Code Similarity via Natural Language Descriptions

Meital Zilberstein (Ben Sinai)
Code Similarity via Natural Language Descriptions

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Meital Zilberstein (Ben Sinai)

Submitted to the Senate of
the Technion — Israel Institute of Technology
Nisan 5775 Haifa April 2015
The research thesis was done under the supervision of Prof. Eran Yahav in the Computer Science Department.

I would like to thank my advisor, Prof. Eran Yahav, for his devoted guidance and encouragement. For ideas, discussions and the knowledge I gained throughout the work with you. Without you this work would not be possible.

I would also like to thank Shir Yadid for being there for me, day after day, as my research partner and most amazing friend.

And finally I would like to thank my family and especially my dear husband, Ran, for their continual support, love and motivation throughout the entire process. You mean the world to me!

The generous financial support of the Technion is gratefully acknowledged. The research leading to these results has received funding from the European Union’s - Seventh Framework Programme (FP7) under grant agreement n°615688 - ERC - COG - PRIME.
Contents

List of Figures iv
List of Tables v
List of Algorithms vi
Abstract 1
Abbreviations and Notations 2

1 Introduction 3

2 Background 6
  2.1 Text Similarity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  2.1.1 Tf.Idf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.1.2 Latent Semantic Analysis . . . . . . . . . . . . . . . . . . . . . . 8
  2.1.3 Align, Disambiguate, and Walk (ADW) . . . . . . . . . . . . . . 8
  2.2 Cosine Similarity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3 Overview 10
  3.1 Motivating Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  3.2 Intuition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  3.3 Key Aspects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

4 Leveraging Collective Knowledge 15
  4.1 Open World Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  4.1.1 Trouble in Paradise . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  4.1.2 The Importance of Data . . . . . . . . . . . . . . . . . . . . . . . . . 16
  4.2 Training Phase . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
8.3 NLP Technique Enriching .......................... 39

9 Related 40
  9.1 Cross-Language Code Similarity .................. 40
  9.2 NL as a Tool for Programming Tasks ............ 40
  9.3 Code Retrieval .................................. 41
  9.4 Question Similarity .............................. 42
  9.5 Similarity Learning ............................... 42

10 Discussion 43
  10.1 Why ADW & LSA? ................................. 43
  10.2 Why Software Specialization? .................. 43
  10.3 Why Type Signatures? ............................ 44
  10.4 Text Similarity vs. Code Similarity .......... 44
  10.5 Limitations .................................... 45

11 Conclusion 46

Bibliography 47

Hebrew Abstract
List of Figures

2.1 Semantically similar sentences ........................................ 6
2.2 Naive example for tf.idf use ........................................... 7

3.1 Semantically related fragments that generate all possible permutations. (a) is written in Python and (b) in Java .......... 11
3.2 simon’s core; mapping snippets to natural language elements 13
3.3 The architecture of simon .............................................. 14

4.1 Similar code fragments that implement distinct representation of a given list ..................................................... 16
4.2 The percentage of code fragments that have syntactically close code in the database, as a function of the database size 17
4.3 Two code fragments with inconclusive similarity level, (a) written in Python, and (b) in Java ................................. 18

5.1 Two syntactically similar code fragments .............................. 20
5.2 Text processing example using a sentence from Figure 2.1 ...... 21
5.3 Code fragment that has a String → String type signature ... 23
5.4 Figure 5.3’s data flow graph ........................................... 23

6.1 Two code fragments and their user classification distribution .. 30

7.1 Analysis of more than 200 incorrect classifications ............. 32
7.2 Syntactically similar code fragment, found using simon ..... 36

10.1 Two code fragments which are the reverse of each other ... 45
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Information about the code fragments in Figure 3.1</td>
<td>12</td>
</tr>
<tr>
<td>6.1</td>
<td>Statistics about Stackoverflow posts</td>
<td>26</td>
</tr>
<tr>
<td>7.1</td>
<td>Results obtained by omitting pairs with low user confidence</td>
<td>33</td>
</tr>
<tr>
<td>7.2</td>
<td>Results obtained by omitting different parts of simon and comparison to other techniques</td>
<td>33</td>
</tr>
<tr>
<td>7.3</td>
<td>Our code retrieval results based on a test set with 1000 code fragments and 55 search queries</td>
<td>35</td>
</tr>
<tr>
<td>8.1</td>
<td>Code fragments and their extracted tags as established using our mapping and native keyword extraction method</td>
<td>38</td>
</tr>
</tbody>
</table>
List of Algorithms

1 Finding the similarity score of two given snippets . . . . . . 25
Abstract

Code similarity is a central challenge in many programming related applications, such as code search, automatic translation, and programming education. In this work, we present a novel approach for establishing the similarity of code fragments by computing textual similarity between their corresponding textual descriptions. In order to find textual descriptions of code fragment, we leverage collective knowledge captured in question-answering sites, blog posts and other sources. Because our notion of code similarity is based on similarity of corresponding 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 support the text-based similarity function, we also apply static analysis on the code fragments themselves and use it as another measure for similarity.

