can we order a list using Python?
(b) In Java, I want a list to be sorted.

3.1.1 TF.IDF

The *term frequency inverse document frequency* (TF.IDF) is a standard VSM based method. It combines the *term frequency* (tf) – the number of occurrences of a term in a document, and the *inverse document frequency* (idf) – indication of the uniqueness of a term across all documents,

each of which can be computed in many ways. We use the natural tf, and logarithmic and smooth idf, as shown in (1).

$$\text{idf}_t = \log(\frac{|D|}{|D_t|+1}) + 1, \quad (1)$$

where $|D|$ is the number of documents and $|D_t|$ is the number of documents that contain the term t. The equation for computing a vector cell t for document d is $t_{dt} = \text{tf}_{dt}\text{idf}_t$ [55]. One drawback of this approach is the lack of semantic knowledge. Going back to our example, the actual similarity value depends on the training set. However, the sentences share only two common words. One of these words is "a", which is not unique and gets low idf. The result is that the two sentences have a low similarity score.

3.1.2 Latent Semantic Analysis

Latent semantic analysis (LSA), also called *latent semantic indexing* (LSI), is another VSM based method that utilizes the latent structure of the text to overcome the lack of semantic knowledge that TF.IDF suffers from: instead of relying only on the words that people choose, it statistically captures their usage patterns. Consequently, it can attribute similarity to different words with similar meaning.

LSA is based on a matrix of terms-documents and on a mathematical model known as *singular value decomposition* (SVD). Any rectangular matrix, M, is decomposed to the product of three other matrices, such that $M = X_0 S_0 Y_0$ when X_0 and Y_0 have orthonormal columns (orthogonal and unit length) and S_0 is diagonal. Due to the existence of many negligible elements in the base term-document matrix, the matrices can be simplified by deletion of the smallest elements (e.g. rows and columns) in S_0 and the corresponding columns of X_0 and Y_0. In other words, only the k most important elements remain. Deerwester et al. [9] further elaborate on the details of SVD. These matrices are used to create the semantic space, wherein closely associated elements are placed near each other. LSA captures the arrangement of the space such that even terms that don't appear in a text may end up around it if they follow a major association pattern. Terms and documents are represented by a vector of factor values derived from the simplified matrices. Like in TF.IDF, the actual similarity value of the example sentences depends on the selected training set, however, the connection between the words "order" and "sorted" is captured by this model and a higher similarity score is obtained.

3.1.3 Align, Disambiguate, and Walk (ADW)

A contemporary, unified semantic representation called ADW was presented by Pilehvar et al. [38]. It leverages a common probabilistic representation over the senses of a word. ADW is based on WordNet 3.0 [30] as its sense inventory and it produces a multinomial frequency distribution by repeated random walks on the WordNet ontology graph. The resulting representation is a probability vector, which aggregates the similarities of the text over the entire graph. It also uses an alignment based disambiguation method and shows state-of-the-art performance. Using ADW with our example sentences, we first remove stop words using an English stop word list expanded with more than 400 programming language names and tag each word with its part of speech. For example, (a) is [order-verb, list-noun] and (b) is [want-verb, list-noun, sort-verb]. Afterwards, the sentences are aligned using WordNet, resulting in a connection between the words *order* and *sort*, and in a relatively high similarity score.

3.2 Cosine Similarity

The *cosine similarity* function [28] is widely used to find the similarity between two given vectors, computed by one of the aforementioned techniques. In cosine similarity the vectors are normalized to unit length and the angle has the only influence on the similarity score. Hence, it is not biased by different text sizes. Cosine similarity is calculated by taking the inner product $(v_1 \bullet v_2)$ of the vectors and dividing it by the product of their vector lengths.

The similarity between two vectors is computed with *cosine similarity* based on the following equation:

$$\text{cosine}(v_1, v_2) = \frac{(v_1 \times v_2)}{\|v_1\|\|v_2\|}, \quad (2)$$

where $\|v\|$ is the Euclidean norm of the vector v.

4. Leveraging Collective Knowledge

In this section, we describe how SIMON leverages collective knowledge towards a similarity notion.

4.1 Open World Approach

Sites such as GITHUB, STACKOVERFLOW, and programming blogs store massive amounts of code, often with additional meta-data. Programmers interact by sharing code or asking questions and create associations between code and its natural language description. STACKOVERFLOW (SO) is a community question-answering site that allows programmers to post any programming related question. Each question is associated with a title, content and tags, chosen by the author to describe the question. SO creates an implicit mapping between code fragments and their descriptions.

For our analysis, we used the SO dump from September 2014, provided by the MSR challenge [54]. The dump contains more than $21M$ posts, divided into around $8M$ questions and more than $13M$ answers.

Finding Representatives. This work presents a radical departure from common approaches for determining similarity between code fragments. A major challenge for our approach is the ability to find representatives in the precomputed database for an arbitrary pair of snippets provided by a user. Indeed, the quality of our results may vary widely in accordance with the quality of representatives.

Covering all possible code fragments is impossible, and it is likely that for some rare code fragments we will not find

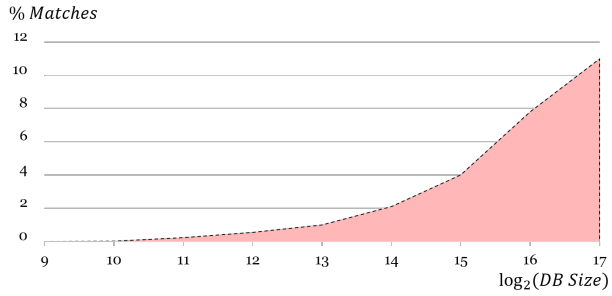

Figure 4. The percentage of code fragments that have syntactically syntactically similar code in the database, as a function of the database size.

good representatives. However, two factors convince us that our approach holds great potential:

1. Software is not unique. As observed by Gabel et al. [12]:

> "Although the number of legal statements in the language is theoretically infinite, the number of practically useful statements is much smaller, and potentially finite."

That is, software contains many parts that are repetitive. If we map these, we may get good coverage of commonly used code fragments. There are more than $16M$ STACKOVERFLOW posts that contain code fragments. These can be utilized for the construction of a rich mapping.

To deal with the "unique" parts, such as variable names or concrete values, we use syntactic similarity that can ignore names and concrete values. This helps us link new snippets to representatives we already have in our mapping. However, we cannot guarantee good representatives for any arbitrary rare code fragment that one might think of.

2. Data increases rapidly. For commonly used code snippets we show that the approach is already feasible. We believe that as this kind of data grows (there are around $15{,}000$ new answers on STACKOVERFLOW every day!), our approach will become more attractive and more widely applicable. Figure 4 shows the percentage of code fragments from the database that have another syntactically similar fragment in the database (rather than the same fragment itself), as a function of the database size. The graph illustrates the importance of a vast database and shows that even within a *limited database* (for which we measured pairwise similarity) we can find duplicates. Note that we show internal syntactic similarity between fragments in the database just to illustrate the likelihood of finding syntactically similar code.

This observation highlights our expectation that new code fragments will be found in our database as it grows. We can see that the larger the database, the more adjacent fragments are found; this, alongside the rapid growth of sites such as STACKOVERFLOW, emphasizes the potential of our approach.

5. Description Based Relatedness

In this section, we describe the technical details of SIMON.

5.1 Similarity Metric Overview

At a high-level, we define the similarity between two given code snippets s, s', using the similarity between their corresponding snippets in our database. This metric can be the basis of a ranking algorithm for code search.

$$sim_{code}(s,s') = \alpha \cdot \max_{\substack{d(s,\hat{s}) < \epsilon, \\ d(s',\hat{s}') < \epsilon}} \widehat{sim}(\hat{s},\hat{s}') + (1-\alpha) \cdot sim_{sig}(s,s') \tag{3}$$

where

$$\widehat{sim}(\hat{s},\hat{s}') = \beta \cdot sim_{title}(\hat{s},\hat{s}') + (1-\beta) \cdot sim_{content}(\hat{s},\hat{s}'), \tag{4}$$

Fragments $\hat{s}$,$\hat{s}'$ are syntactically similar to s,s', respectively, and $\alpha, \beta \in [0,1]$.

To compute these metrics, we rely on the following:

- Finding corresponding snippets in our database. This is done by syntactic matching of the snippets based on a distance d (either AST comparison or alignment) being smaller than some threshold ϵ.
- sim_{title}, which uses ADW similarity algorithm applied to titles that have been enriched with software-oriented query expansion.
- $sim_{content}$, which uses LSA similarity that has been trained on millions of textual software descriptions.
- sim_{sig}, which measures similarity between type signatures that was revealed in type flow analysis.

5.2 Semantic Relatedness of Code Fragments

The term *semantic relatedness* first appeared in the NLP domain. We adapt and modify this term for the programming domain. In NLP, *semantic relatedness* is the finer case of *semantic similarity*, where similarity is based on *is-a* relations and can't be established across different parts of speech. The notion of relatedness captures similarity, but also includes other relations, such as: *has-part, is-made-of* [37]. We use the modified term relatedness to refer to similarity between two code fragments based on their functionality. Our notion of relatedness includes equivalent or similar behaviors of the programs, inclusion or opposite functionalities. It not only captures binary decisions that define whether the programs are similar, but a quantitative spectrum of similarities.

5.3 Syntactic Similarity

Assuming we have a *description oracle* that maps code to its corresponding textual descriptions, we need to be able to look for new code and find its matches in order to borrow their textual descriptions. Good matches are code fragments that can share the same textual description. Towards that end, we look for syntactically similar code fragments. In this

```
ip = "192.24.1.17"
InetAddress x = InetAddress.getByName(ip);
String h = x.getHostName();
System.out.println(h);
```

(a)

```
InetAddress addr =
    InetAddress.getByName("173.194.36.37");
String host = addr.getHostName();
System.out.println(host);
```

(b)

Figure 5. Two syntactically similar code fragments.

work we refer to two code fragments as syntactically similar in accordance with the notion of clone types $1-3$ [18]. Mostly we refer to changes in formatting, naming or literals; however, a change can be broader as long as it preserves the flow and functionality of the original program.

From Code Fragment to Description. Given two arbitrary code fragments s, s', and a description oracle containing code fragments with their textual descriptions, we would like to compute the semantic similarity $sim_{code}(s,s')$ between s and s', or the opposite notion: $d(s,s') = 1 - sim_{code}(s,s')$. Often, the snippets don't appear as in the oracle, but with some negligible differences, such as naming, formatting or concrete values. To compute their similarity we use standard techniques for *syntactic similarity* of code fragments, as explained below. Specifically, we can find similar fragments in the oracle's database even when they differ in variable names or parameter values. Monperrus et al. investigated whether the SO search engine can handle code snippets as input. They searched for code, extracted from the site, but only 15% of the snippets from their dataset were found [32]. These results along with our dealing with multiple programming languages and our desire to find snippets that don't necessarily appear in the exact same way led us to consider other options. We suggest two similar but not identical techniques to deal with this problem: (i) the first treats the code as text and therefore works for all programming languages, and (ii) the second is language specific and based on the syntax of the chosen language.

Generic Syntactic Similarity. To deal with multiple programming languages we extract only the relevant tokens from each fragment – types, functions, keywords, special characters, etc., while preserving their order. Our approach is based on the common parts between the desired code and any code fragment from the search group. Initially, we use a standard keyword matching technique to find the group of possible matches, followed by alignment of the common tokens. As an example of syntactically similar fragments, consider the two in Figure 5.

Language-Specific Syntactic Similarity. The first approach yields good results, tested on a subset of our database; however, we suggest another technique, customized for code.

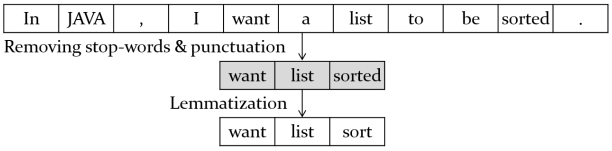

Figure 6. Text processing using a sentence from Section 3.