To experiment with our approach, we implemented it using data obtained from the popular question-answering site, STACKOVERFLOW, and used it to determine the similarity of 100,000 pairs of code fragments which are written in multiple programming languages. We developed a crowdsourcing system, LIKE2DROPS, that allows users to label the similarity of code fragments. We utilized these classifications to build a massive corpus of 6,500 labeled program pairs. Our results show that our technique is effective in determining similarity and relatedness, and presents more than 80% precision, recall and accuracy.
## Abbreviations and Notations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADW</td>
<td>Align, Disambiguate, and Walk</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>IR</td>
<td>Information Retrieval</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>LSA</td>
<td>Latent Semantic Analysis</td>
</tr>
<tr>
<td>SOF</td>
<td>Stackoverflow</td>
</tr>
<tr>
<td>tf.idf</td>
<td>Term Frequency Inverse Document Frequency</td>
</tr>
<tr>
<td>VSM</td>
<td>Vector Space Model</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

We address the problem of similarity between code fragments. Code similarity and equivalence are classic problems in programming languages and software engineering [3, 29, 6]. There has been a lot of work on syntactic code similarity. However, most of these approaches cannot capture similarity across programs using different APIs or different algorithms, let alone different programming languages.

There has also been a lot of work on semantic equivalence and differencing of programs [12, 25, 26]. However, most of these approaches require that the programs being compared are not too different from one another (e.g., different versions of the same program with small patches [26], or the same program at different abstraction levels [29]).

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 from various programming languages.

Our Approach: Combine Big Code and Natural Language Processing Inspired by Hindle et al. [11], 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. Our work employs techniques from natural language processing to obtain insights regarding software. The challenge is to establish such semantic similarity (more generally, semantic relatedness) automatically. The main idea of this
work 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), and (ii) measuring semantic similarity between textual descriptions.

To address the first challenge, we can rely on relationships between code and its description as created, for example, in common question-answering sites (such as STACKOVERFLOW (SOF)), in documentation, on blog posts, and in method and variable names.

To address the second challenge we utilize state-of-the-art text similarity methods as explained in Section 2.1. In addition, we use lightweight dataflow analysis to extract basic type information, and improve the precision of relatedness results.

Our approach represents a radical departure from standard techniques that only use the code, and takes an “open world” approach that leverages collective knowledge.

**Research Question** Is it possible to use the data, achieved from common sites and created by the crowd to capture the similarity between two given code fragments?

**Main Contributions** The main contributions of this research are:

- We present a framework for semantic relatedness of code, based on similarity of corresponding natural language descriptions.

- Our main insights are: (i) We can use Big Code to capture relationships between code fragments and related textual descriptions; (ii) We can use NLP algorithms, adapted to text describing programming artifacts, to establish quantitative similarity between code fragments.

- We have implemented our technique in a tool called simon, and applied it to determine relatedness between 100,000 pairs of code fragments. Our approach works for large number of code fragment structures, many of which cannot be handled using any other technique.

- To evaluate our approach, we developed a crowdsourcing platform in which users can classify the relatedness of pairs, and collected results from 40 users to classify a sample of 6,500 pairs. The labeled data is of independent interest for researchers in this area.
We show that combining NLP-based textual similarity, together with simple code analysis, can effectively and efficiently determine relatedness between code fragments. Our approach has more than 80% precision, recall and accuracy. These values make our approach a valuable complement to techniques based purely on semantic similarity of code.
Chapter 2

Background

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

2.1 Text Similarity

Determining the similarity between two texts is one of the main problems in many NLP and Information Retrieval (IR) applications [41]. The Vector Space Model (VSM)[32] 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 (sub-group of texts that is used for the model training). Each cell in the vector is associated with a weight that indicates its importance and can be calculated in certain ways.

Example We use the semantically similar sentences from Figure 2.1 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.

Figure 2.1: Semantically similar sentences
2.1.1 Tf.Idf

The Term Frequency Inverse Document Frequency (tf.idf) is the 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. In this work, we use the natural tf, and the logarithmic and smooth idf, as shown in (2.1). The smoothing is used to prevent zero division and negative values:

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

(2.1)

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 $t_{dt} = \text{tf}_d \cdot \text{idf}_t$ [43]. One drawback of this approach is the lack of semantic knowledge.

Regarding the example sentences from Figure 2.1, 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 low similarity score between the sentences.

Figure 2.2 shows a naive example for tf.idf use. The left side is the training phase wherein the idf matrix is built. The right side is the input document. The combination between them is the output vector.
2.1.2 Latent Semantic Analysis

*Latent Semantic Analysis* (LSA), also called *Latent Semantic Indexing* (LSI), is another VSM based method that utilizes the latent semantic structure in 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. Consequentially, it can attribute similarly to different words with similar meaning. LSA is based on a matrix of terms-documents and the mathematical model of *Singular Value Decomposition* (SVD). Deerwester et al. [5] elaborate on the technical details of SVD. Intuitively, any rectangular matrix, $M$, is decomposed to the product of three other matrices, such that $M = X_0S_0Y_0$ 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 terms-documents matrix, the matrices can be simplified by deleting 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. These matrices are used to create the semantic space, wherein closely associated elements are placed near each other. It 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 two example sentences (Figure 2.1) depends on the selected training set, but here we can spot the connection between the words “order” and “sort” (which is the base form of “sorted”) and we get higher similarity score.

2.1.3 Align, Disambiguate, and Walk (ADW)

A contemporary, unified semantic representation called ADW was presented by Pilehvar et al. [28]. It leverages a common probabilistic representation over the senses of a word. ADW is based on WordNet 3.0 [20] 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 (Figure 2.1), we first remove stop words 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 getting a relatively high similarity score.

2.2 Cosine Similarity

The Cosine Similarity function [19] is a widely used function to compute the similarity between two given term vectors, computed by one of the aforementioned techniques. It is calculated by taking the inner product \((v_1 \cdot v_2)\) of the vectors and dividing it by the product of their vector lengths. Because the vectors are normalized to unit length, the angle is the only influence on the vector’s similarity score and it prevents bias originating from different text sizes. (2.2) shows the equation for computing the similarity between the vectors \(v_1\) and \(v_2\):

\[
\text{cosinse}(v_1, v_2) = \frac{(v_1 \times v_2)}{\|v_1\| \|v_2\|},
\]

(2.2)

where \(\|v\|\) is the Euclidean norm of the vector \(v\).
Chapter 3

Overview

In this chapter, we provide an informal, high-level overview of simon using an example.

3.1 Motivating Example

Consider the two code fragments in Figure 3.1. Both of these fragments generate permutations of a given string. The code fragment in Figure 3.1(a) is written in Python and the one in Figure 3.1(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 languages’ syntax and their use of two different computation structures. Even semantic approaches that are based on input-output relations will have difficulty finding the connection between the snippets because Figure 3.1(a) holds concrete values (line 7) and Figure 3.1(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. To address these challenges we present an approach that sidesteps them, using an external source of information - textual descriptions.
Figure 3.1: Semantically related fragments that generate all possible permutations. (a) is written in Python and (b) in Java.
Our Approach simon has two parts, one for training and the other for performing relatedness queries. Below we describe the querying process. In the training phase, we build the description mapping, based on sources that correlate code and text. We also use portions of these textual descriptions as our text similarity train set. We elaborate on this step in Section 4.2.

The similarity querying process contains the following high-level steps:

(i) linking code fragments to corresponding textual descriptions,
(ii) extracting type signatures from code fragments,
(iii) comparing textual description and type signatures.

Step 1 - Linking Code to Textual Descriptions The first step is to link the code fragments to their corresponding textual descriptions. Table 3.1 shows the STACKOVERFLOW questions, which contain the programs from Figure 3.1 as a possible answer. We build the textual description based on this data. In this example, for convenience, we represent the textual description as the title only. However, simon uses the content of the entire question. Section 6.1.1 elaborates on the retrieval method we use to find these descriptions, and Section 5.3 specifies the text processing steps.

Step 2 - Extracting Type Signatures Next, we use static analysis to extract the type signature (inputs → outputs) from a code fragment (Section 5.4

Table 3.1: Information about the code fragments in Figure 3.1

<table>
<thead>
<tr>
<th>Snippet (a)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to generate all permutations of a list in Python?</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>How do you generate all the permutations of a list in Python, independently of the type of elements in that list? For example: <em>(some code)</em></td>
<td></td>
</tr>
<tr>
<td>Tags</td>
<td></td>
</tr>
<tr>
<td>python, algorithm, permutation, combinatorics, python-2.5</td>
<td></td>
</tr>
<tr>
<td>Votes</td>
<td></td>
</tr>
<tr>
<td>171</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Information about the code fragments in Figure 3.1

<table>
<thead>
<tr>
<th>Snippet (b)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generating all permutations of a given string.</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>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?</td>
<td></td>
</tr>
<tr>
<td>Tags</td>
<td></td>
</tr>
<tr>
<td>java, algorithm</td>
<td></td>
</tr>
<tr>
<td>Votes</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.2: Simon’s core; mapping snippets to natural language elements provides more details about the type signatures). This step is used to improve the precision of the text-based similarity. In our example, for the snippet of Figure 3.1(a), we extract the type signature $\text{String} \times \text{Iterable} \rightarrow \text{String}$, and for the snippet of Figure 3.1(b) we get $\text{String} \rightarrow \text{Set}(\text{String})$.

**Step 3 - Comparison** The third step is the comparison between the new entities. We consider the titles as the codes’ descriptions, the relatedness between them is high, and indeed, their similarity score is higher than 0.79, indicating of strong similarity. This value was computed using a state-of-the-art text similarity approach, specialized for the programming domain, as explained in Section 5.3. The strong affinity between the type signatures allows us to increase the confidence of the text similarity score.

### 3.2 Intuition

Figure 3.2 shows a simplified view of our approach (the actual system architecture is shown in Figure 3.3). Our goal is to determine semantic similarity between the code snippets $s$ and $s'$. We do that by mapping the snippets to corresponding textual descriptions and computing semantic similarity between their descriptions. 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 textual descriptions to determine the semantic similarity of code fragments.
3.3 Key Aspects

This example shows some key aspects of SIMON that distinguish it from existing techniques:

- **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 (language specific in this case) libraries.

- **Similarity between partial programs**: SIMON determines similarity between code fragments where input and output are not explicitly defined (e.g., without return value, without explicit parameters).
Chapter 4

Leveraging Collective Knowledge

In this chapter, we present the data used to build SIMON. We describe how SIMON leverages collective knowledge towards a similarity notion.

4.1 Open World Approach

Sites such as GitHub and programming blogs are storing massive amounts of code, often with additional meta-data. Programmers interact with each other by sharing code or asking questions (using natural language) and creating associations between code and its natural language description. Specifically, we can rely on Question-Answering sites, such as Stackoverflow, JustAnswer or Google Groups, that provide millions of code fragments associated with textual descriptions.

Stackoverflow (SOF) is a community question-answering site that allows programmers to post any computer programming related question. Each question is associated with a title, content and tags, chosen by the author to describe the question. Each of these elements can be modified by the crowd. SOF creates an implicit mapping between code fragments and their descriptions. For our analysis, we used the SOF dump from September 2014, provided by the MSR challenge [42]. The dump contains more than 21M posts, divided into almost 8M questions and more than 13M answers. Figure 4.1 shows pair of similar code fragments that were extracted from
from itertools import groupby
[ key for key, _ in groupby(sortedList)]

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

Figure 4.1: Similar code fragments that implement distinct representation of a given list

SOF.