This technique is language specific and based on AST. For each snippet, we create a representative string. The string from the snippet in Figure 8 is:

Assign Name Store Call $_{urlopen}$ *Attribute Name Load Load Str Assign Name Store Call* $_{read}$ *Attribute Name Load Load*

This string captures only the important information about the code: order, API calls, arithmetics, etc. Identifiers and concrete values are not captured in this representation. Then, when searching for new code within our database, we need to compute its representative string, after which we look for almost perfect matches, using ideas similar to those of the first approach. Mismatches of calls to built-in or common library functions are given a higher penalty.

5.4 Semantic Similarity of Descriptions

The question of semantic similarity between text documents is central in many NLP tasks, and is studied extensively in the NLP community. We assume the existence of a *description oracle*, which takes a code fragment and returns a set of natural-language documents describing its functionality. We refer to the set of natural-language documents related to a code fragment as its *semantic description*. We measure the distance between two *semantic descriptions* using text similarity methods. First, we take the *semantic description* and process it by lemmatizing (changing each word to its base form), removing stop words and punctuation signs. Figure 6 shows a text processing demonstration. Pilehvar et al. [38] claim that techniques such as LSA are still most suitable for long texts. Hence, we decided to combine various text similarity techniques. For the titles, which are short and play a major role in the description, we use ADW with software specialized vocabulary as described in Section 5.4.1. For the question content, which is a longer text, we use LSA on top of TF.IDF. LSA was trained on top of more than $1M$ SO posts and with a reduction to 300 dimensions. This value has been empirically chosen and has been shown before to be effective in practice. [26]. We compare the constructed vectors using *cosine similarity*.

5.4.1 Specialized NLP

Programming has many domain specific terms, such as *String* or *Int*. WordNet based approaches such as ADW might have difficulty dealing with these unique words and produce incorrect results. To deal with this problem we apply a *query expansion* approach that is based on data from a software-specific word similarity database [47]. For each

compared text, we look for words that originate from the programming domain. Each such word is searched for within the database and its two most related words (if exist) are added to the text. This way, we specialize the NLP techniques for programming; words that are similar only in this context are linked and increase the texts' similarity score. For example, the words *"text"* and *"string"* are considered as similar in the programming domain, a connection we might miss if we use only standard WordNet.

5.5 Type Signatures

While keeping in mind the problem of similarity across programming languages, we wanted to utilize the code itself to support the text-based similarity function. Towards that end, we use type signatures as another measure for similarity. Ideally, we would like to use type signatures as a strong source of information for similarity. However, there are several challenges that make this harder than it seems at first glance: (i) multiple signatures in the same snippet: for snippets that consist of a single method or function, it may be possible to extract a clean type signature. However, this is not true for an arbitrary snippet that may contain several different functions, and thus also multiple signatures. (ii) different types across programming languages: a language's set of types, especially when dealing with an OOP language, might be infinite and does not necessarily correspond to other languages. In this paper, we take a rather simple approach to the problem, and only consider type signatures of short snippets, which mostly have single and primitive type signatures. This contributes some improvement to the precision, but not a substantial one. In future work, we plan to treat signatures more comprehensively in the hope of further improving.

Motivation. Sometimes functionally different code fragments hold semantic descriptions that might be classified as similar. Here is an example where using type information is critical. Consider two programs: the first converts a byte array to a string and the second does the opposite. Figure 7 shows the programs. Their partial descriptions (only titles) are the following (respectively):

(i) How do you convert a byte array to a hex string;
(ii) Convert a string representation of a hex to a byte array;

These descriptions share many common words; hence they get a high similarity score. However, their associated programs are not the same. The use of type signatures reveals that the two are indeed different and decreases their score.

Example. Consider the code fragment from Figure 8. The code finds, for a given string (that represents an URL address), its html source. The string is hardcoded but the functionality of the code, the fact that it finds the html source, is the thing we want to isolate. In Figure 9 we can see the data flow of this code fragment. The string literal is not affected by other variables; hence it is the first node of the graph. The code passes a string as an argument to `urlopen` and saves the return value into a new variable named `res`. Using `res`

```
static string ByteToHex(byte[] bytes){
  char[] c = new char[bytes.Length * 2];
  int b;
  for (int i=0; i < bytes.Length; i++){
    b = bytes[i] >> 4;
    c[i * 2] = (char)(55 + b + (((b-10)>>31)&-7));
    b = bytes[i] & 0xF;
    c[i * 2 + 1] = (char)(55 + b + (((b-10)>>31)&-7));
  }
  return new string(c);
}
```

(a)

```
public byte[] hexToBytes(String hStr){
    HexBinaryAdapter adapter = new HexBinaryAdapter();
    byte[] bytes = adapter.unmarshal(hStr);
    return bytes;
}
```

(b)

Figure 7. Reversed code fragments.

```
import urllib2
res = urllib2.urlopen('http://www.example.com')
html = res.read()
```

Figure 8. Code fragment example that has a $String \rightarrow String$ type signature.

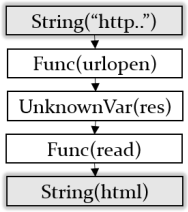

Figure 9. Figure 8's data flow graph.

and the function `read`, it finds the html source, which is also a string. We can see that the first node has $\text{in}_{degree} = 0$ and the last node has $\text{out}_{degree} = 0$, so we can say that the type signature of this code is indeed String $\rightarrow$ String.

Snippet Inputs and Outputs. The inputs and outputs of snippets are not always explicit. If the snippet has a full function declaration and the language is statically typed, the signature is obvious and originates directly from the declaration. But for dynamically typed languages and code fragments that are not part of defined functions, the task of extracting the inputs and outputs is slightly more difficult. For code fragments that have no function declaration, as in the example from Figure 8, we build the data flow graph as explained in Section 6.2.2 and skim for nodes with in_{degree} and out_{degree} equal to zero. These nodes are the snippet's inputs and outputs, respectively. A *snippet's type signature* is the relationship between its inputs and outputs.

Cross-Language Set of Types. Any language has its own set of types. We created a generic type set that includes all the most prevalent types in each language. Each language-specific type is mapped into its generic form, and this way

we can compare signatures from different programming languages. This set of types includes the following: `iterable`, `list`, `set`, `string`, `number`, `dictionary`, `file`, `bool` and `object`. Connections between different types were also created. Consequently, closely related types, such as `iterable` and `list`, will get a higher similarity score than completely different ones.