4.1.1 Trouble in Paradise

This work presents a radical departure from the common approaches to determining similarity between code fragments. An obvious major challenge for our approach is the ability to find representatives in the pre-computed database for an arbitrary pair of snippets provided by a user. Indeed, the quality of our results may vary widely based on the quality of representatives found in the database. Section 5.2 presents more details about the code retrieval step.

That being said, for code snippets that serve a common purpose (and are commonly used), 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.

4.1.2 The Importance of Data

Figure 4.2 shows the percentage of code fragments that have syntactically close code 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. We can see that the larger the database, the more adjacent fragments are found. This emphasizes the potential of our approach. Section 7.3 presents another point of view regarding the ability to obtain good program coverage.
4.2 Training Phase

Utilizing this data, the first component of simon is the building of a massive code-to-descriptions mapping. Performed only once, this step is a crucial component, used by all the following steps. To build the Semantic Description of a piece of code, we will use the text of a question from SOF, whose the code is within its answers (or other syntactically similar code). We created a Description Oracle, which contains more than one million code fragments linked to their descriptions, as described in Section 6.1.1.

In addition, we carry out natural language text similarity model training, using more than \(1M\) textual descriptions exported from SOF.

4.3 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 by itself. It contains 6500 labeled pairs, based on more than 10000 user tags, and is continuously growing. The possible classifications and explanation about how this information was obtained using the crowd can be found in Section 6.3. Because Similarity is not Equivalence, it is not always clear whether two code fragments are really similar. Therefore, we had to build this corpus care-
int x = Integer.parseInt("8");

(a)

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

(b)

Figure 4.3: Two code fragments with inconclusive similarity level, (a) written in Python, and (b) in Java

fully. The corpus base is the crowd’s knowledge, and each pair's similarity was determined by agreement among several users. One example for a pair whose similarity is inconclusive and might be tagged as similar or different, depending on who labeled it, can be found in Figure 4.3. In principle, both fragments transform a string number to its integer form. However, the first can handle any integer while the second is limited to digits only. This might lead to different opinions among users.
Chapter 5

Description Based Relatedness

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

5.1 Semantic Relatedness of Code Fragments

The term Semantic Relatedness initially appeared in the NLP domain, as a finer case of Semantic Similarity. Similarity is based on is-a relations and can’t be established across different parts of speech. The notion of relatedness captures similarity relations, but also includes other relations, such as: has-part, is-made-of [27].

We adapt and adjust this term for the programming domain. We consider relatedness between two code fragments based on their functionality, including similar behaviors of the programs, inclusion or opposite functionalities. Problems regarding this terminology are discussed in Chapter 10.

5.2 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 appropriate matches, in order to borrow their textual description. Good matches are code fragments that can share the same textual descriptions to the one we have. Towards that end, we look for syntactically similar snippets.
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 5.1: Two syntactically similar code fragments

5.2.1 From Code Fragment to Descriptions

Given two arbitrary code fragments $s$ and $s'$, and a description oracle containing a massive number of code fragments with their textual descriptions, we would like to compute the semantic similarity $\text{sim}_{\text{code}}(s, s')$ between $s$ and $s'$, or the opposite notion of distance:

$$\text{sim}_{\text{code}}(s, s') = 1 - \Delta(s, s').$$  \hfill (5.1)

Often, the snippets don’t appear as in the oracle, but with some negligible differences, such as naming, formatting or even 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 literals.

Monperrus et al. [22] investigated whether SOF’s search engine can handle code snippets as input. They searched for code, extracted from the site, but only 15% of the snippets from their test dataset were found. Due to these results, our dealing with multiple programming languages, and our desire to find snippets that don’t necessarily appear in the exact same way, we had 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 the programming languages, and (ii) the second is language specific and based on the syntax of the chosen language.
5.2.2 Generic Syntactic Similarity

To deal with multiple programming languages we extract only the relevant tokens from each fragment - parameters, values, 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 global pairwise alignment of the common tokens.

As an example, consider the two code fragments of Figure 5.1.

5.2.3 Language Specific Syntactic Similarity

The first approach yields good results, tested on a subset of our database; however we suggest another technique, which is more complicated and customized for code. This technique is language specific and based on AST structure. For each snippet, we create a representative string; Figure 3.1(a)’s string is:

\[\text{Assign Name Store Call } \text{urlopen Attribute Name Load Load Str Assign Name Store Call } \text{read Attribute Name Load Load}\]

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 function names are given a higher penalty.
5.3 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 explain how to construct such a Description Oracle in Section 6.1.1. 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 stemming (changing each word to its base form), removing stop words and punctuation signs. Figure 5.2 shows a text processing demonstration.

Pilehvar et al. [28] claim that techniques such as LSA are still most suitable for long texts. Hence, we decided to combine various text similarity techniques. For titles, which are short and play a major role in the description, we use ADW with software specific specialization as described in Section 5.3.1. For question content, which is a longer text, we use LSA on top of tf.idf. LSA was trained with more than 1M SOF posts and with a reduction to 300 dimensions, after experimenting with different values. We compare the constructed vectors using Cosine Similarity.