6. Implementation

We implemented and evaluated our approach using the well-known question-answering site STACKOVERFLOW. One of the main challenges was to obtain labeled data, to which we could compare our results. Towards that end, we implemented a system that allows us to crowd-source this task. The system and experiments were written in Python and Java, and run on an Intel(R) Core(TM) $i7-2720QM$ CPU @ 2.20Ghz machine with 16GB installed RAM, allocating a maximum of 750MB heap space.

6.1 Training

The first component of SIMON is the building of a massive code-to-descriptions mapping. This step is performed only once (or more accurately, once in awhile in order to keep the database up-to-date) and used by all the following steps.

6.1.1 Building a Description Oracle

To build the *semantic description* of a piece of code, we will use the text of a question from SO, whose code is in its answers (or other but still *syntactically similar* code). We created a *description oracle*, that contains more than $1M$ code fragments linked to their descriptions. We used SO to construct our *description oracle*, due to its volume and coverage, but any code can be analyzed. SO includes many *"How to?"* questions answered with a code fragment. Our investigation shows that in most cases the question can play a major role in the semantic description of a piece of code. To build the semantic description of a code fragment, we use the text of the questions that contain the code fragment in one of their answers. The semantic description is composed using the title and the question, with different weights assigned to each. The method for choosing parameter values is described in Section 6.1.2. We concluded that the title should have a major influence on the final score, because it seems to be the most descriptive part of the question. This conclusion was also drawn by Jeon et al. [17] in previous research regarding question similarity. Because SO is crowd-based, it might contain wrong answers, which are expressed by bad code fragments. In order to deal with this challenge, we use the SO voting system, whereby each post (questions and answer) is assigned a score by the site users. We collect statistics about these scores and use them as a quality measure. Table 2 shows the collected information. According to Nasehi et al. [33] and our statistics, we know that only 20% of all answers and 15% of all questions have a score of 3 or higher. Moreover, the average score of a SO post with a score higher than 0 is 3. Therefore, we chose 3 to be the high score threshold for considering questions and answers as informative and we considered only code fragments that originate from posts with high enough scores. In this step we also determine the programming language of the extracted code, using the tags and the text of the post itself, and we train an LSA model, using more than $1M$ textual descriptions exported from SO.

Type	$Avg_{votes>0}$	$\#_{votes>0}$	$\#_{all}$	$\#_{votes>2}$
Question	3	$> 3.7M$	$> 7.9M$	$> 1.1M$
Answer	3	$> 8.1M$	$> 13.5M$	$> 2.8M$

Table 2. Statistics about STACKOVERFLOW posts.

6.1.2 Parameter Tuning

Our system contains several parameters, such as the weight assigned to each question part, the influence of the type analysis on the final similarity score, and the threshold, which separates *similar* from *different* tags. Different parameters produce different classifiers, and we want to select the best one. We built a dataset with many pairs of code fragments and a boolean, which represents whether the fragments are similar. We use our labeled data, and look for the best values using *ten-fold cross-validation*, a common choice for model selection problems. Ten-fold cross-validation means that the dataset is randomly split into ten equal size subsets. With ten validation steps, trained on nine out of ten subsets and tested on the remaining one, we can find the best values [22]. We found the parameters that achieve the highest measurement values (as discussed in Section 7.2), while maintaining relatively low deviation, and set them to be our system parameter values. Our analysis showed that 0.4 of the score should come from the titles and 0.6 from the entire question text. The type signature gets 0.15 of the final similarity score, and the threshold was determined to be 0.52.

6.2 Measuring Similarity

6.2.1 Textual Similarity

The text similarity method is one of the building blocks of our work. Hence, we had to choose it wisely. We decided to use a combination of certain text similarity techniques.

For the Titles. Titles are mostly short and descriptive (e.g., the titles in Table 1). Hence, we decided to use query expansion [49], expanding only terms from the programming domain to avoid reduction in the quality of the results. Our specialized technique is based on the database created by Tian et al. [47]. With the expanded titles, we used ADW [38] to build the corresponding vectors and compute the relevant cosine similarity between them. This part was written in Java using the open source code of ADW and SEWordSim.

For the Content. The combination of the title and question is a longer text (mostly around five sentences). We used TF.IDF

and SVD (as implemented in Scikit Learn and explained in Section 3.1.2) to implement LSA. We trained the model once and created a semantic space, on top of $1M$ texts that were exported from SO.

6.2.2 Signature Similarity

To infer the type signature of a given snippet, we visit its AST and extract type and flow information. Our snippets are partial programs; they are often not parsable, and hence we can't create the AST immediately. In the first step, the snippet is processed to the code which is most likely to parse; in Java, for example, we initially import required classes, and then add variable initialization, fake method and class declarations, using ASTVisitor. After we get parsable code, we generate its AST and then we visit all its nodes. For each node we decided separately which information we want to keep. In Python, which is a dynamically typed language, we also extracted its function input and output types, using the documentations. The final product of this step is a labeled graph, from which we can find the snippet's inputs and outputs. To compute the similarity of type signatures for two given code fragments, we first try to match each type in one signature to its best fit in the other signature. Perfect matches (the same cross-language type) were assigned a higher score. Partial matches (e.g., `set` and `list`), were assigned a lower score. The final value is in the range of $[0, 1]$. We implemented our analysis for Java and Python.

6.3 Labeling System

To build the labeled corpus we had to find a way to determine the similarity level between a vast group of code fragments. This task requires human input and cannot be performed automatically. The idea to use crowdsourcing to support software engineering tasks has recently elicited great interest [7, 46], so we decided to exploit it; we developed a crowdsource based web application called LIKE2DROPS. The system is available online here: `http://like2drops.com`. In this system, the user's task is to choose the tag that most appropriately describes the pair's similarity on a 5-level Likert scale [27]. The possible tags are: *very similar, pretty similar, related but not similar, pretty different* and *totally different*. We designed a *human intelligence task* (HIT) of classification (100 pairs each) and looked for qualified programmers, with broad programming knowledge. To get many experts' opinions, we posted our system as a job in many freelancer sites, such as Elance, oDesk and Freelancer, and collected users' answers. Because we couldn't blindly trust any user, we had to integrate a *trust test*. Each fifth pair was taken from a sample set that we manually tagged. Users with high percentages of disagreement were removed from our experiments, due to our suspicion that they might have lied about their expertise. In our labeling system, we required that pairs be assigned a quantitative score from 1 to 5; however, the final evaluated product is a binary classifier. We saw that the overall direction of different users was often

```java
int x = Integer.parseInt("8");
```

(a)

```java
char c = '1';
int i = c - '0';
// i is now equal to 1, not '1'
```

(b)

Figure 10. Two code fragments with inconclusive similarity level, both written in Java.

```python
def f7(seq):
    seen = set()
    seen_add = seen.add
    return [ x for x in seq if x not in
            seen and not seen_add(x)]
```

(a)

```
List<Type> liIDs =
     liIDs.Distinct().ToList<Type>();
```

(b)

Figure 11. Semantically related fragments for creating a distinct list that was found with SIMON

the same, e.g., *very similar and pretty similar*, but the specific tags were not. We also saw that a large portion of the *related* labels were accompanied by *similar* labels; hence we decided to treat *related* as *similar* for our binary evaluation (note that SIMON's output is quantitative).

7. Evaluation

7.1 Labeled Corpus of Program Pairs

For our evaluation we created a large corpus of program pairs, tagged by similarity level. This corpus is of possible interest of itself. It contains 6,500 labeled pairs, based on more than 15,000 user tags, and is continuously growing.

Because *similarity is not equivalence*, it is not always clear whether two code fragments really are similar. Therefore, we had to build this corpus carefully. The corpus base is the crowd's knowledge, and each pair's similarity was determined by agreement among several users. Cronbach's alpha [43] value for program pairs with exactly 3 raters is 0.847 and for 4 raters is 0.81. These values indicate hat the tags assigned by different raters are consistent. Figure 10 shows an example of a pair whose similarity is inconclusive and might be tagged as similar or different. The figure shows two code fragments that in principle transform a string representation of a number to its integer form. However, the first can handle any integer while the second is limited to digits only. Thus different users might assign this pair different tags.

7.2 Measurements

Using tags fetched from LIKE2DROPS, we computed the precision, recall, accuracy and AUC. We denote *Precision* as the fraction of snippet pairs that are labeled as similar and are indeed similar. *Recall* is denoted by the fraction

#	Configuration	Precision	Recall	Accuracy	AUC
1	Full approach	**86.2%**	**85%**	**87.3%**	**0.9391**
2	Full, without type signatures	85.8%	85%	87.1%	0.9386
3	Full, without software specialized vocabulary	85.1%	83.7%	86.3%	0.9358
4	Full, without ADW and software specialized vocabulary	83.3%	83.1%	85.1%	0.9335
5	Full, without LSA	83.6%	82.4%	85.1%	0.9343
6	Only titles (ADW and software specialized vocabulary)	83.8%	73.7%	82%	0.9011
7	Only content (LSA)	81.1%	73.9%	80.8%	0.8726
8	Tokenized-code, TF.IDF [11]	77.3%	76.5%	79.6%	0.8342
9	Tokenized-code, LSA	79.8%	69.6%	78.7%	0.825
10	Random	45.6%	47.4%	50.7%	0.4993

Table 3. Results obtained by omitting different parts of SIMON and comparison to other techniques.

of similar code fragment pairs that are actually labeled as similar. *Accuracy* is the fraction of correct labels.

When building a classifier the challenge is to find a good threshold (used to separate similar from different samples) across experiments. Plotting the accuracies using different thresholds yields the *receiver operating characteristic* (ROC) curve. The accuracy of the classifier is the *area under the curve* (AUC) value [6]. The ROC curve allows us to compare different classifiers.

7.3 Similarity Classifier Evaluation

Dataset Setup. The experimental dataset contains 6,500 pairs of code fragments that were extracted from SO. This dataset consists of code fragments written in 37 different programming languages, such that Python, Java, C#, php and Javascript are the dominant languages. The average snippet length is 165 characters, while the smallest snippet has 35 characters and the longest one has 600. Figure 11 is an example of a similar pair from the database. We collected more than 15,000 user classifications. For 94% of the labeled pairs there was a consensus regarding similarity (similar or not); however, in the remaining 6% we couldn't determine the correct label because the answers varied greatly. It thus appears that no conclusive decision is possible. We therefore omitted these pairs from our experiment.

What is the best configuration for establishing similarity? For pairs with more than one user label (sample set of 2749 pairs), the results show that 87.3% of our labels are consistent with those of the users, while the precision is 86.2% and the recall is 85%. Table 3 shows the results using different configurations of SIMON; note that we chose the best threshold for each configuration, and that these values are the candidates that maximize the accuracy while maintaining relatively high recall and precision. The first row is the full configuration, that is, SIMON with all its steps. The next rows show the numbers achieved using a partial configuration of SIMON. *Row 2* is the full configuration, using the text analysis only, without type signatures. *Row 3* is the full configuration, using the text analysis, without software-specific specialized vocabulary. *Row 4* is the full configuration, using LSA alone as the text similarity method. *Row 5* is the full configuration, using TF.IDF as the text similarity method. *Row 6* is the full configuration, using the title only for the textual description (ADW as the text similarity method). *Row 7* is the full configuration, using the content only for the textual description (LSA as the text similarity method). *Row 8* shows the tokenized-code approach, using TF.IDF for the text analysis. This result was achieved using the approach of the DeSoCoRe tool [11]. *Row 9* shows the tokenized-code approach, using LSA for the text analysis. *Row 10* shows a random choice for each pair (uniform). Tokenized-code refers to the naive approach, where we use the code as its textual description. The results of lines 8 and 9 were achieved with pre-processed code (token decomposition and operator removal), and the results without this step are less accurate (around 10% less). The relatively high values achieved using this approach can be explained by wrong user labels that were based on the syntax instead of the semantics. These wrong labels acted in favor of the naive approach. It can be seen that the difference between the first two lines is not significant: the contribution of the type signatures is almost unobservable due to a bias caused by relatively few pairs having signatures (the implementation does not cover all the programming languages). We believe that the type signature could be more useful than what we have demonstrated in our experiments in datasets with more pairs that have similar description and different type signatures. McNemar's test [29] between the results obtained with our approach and each of the other configurations (excluding the second) showed two-tailed P-values of less than 0.0001, which by conventional criteria indicate significant difference.