5.3.1 Specialized NLP

Programming has many domain specific terms, such as String and Int. WordNet based approaches such as ADW might have difficulty dealing with these unique words and might produce incorrect results. 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 data. To deal with this problem we apply a query expansion approach that is based on data from a software-specific word similarity database [37]. For each compared text, we look for words that originate from the programming domain. Each found word is searched within the database and its most related words (if exist) are added to the text. In this way, we specialized the NLP techniques for the programming context and words that are similar only in this context are linked and increase the texts’ similarity score.
import urllib2
res = urllib2.urlopen('http://www.example.com')
html = res.read()

Figure 5.3: Code fragment that has a $\text{String} \rightarrow \text{String}$ type signature

![Data flow graph](image)

Figure 5.4: Figure 5.3’s data flow graph

5.4 Type Signatures

While keeping in mind the problem of similarity across a variety of programming languages, we wanted to utilize the code itself to support the indirect text-based similarity function. Towards that end, we use the codes’ type signatures, as another measure for similarity.

**Example** Consider the code fragment from Figure 5.3. The code finds, for a given string (that represents an URL address), its html source. The string is hardcoded as part of the code, but the functionality of the code, the fact that it finds the html source, is the thing we want to isolate.

Figure 5.4 shows the data flow of this code fragment. The string literal is not affected by any other variable, 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` and the function `read`, it finds the html source, which is also a string. We can see that the first node has in\textit{degree} = 0 and the last node has out\textit{degree} = 0, so we can say that the type signature of this code is indeed $\text{String} \rightarrow \text{String}$.

5.4.1 Snippet Inputs and Outputs

A Snippet’s Type Signature is a relation between its 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 3.1, we build the data flow graph as explained in Section 6.2.2 and skim for nodes with indegree and outdegree equal to zero. These nodes are the snippet’s inputs and outputs, respectively.

5.4.2 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 in this way we can compare signatures originating 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, close types, such as iterable and list, will get a higher similarity score than totally different ones, such as number and dictionary.

5.5 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:

\[
sim(s, s') = \alpha \cdot \max_{d(s, \hat{s}) < \epsilon, \ d(s', \hat{s}') < \epsilon} \ \widehat{\sim}(\hat{s}, \hat{s}') + (1 - \alpha) \cdot \sim_{\text{sig}}(s, s')
\]

where

\[
\widehat{\sim}(\hat{s}, \hat{s}') = \beta \cdot \sim_{\text{title}}(\hat{s}, \hat{s}') + (1 - \beta) \cdot \sim_{\text{content}}(\hat{s}, \hat{s}')
\]

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 \(\epsilon\).
• \textit{sim}_title uses the ADW similarity algorithm applied to titles that have been enriched with software-oriented query expansion.

• \textit{sim}_content uses LSA similarity that has been trained on millions of textual software descriptions.

• \textit{sim}_sig measures similarity between type signatures based on the correlation between the types revealed in type flow analysis.

Algorithm 1 shows a step by step illustration of the similarity computation between two given code fragments. This is a sequential version of the algorithm, while many steps can (and are) executed in parallel.

\textbf{Algorithm 1:} Finding the similarity score of two given snippets

\begin{itemize}
  \item \textbf{Input:} Snippets \( s, s' \)
  \item \textbf{Output:} Similarity score between \( s, s' \)
\end{itemize}

\begin{algorithm}
\begin{itemize}
  \item \( \hat{s} = \text{findInDB}(s) \)
  \item \( \hat{s}' = \text{findInDB}(s') \)
  \item \( t = \text{getTextualDescription}(\hat{s}) \)
  \item \( t' = \text{getTextualDescription}(\hat{s}') \)
  \item \( t_{spec} = \text{softwareSpecialization}(t) \)
  \item \( t'_{spec} = \text{softwareSpecialization}(t') \)
  \item \( s_{sig} = \text{extractTypeSignature}(s) \)
  \item \( s'_{sig} = \text{extractTypeSignature}(s') \)
  \item \( \text{title}_{sim} = \text{ADW}(t_{spec}, t'_{spec}) \)
  \item \( \text{content}_{sim} = \text{LSA}(t, t') \)
  \item \( \text{sign}_{sim} = \text{getSignaturesSimilarity}(s_{sig}, s'_{sig}) \)
  \item \( \text{simScore} = \alpha \cdot \text{title}_{sim} + \beta \cdot \text{content}_{sim} + \gamma \cdot \text{sign}_{sim} \)
  \item \textbf{return} \text{simScore}
\end{itemize}
\end{algorithm}
Chapter 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 ran on an Intel(R) Core(TM) i7−2720QM CPU @ 2.20Ghz machine with 16GB installed RAM.

6.1 Training

6.1.1 Building a Description Oracle

We chose to construct our Description Oracle based on SOF, due to its volume and coverage. SOF 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 its answers. The semantic

<table>
<thead>
<tr>
<th>Type</th>
<th>(\text{avg}_{\text{votes}&gt;0})</th>
<th>(\text{num}_{\text{votes}&gt;0})</th>
<th>(\text{num}_{\text{all}})</th>
<th>(\text{num}_{\text{votes}&gt;2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question</td>
<td>3</td>
<td>&gt; 3.7M</td>
<td>&gt; 7.9M</td>
<td>&gt; 1.1M</td>
</tr>
<tr>
<td>Answer</td>
<td>3</td>
<td>&gt; 8.1M</td>
<td>&gt; 13.5M</td>
<td>&gt; 2.8M</td>
</tr>
</tbody>
</table>

Table 6.1: Statistics about Stackoverflow posts
description is composed using the title and the question text, with different weights 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 obtained by Jeon et al. [13] in previous research regarding question similarity.

Because SOF is crowd-based, it might sometimes contain wrong answers, which are expressed by wrong code fragments. In order to deal with this challenge, we use the SOF 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 6.1 shows the collected information. According to Nasr et al. [23] 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 SOF 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 take into account 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.

### 6.1.2 Parameter Tuning

We applied some machine learning techniques to achieve the best results. Our system contains several parameters, such as the weight assigned for each question part, the type analysis influence 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, containing many pairs of code fragments and a boolean, which represents whether the fragments are similar, as described in Section 6.3.

We use our labeled data, and look for the best values using ten-fold cross-validation, which is commonly used as a solution for problems of model selection. Ten-fold cross-validation means that the dataset is randomly split into 10 equal size subsets. With 10 validation steps, trained on 9 out of 10 subsets and tested on the remaining one, we can find the best values [17]. We found the parameters that achieved the highest measurement values (as discussed in Section 7.1.1), while maintaining relatively low deviation, and
set them to be our system parameter values. Our analysis showed that 0.35 of the score should come from the titles’ similarity, and 0.5 from the entire question text. The type signatures get 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 Title* Titles are mostly short and informative (e.g., the titles in Table 3.1). Hence, we decided to use query expansion [39], expanding only terms from the programming domain to avoid reduction in the results quality. The specialization is based on the database created by Tian et al. [37]. Using the expanded titles we used ADW [28] 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 [37].

*For Content* The combination of the title and question content is longer text (mostly around 5 – 6 sentences). We used tf.idf and SVD (as implemented in Scikit Learn for Python) to implement LSA. We trained the model once, on top of 1M texts, and stored it for future use.

6.2.2 Signature Similarity

To infer the type signature of a given snippet we visit the AST and extract type and flow information. The snippets we work with are partial programs; many times they are not even 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, fake declarations are added, and in Python the indentation is fixed. After we get a parsable code, we generate its AST and then we visit all its nodes. Each node gets special treatment, so we can decide which information we want to keep. For Python, which is
a dynamically typed language, we 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 type signature similarity of two given code fragments, we first try to match each type in one signature to its best fit in the other signature. Perfect matches (exactly the same cross-language type) were assigned a higher score. Partial matches (e.g., set and list), were assigned lower score. The final value is in the range of $[0, 1]$. We implemented our analysis for Java and Python.

### 6.2.3 Quantitative Similarity between Code Fragments

Given two code fragments $s$ and $s'$, we would like to define a quantitative similarity metric $\text{sim}_{\text{Code}}(s, s') \rightarrow [0, 1]$ between $s$ and $s'$, such that measures the semantic similarity of the code fragments. In other words, $\text{sim}_{\text{Code}}(s, s')$ should measure similarity of the functionality of $s$ and $s'$, regardless of their syntactic differences. As closer the code fragments, as higher their similarity score.

### 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 [36], so we decided to exploit it.

We developed a crowd-source 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 level. The possible tags are: Very Similar, Pretty Similar, Related but not Similar, Pretty Different and Totally Different. We designed a Human Intelligence tasks (HITs) 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 many users’ answers. Because we couldn’t blindly trust any user, we had to integrate a
```java
rand.nextInt((max+1) - min) + min;
```

(a)

```java
Random r = new Random();
int i1=r.nextInt(80-65) + 65;
```

(b)

<table>
<thead>
<tr>
<th>Very Similar</th>
<th>Pretty Similar</th>
<th>Related</th>
<th>Pretty Different</th>
<th>Very Different</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.1: Two code fragments and their user classification distribution

*trust test* into the task. Each fifth pair was taken from a chosen 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 is often the same, e.g., *very similar and pretty similar*, but the specific tags are not. We also saw that a large portion of the *Related* labels were accompanied by *Similar* labels, hence we decided to treat *Related* labels as *Similar* labels for our binary evaluation (note that simon’s output is quantitative).

Figure 6.1 shows an example pair and its variant user classifications. This example illustrates that each user has different similarity scale; however, there is consensus of similarity and therefore we regard this pair as similar.
Chapter 7

Evaluation

7.1 Similarity Classifier Evaluation

We compared our results with tags obtained from LIKE2DROPS and computed the precision, recall and accuracy of similar code fragments. The experimental database contains 6,500 pairs of code fragments and we collected more than 10,000 user classifications. For most pairs, all the answers indicated the same label: similar or not. In some cases, however, we couldn’t decide what the correct label is because the answers varied greatly. This was the case for around 6% of the human-labeled pairs, leading us to conclude that no conclusive decision is possible. We therefore omitted these pairs from our experiment.

Despite our efforts to get many answers for each pair (e.g., many users opinions), there is 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 analyzed a representative group (|group| > 200) of wrong tags. Figure 7.1 shows the error distribution. Many errors were caused by wrong labels, given by one user.

We tried to understand if agreement between more than one user can help us achieve better results. Table 7.1 shows the different measurements achieved using pairs with an increasing agreement threshold. Notably, even agreement between two users dramatically improves our 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 to always be perfect.
The results show that 82% of our labels are consistent with the users’ labels, while the precision is 80.3% and the recall is 80%. Table 7.2 shows the results using different configurations of simon; note that we chose the best threshold for each configuration, and that these 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 the above 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 types signature. Row 3 is the full configuration, using the text analysis, without software specific specialization. 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. 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. These results were achieved with pre-processed code, and the results without this step are less accurate (average 10%). No major differences between the different configurations are evident, except for the naive approach, which yields worse results.