Method	Precision	Recall	Accuracy
SIMON	92.2%	92.6%	89%
DECKARD [18]	71.5%	51.5%	55.6%
JPlag [41]	65.3%	45.1%	48.5%

Table 4. Comparison results on Java-Java pairs dataset.

Comparison to other code similarity techniques. It is important to note that when the given programs are very close syntactically, syntactic clone detection might be more pre-

#Users	#Labels	Precision	Recall	Accuracy	AUC
All	3951	80.4%	81.7%	83.1%	0.9036
> 1	2749	86.2%	85%	87.3%	0.9391
> 2	1430	89.5%	86%	89.7%	0.9591
> 3	710	91.7%	88.5%	91.4%	0.9720
> 4	419	92.6%	88.5%	91.9%	0.977
> 5	282	91.5%	87%	90.8%	0.9727
> 6	200	**94**%	91.9%	94%	0.9834
> 7	147	93.4%	**93.3**%	**94.6**%	**0.9889**

Table 5. Results obtained by different confidence levels.

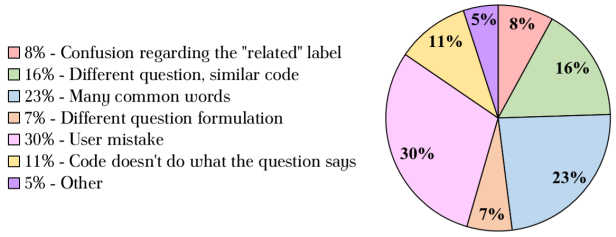

Figure 12. Analysis of more than 200 wrong classifications.

cise than our approach. The goal of our approach is to handle the case in which the given code snippets are not similar syntactically, and applying techniques that rely on syntax is unlikely to succeed. This distinguishes our technique from classic clone detection techniques. Figure 1 is an excellent example of such case. To evaluate our approach, against a pure structural approach, we compared it to JPlag [41], a tool that finds similarities among multiple sets of Java source code files in order to detect code plagiarism. Towards the comparison, we filtered only Java-Java pairs, made them parsable (as described in Section 6.2.2), and used them for this evaluation. The results are shown in Table 4 and based on 354 Java-Java pairs. JPlag achieved "high" precision due to the large number of pairs that were classified with a score of 0. We used the same Java-Java dataset with Deckard [18], using the following parameters that achieved optimized results: 10 minimum tokens, 0 stride and 0.45 similarity threshold. These results are also shown in Table 4. Vinayakarao et al. [48] present a technique for finding functionally similar snippets, using textual similarity and program analysis techniques. The approach is demonstrated with STACKOVERFLOW posts, focused on Java snippets. The similarity between two snippets is determined by applying a simple text similarity method to the corresponding posts' vocabulary, and by the snippet's structural information. Our work differs from this work in the following ways: Vinayakarao et al.'s goal is to find code fragments that match a given query, while ours is to capture similarities between code fragments. Moreover, the second step of their approach is to filter out syntactically similar fragments by their complexity level, while we want to preserve fragment similarity. We tried to adapt Vinayakarao et al.'s approach to our case; however, many implementation details are missing and their approach being query-post oriented, led to bad performance, so we decided to omit this evaluation.

Does agreement between more raters improve the results? Despite our efforts to get many answers for each pair (e.g., many user opinions), there was still a group of pairs with only a few tags. To check the source of the mismatches between the labeled data and SIMON's classifications, we manually analyzed a representative group ($|group| > 200$) of misclassified tags. Figure 12 shows the error distribution. Many errors were caused by wrong labels, given by one user. We tried to understand whether agreement between more users can help us to achieve better results. Table 5 shows the different measurements achieved using pairs with an increasing agreement threshold. The threshold is based on the number of distinct users that agreed on a specific pair. Notably, even an increase to agreement between two users dramatically improves the results. The pie chart also indicates the possibility of improving the results by adopting better text similarity techniques and shows that we can't blindly trust the descriptions.

7.4 Code Retrieval Evaluation

To find representative fragments, as explained in Section 5.3, we have to query our database using syntactic similarity technique. We performed several experiments to evaluate our ability to query the *documentation oracle*.

Generic method. The first experiment examines our ability to retrieve syntactically similar snippets with the generic method. We checked two sets of code fragments from multiple programming languages: (i) control group: 25 fragments randomly chosen from our 1M database, and (ii) 25 fragments randomly chosen from our 1M database that were mutated by a programmer. He was instructed to apply mutations that consist of changes in the formatting, naming and literals – changes that preserve the flow and functionality of the original program.

This experiment consists of two sub-experiments:

Full database (our original 1M snippets database) - We used each fragment as an input query and evaluated our ability to retrieve the correct fragment. Fragments from the first group were correctly returned as the first match in all cases. Fragments from the second group were retrieved as the first result in $16/25$ of the cases, as the second result in $8/25$, and not at all only once.

Limited database - We created a test set with 1,000 code fragments that were randomly extracted from SO. A programmer with more than 7 years of experience tagged each of the 50,000 pairwise options of query and database entry. This process was long and took around 4 months using a dedicated system that allowed him to easily tag many pairs. With these labels we managed to compute the precision, recall and accuracy, as shown in Table 6.

Precision	Recall	Accuracy
94%	92.1%	99.987%

Table 6. Our code retrieval results based on a test set with 1,000 code fragments and 50 search queries.

```
>>> wordlist = ['Schreiben\nEs', 'Schreiben',
        'Schreiben\nEventuell', 'Schreiben\nHaruki']
>>> [ i.split("\n")[0] for i in wordlist ]
['Schreiben', 'Schreiben', 'Schreiben', 'Schreiben']
```

(a)

```
>>> la = ["a.b.c.d", "a.b.c", "y.d.k", "z"]
>>>
>>> [elem.split('.')[-1] for elem in la]
['d', 'c', 'k', 'z']
```

(b)

Figure 13. Syntactically similar code fragment.