The relatively high values achieved using the tokenized code can be ex-
Table 7.1: Results obtained by omitting pairs with low user confidence

<table>
<thead>
<tr>
<th>Answers per pair</th>
<th>#Labels</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>3951</td>
<td>80.3%</td>
<td>80%</td>
<td>82%</td>
<td>0.894</td>
</tr>
<tr>
<td>&gt; 1</td>
<td>1792</td>
<td>86.9%</td>
<td>84.1%</td>
<td>86.8%</td>
<td>0.9363</td>
</tr>
<tr>
<td>&gt; 2</td>
<td>725</td>
<td>90.3%</td>
<td>87.3%</td>
<td>90%</td>
<td>0.9636</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>344</td>
<td>90.2%</td>
<td>84.1%</td>
<td>88.1%</td>
<td>0.9576</td>
</tr>
<tr>
<td>&gt; 4</td>
<td>212</td>
<td>93.7%</td>
<td>87.3%</td>
<td>91%</td>
<td>0.9758</td>
</tr>
<tr>
<td>&gt; 5</td>
<td>145</td>
<td>95.4%</td>
<td>87.5%</td>
<td>91.7%</td>
<td>0.9838</td>
</tr>
<tr>
<td>&gt; 6</td>
<td>95</td>
<td>95%</td>
<td>88.4%</td>
<td>93.7%</td>
<td>0.9924</td>
</tr>
<tr>
<td>&gt; 7</td>
<td>56</td>
<td>95.2%</td>
<td>83.3%</td>
<td>79.1%</td>
<td>0.9857</td>
</tr>
</tbody>
</table>

Table 7.2: Results obtained by omitting different parts of simon and comparison to other techniques

<table>
<thead>
<tr>
<th>#</th>
<th>Configuration</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Full approach</td>
<td>80.3%</td>
<td>80%</td>
<td>82%</td>
<td>0.8939</td>
</tr>
<tr>
<td>2</td>
<td>Full, without types signature</td>
<td>79.4%</td>
<td>80.9%</td>
<td>81.7%</td>
<td>0.8932</td>
</tr>
<tr>
<td>3</td>
<td>Full, without specialization</td>
<td>79.5%</td>
<td>78.6%</td>
<td>81.1%</td>
<td>0.8918</td>
</tr>
<tr>
<td>4</td>
<td>Full, without ADW &amp; specialization</td>
<td>77.5%</td>
<td>80.7%</td>
<td>80.6%</td>
<td>0.889</td>
</tr>
<tr>
<td>5</td>
<td>Full, without LSA</td>
<td>79%</td>
<td>78.4%</td>
<td>80.7%</td>
<td>0.8912</td>
</tr>
<tr>
<td>6</td>
<td>Only titles (ADW &amp; specialization)</td>
<td>79.5%</td>
<td>69.2%</td>
<td>77.9%</td>
<td>0.854</td>
</tr>
<tr>
<td>7</td>
<td>Only content (LSA)</td>
<td>73.5%</td>
<td>80.8%</td>
<td>78%</td>
<td>0.8428</td>
</tr>
<tr>
<td>8</td>
<td>Tokenized-code, tf.idf</td>
<td>76.7%</td>
<td>70%</td>
<td>76.7%</td>
<td>0.79</td>
</tr>
<tr>
<td>9</td>
<td>Tokenized-code, LSA</td>
<td>77.7%</td>
<td>63.9%</td>
<td>75%</td>
<td>0.7832</td>
</tr>
<tr>
<td>10</td>
<td>Random</td>
<td>45.6%</td>
<td>47.4%</td>
<td>50.7%</td>
<td>0.4993</td>
</tr>
</tbody>
</table>

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. Moreover, the contribution of the type signatures is almost unobservable due to a bias caused by relatively few (< 400) pairs having signatures (implemented only for Java and Python).

7.1.1 Measurements

In our case, we denote Precision as the fraction of code fragment pairs that are labeled as similar, and are indeed similar. Recall is denoted by the fraction of similar code fragment pairs that are actually labeled as similar. Accuracy is the fraction of correct labels.
(7.1)-(7.3) are the accuracy, precision and recall formulaes. True Positive (TP) are similar pairs that we found. True Negative (TN) are different pairs that we also labeled as different. False Positive (FP) are false alarms (e.g. different pairs that were labeled as similar) and False Negative (FN) are the similar pairs that we missed.

When building a classifier the challenge is to find a good threshold (used to separate between similar and different samples), across experiments. Plotting the accuracies obtained using different thresholds yields a Receiver Operating Characteristic (ROC) curve. The accuracy of the evaluated classifier is the Area Under the Curve (AUC) value [4]. The ROC curve gives us the ability to compare different classifiers.

\[
\begin{align*}
\text{Accuracy} & = \frac{TP + TN}{TP + TN + FP + FN} \quad (7.1) \\
\text{Precision} & = \frac{TP}{TP + FP} \quad (7.2) \\
\text{Recall} & = \frac{TP}{TP + FN} \quad (7.3)
\end{align*}
\]

### 7.2 Code Retrieval Evaluation

We performed two different types of experiments to evaluate our ability to query the Documentation Oracle. Partial database analysis (|database| > 100,000) revealed that the average number of close code fragment matches is 2.7, while the maximum value is 175 and the minimum is of course 0.

#### 7.2.1 Generic method

We checked two sets of snippets from multiple programming languages:

(i) randomly picked 25 fragments from our database (control group), and

(ii) randomly picked 25 fragments from our database, with mutations.