Python-specific method. To evaluate the Python-specific syntactic similarity method, we designed a simple and precision-oriented experiment. We chose this method of evaluation due to the success of the previous technique and the fact that this method is recall-preservative and may negatively affect only the precision (each pair that was found similar using the first method will necessarily be similar in this method). We initialized the pair database with more than 1,000 pairs. Then, we collected pairs that were labeled as similar by SIMON (Figure 13 shows an example of such a pair). Another programmer, with more than 8 years of experience, validated the results and determined whether each pair indeed represents similar snippets. For identical fragments with no modifications except to their variable names, perfect results are obtained. The results show almost 94% agreement along all the checked pairs. Analysis of the errors revealed that many of them are the product of very short programs, important function parameters (that change the code's functionality) or use of functions with the same name, originating from different libraries. We could have tightened the requirements for a pair to be declared as syntactically similar; however, had we chosen to do so, we would have lost good matches. Our current preference is to use this approach, but it is open for future research.

7.5 Performance

SIMON's performance was not a priority, and therefore no major optimization steps were implemented. However, we aimed to achieve reasonable computation times. Nonetheless, the first phase, wherein we build the documentation oracle and train the text similarity models, is long: around 8 hours. This step is done only once in awhile to keep the database updated. The average running time for the text similarity (LSA and ADW) and type signature comparison steps is around 1 second each, while the code retrieval step takes around 0.5 seconds in the generic retrieval method, and 1 second for the Python-specific method.

8. Discussion

In this section we present justifications for some of the choices we made, interesting points, and certain limitations.

Limitations. The significant challenge of our approach is to find representative code in our database for an arbitrary code fragment provided by a user. While it seems a major challenge, we believe that as a more code is integrated into our database, the greater the chance that any reasonable fragment will have a proper representative. Even a closed code base could be annotated with descriptions from SO. This challenge and its solution are discussed in greater detail in Section 4.1. Another challenge in our approach is that it is heavily crowd-based. When a given program does not have sufficiently detailed or precise corresponding textual descriptions, the textual similarity may be too weak. In such cases, our approach may fail to establish similarity. We adopted elimination techniques, such as ignoring code fragments that originate from posts with a low voting score and removing bad user classifications; however, our approach might still be biased. Another limitation is associated with *relatedness*. While we believed that this term is well defined and clear, a review of classified pairs showed that it is controversial. Consequently, we got many pairs classified as *related*, some of which indeed follow our definition, but some of which do not. This reduced SIMON's performance and we think that the term *relatedness between code fragments* can be a foundation for future work. One should also note that our approach is limited by the performance of its syntactic similarity and text similarity techniques. With that said, our evaluation shows that the chosen methods are a good match.

ADW & LSA. Simple text similarity techniques such as TF.IDF are sufficient and can get good results in many cases; however, they cannot capture semantic relationships due to the randomness of human expression. Consider, for example, two code fragments that *sort a list*. One is associated with the textual description: *Arrange group of numbers in Python* and the second with *Sort my digits list - Java*. It is easy to see the affinity between the descriptions; however, without the addition of advanced techniques such as ADW or LSA, they might be classified as different.

Specialized Vocabulary. Software contains unique words that are not part of "everyday speech", such as types (e.g., *String, Int, Integer*) and keywords. It also contains words whose meanings are different from their everyday sense, such as *directory*. The meaning of the word "directory" (according to Merriam-Webster) is "serving to direct; specifically: providing advisory but not compulsory guidance" or "a book that contains an alphabetical list of names of people, businesses, etc." But in the programming domain *directory* is a folder, where we can store our files. The words *folder* and *directory* are strongly related when dealing with software, a fact we would miss if our method were unable to recognized the unique meaning of these words in the software context.

However, we did not observe that this distinction had any significant effect on the results in general. This is probably due to the use of mostly consistent terms in the descriptions.

Text Similarity vs. Code Similarity. Semantic text similarity is a hard problem due to ontologies and negation. Much research has been conducted on this subject [16, 38]. Common techniques are based on knowledge bases, such as WordNet, which help to clarify the randomness created by ontologies and achieve good results. On the other hand, the nature of code complicates the use of standard text similarity techniques. Code fragments have different syntax, according to their programming language, different documentation, formatting, identifiers, data types, APIs, libraries, algorithms, literals, etc. Similar code fragments can be much more complex than texts and differ in many ways. In contrast, the text similarity problem we deal with is even easier in that we have to compare texts that originate from the narrow domain of programming, all written in English. These restrictions make our specific text similarity problem solvable and, combined with snippet analysis, which eliminates the errors of the text similarity method – it yields great results.

9. Related Work

In this section, we briefly survey a few related works.

Cross-Language Code Similarity. The topic of cross-language code similarity has not been widely investigated. However, some research exists. Flores et al. [11] approached this problem to detect source code re-use across programming languages, using NLP techniques. After splitting source code to functions, they compare the similarity of functions from the source codes, using basic NLP techniques, such as n-grams and tf. To measure the distance between two code representations, they used cosine similarity. There has also been some work that relies on probabilistic models of programs based on large code bases [19]. In this work, Karaivanov et al. suggested an approach for program translation based on mapping between C# and Java grammars.

Clone Detection. In general, syntactic clone matching technique works on the structure of the program. For example, Deckard [18] turns ASTs into a vector and uses locality sensitive hashing to find similar vectors. Deckard uses features that are directly extracted from the program's AST and does not rely on mapping them to more abstract concepts like natural language description as done in our work. Tools like Deckard can be used for initial mapping of a given program to corresponding representatives in the database.

NL as a Tool for Programming Tasks. Hindle et al. [14] were the first to apply statistical language models originating from the NLP world to programming languages. They showed that code is more repetitive and predictable than natural language and hence can be modeled using statistical language models. Furthermore, they suggest that code fragment retrieval can rely on English descriptions, a task we deal with. Kuhn, Ducasse and Gírba [25] leverage the semantic knowledge expressed in code naming and comments to cluster code sections. They use LSI and other IR techniques to find topics, utilizing the vocabulary in the code. Applying such techniques directly on the code may be the first step in building the textual descriptions for "new" code fragments. There has been a lot of work on the use of natural language in software engineering tasks. Some work has addressed the problem of inferring specifications from natural language descriptions [35, 57]. Other works have dealt with security policy extraction [51], detection of API documentation errors [56], bug identification [13], code convention modification [1], traceability link recovery [34] and source code search [21]. To the best of our knowledge, we are the first to use the mapping from code to textual description captured in big code to establish cross-language code similarity.

Automatic Tagging. Our approach is also inspired by automatic tagging of images used in the search context [8, 24]. In such methods images are annotated with textual tags that are used for retrieval. Existing work on synthesis of high-level descriptions from code fragments [45] typically relies on structural analysis of the code itself. While this has many advantages, this approach is often limited to describe the code only in terms of its actual operations (in terms of the solution), without any mention of the problem being solved.

Code Retrieval. Mishne and De Rijke [31] used conceptual modeling of code to perform the task of code retrieval, avoiding the differences in programming language syntax. They represent each code as a *conceptual graph* (CG) [44] and thus were able to capture the content and the structure of the code. Ponzelli et al. [40] developed an Eclipse plug-in which connects code snippets to relevant STACKOVERFLOW discussions. They built a ranking model to evaluate the relevance of the discussion to the code. Keivanloo et al. [20] suggested approach for spotting working code examples via free-form queries used for Internet-scale code engines. Their approach is based on p-strings, cosine similarity and ranking models.

10. Conclusion

We presented a novel approach for measuring semantic relatedness between code fragments based on their corresponding natural language descriptions and their type signatures.

Our approach is inspired by work on image retrieval, where image similarity is determined by first associating images with textual descriptions (tags), followed by comparison of these descriptors. We believe that this presents a new direction to be explored for program similarity, where the approach presented in this paper is a first modest step.

We implemented our approach in a tool called SIMON, using STACKOVERFLOW, and applied it to determine relatedness between program pairs. We used the crowd to collect labeled data, which may itself be of interest. We combined an open-world approach, text similarity techniques, and type analysis, and showed that it leads to promising results.

Acknowledgments

The authors would like to thank Ran Zilberstein for commenting on an earlier version of this paper. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME.

References

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *FSE*. ACM, 2014.

[2] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technology (TOIT)*, 2001.

[3] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 1993.

[4] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*. IEEE, 1998.

[5] H. Berghel and D. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 1984.

[6] A. P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 1997.

[7] C. Chen and K. Zhang. Who asked what: integrating crowdsourced FAQs into API documentation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[8] L. Chen, D. Xu, I. W. Tsang, and J. Luo. Tag-based web photo retrieval improved by batch mode re-tagging. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.

[9] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *JAsIs*, 1990.

[10] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *IEEE International Conference on Software Maintenance (ICSM) Proceedings*, 1999.

[11] E. Flores, A. Barrón-Cedeno, P. Rosso, and L. Moreno. DeSoCoRe: Detecting source code re-use across programming languages. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Demonstration Session*, 2012.

[12] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[13] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[15] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[16] A. Islam and D. Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2008.

[17] J. Jeon, W. B. Croft, and J. H. Lee. Finding similar questions in large question and answer archives. In *Proceedings of the 14th ACM International conference on Information and knowledge management*, 2005.

[18] L. Jiang. *Scalable Detection of Similar Code: Techniques and Applications*. PhD thesis, University of California, Davis, 2009.

[19] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014.

[20] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[21] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[22] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, 1995.

[23] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *13th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2006.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[25] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 2007.

[26] T. K. Landauer and S. T. Dumais. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 1997.

[27] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[28] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to Information Retrieval*. Cambridge university press, 2008.

[29] Q. McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 1947.

[30] G. A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 1995.

[31] G. Mishne and M. De Rijke. Source code retrieval using conceptual similarity. In *RIAO*, 2004.

[32] M. Monperrus and A. Maia. Debugging with the crowd: A debug recommendation system based on StackOverflow. 2014.

[33] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q&A in

StackOverflow. In *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[34] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *International Conference on Program Comprehension (IPCP)*, 2010.

[35] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[36] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[37] T. Pedersen, S. Patwardhan, and J. Michelizzi. WordNet:: Similarity - measuring the relatedness of concepts. In *Demonstration papers at hlt-naacl*. Association for Computational Linguistics, 2004.

[38] M. T. Pilehvar, D. Jurgens, and R. Navigli. Align, disambiguate and walk: A unified approach for measuring semantic similarity. In *ACL*, 2013.

[39] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. 1998.

[40] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the IDE into a self-confident programming prompter. In *Proc. of the Working Conference on Mining Software Repositories*, 2014.

[41] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *J. UCS*, 2002.

[42] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 1975.

[43] J. R. A. Santos. Cronbach's alpha: A tool for assessing the reliability of scales. *Journal of extension*, 1999.

[44] J. F. Sowa. Conceptual structures: information processing in mind and machine. 1983.

[45] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2011.

[46] K. T. Stolee and S. Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010.

[47] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Conference on Software Maintenance, Reengineering and Reverse Engineering, Software Evolution Week-IEEE*, 2014.

[48] V. Vinayakarao, R. Purandare, and A. V. Nori. Structurally heterogeneous source code examples from unstructured knowledge sources. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*. ACM, 2015.

[49] E. M. Voorhees. Query expansion using lexical-semantic relations. In *SIGIR*. Springer, 1994.

[50] C. Wang, J. P. Reese, H. Zhang, J. Tao, Y. Gu, J. Ma, and R. J. Nemiroff. Similarity-based visualization of large image collections. *Information Visualization*, 2013.

[51] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[52] T. Yeh, K. Tollmar, and T. Darrell. Searching the web with mobile images for location recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.

[53] W.-t. Yih. Learning term-weighting functions for similarity measures. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: Volume 2*. Association for Computational Linguistics, 2009.

[54] A. T. Ying. Mining challenge: Comparing and combining different information sources on the stack overflow data set. In *Working Conference on Mining Software Repositories*, 2015.

[55] W. Zhang, T. Yoshida, and X. Tang. A comparative study of TF-IDF, LSI and multi-words for text classification. *Expert Systems with Applications*, 2011.

[56] H. Zhong and Z. Su. Detecting API documentation errors. In *ACM SIGPLAN Notices*, 2013.

[57] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2009.