We used each of the fragments 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.
Table 7.3: Our code retrieval results based on a test set with 1,000 code fragments and 55 search queries

<table>
<thead>
<tr>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>94%</td>
<td>92.1%</td>
<td>99.987%</td>
</tr>
</tbody>
</table>

To get a more accurate evaluation, we created a test set with 1,000 code fragments. A programmer with more than 7 years of experience tagged each of the 55,000 pairwise options of query and database entry. Using this labeled data we managed to compute the precision, recall and accuracy as shown in Table 7.3.

7.2.2 Python-specific method

To evaluate the Python-specific syntactic similarity we initialized the database with more than 1,000 pairs. These pairs were chosen by a tagging process (within our 1M database), using simon. We collected pairs that were labeled as similar (Figure 7.2 shows an example for such pair), and another programmer, with more than 8 years of experience, validated the results and determined whether the two are indeed similar.

This evaluation is quite simple, but precision-oriented. Looking for identical fragments, modifying only their variable names, parameter values or formatting, yields perfect results. The expert’s labels show 94% agreement with simon’s labels.

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.

7.3 Coverage

"Although the number of legal statements in the language is theoretically infinite, the number of practically useful statements is much smaller, and potentially finite.” Gabel et al. [9]
>>> wordlist = ['Schreiben
Es', 'Schreiben', 'Schreiben
Eventuell', 'Schreiben
Haruki']
>>> [i.split('
')[0] for i in wordlist]
['Schreiben', 'Schreiben', 'Schreiben', 'Schreiben']

(a)

>>> la = ['a.b.c.d', 'a.b.c', 'y.d.k', 'z']
>>> [elem.split('.')[0].split('.')[0] for elem in la]
['d', 'c', 'k', 'z']

(b)

Figure 7.2: Syntactically similar code fragment, found using simon

Getting a full mapping coverage of any code fragment that is possible to write is of course impossible. This is mainly because the program's domain is infinite. With that said, software is usually an aggregation of much smaller parts. These parts are repetitive and by mapping them, we may get good coverage of the code fragments that are likely to be used by a programmer.

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, which helps us to link new snippets to the ones we already have in our mapping.

With that said, we can’t guarantee a solution for any random code fragment that one might think of, and we leave this part for future research.

7.4 Performance

Performance was not a priority while implementing simon, and therefore no major optimization steps were implemented. However, we aimed to achieve reasonable computation times. With that said, 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.

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 second in the generic retrieval method, and 1 second for the Python-specific method.
Chapter 8

Applications

In this chapter we present some of the applications that are possible using our approach. It is only a partial application list while the possibilities are vast and one can easily find many other use cases.

8.1 Automatic Tagging of Snippets

We can address the problem of automatic tagging of code fragments. Given a code fragment, the goal of automatic tagging is to predict a set of textual labels that describe the semantics of the code fragment. The challenge is to find descriptive labels beyond tokens that appear in the snippet or ones that can be obtained from documentation of APIs used in the snippet.

Existing work on synthesis of high-level descriptions from code fragments [35] typically works by structural analysis of the code itself. While this has many advantages, this approach is often limited to describing the code only in terms of its actual actions (in terms of the solution), without any mention of the problem that is being solved.

Recently, Rastkar et al. [31], used extractive classifier-based techniques to summarize bug reports. A long-term goal is to be able to produce similar natural-language summaries for code snippets, and our approach provides a first step towards that goal.

Using our mapping from code fragments to their textual descriptions we can develop a framework for automatic tagging of code snippets, based on label extraction from their corresponding natural language descriptions.
We implemented this approach using tf.idf as our keyword extraction scheme and tagged more than 7000 snippets. We took only initial step towards this direction and it is yet far from being perfect; however, we found many examples where even a naive implementation works well. Table 8.1 shows sub-set of these examples.

### 8.2 Code Search

There has been some work on searching code using natural language queries [15], and work on query expansion to enable more natural search in source code. Again, these are mostly based on analysis of the code itself. Our approach moves us step forward in the field of code search.

First, we can utilize the code to textual description mapping to search for code based on it textual description. The description represents the functionality of a snippet, thus we can look for by what it actually does, using natural language description.

Another option is to search for snippets using semantically similar snippets that perform the same functionality. It can help to find semantic duplicates within a source code or to learn how to implement the same func-

---

**Table 8.1: Code fragments and their extracted tags as established using our mapping and native keyword extraction method**

<table>
<thead>
<tr>
<th>Code</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if not a: print &quot;List is empty&quot;</code></td>
<td>list • empty • check</td>
</tr>
<tr>
<td><code>function remove(arr, item) { for(var i = arr.length; i--; ) { if(arr[i] === item) { arr.splice(i, 1); } } }</code></td>
<td>remove • array • element</td>
</tr>
</tbody>
</table>
| `import os.path
os.path.isfile(fname)`                                         | file • exists • check             |
| `sorted(d.items(), key=lambda x: x[1])`                              | dictionary • sort • value         |
| `import java.util.UUID;
String uuid = UUID.randomUUID().toString(); System.out.println("uuid = " + uuid);` | random • string • generate        |
| `int foo = Integer.parseInt("1234");`                             | string • int • converting         |
tionally in different manner or even different programming language. So it is also have an educational aspect.

Furthermore, it can help to translate code fragments between different languages or to create clusters of related code fragments that can be used for variant purposes (note that the triangle inequality does not hold with cosine similarity as the distance metric; however, the metric can easily be replaced).

8.3 NLP Technique Enriching

We can also go the other way around. When we know about the similarity of two pieces of code (using different code similarity techniques or the data achieved from our crowdsourcing system), we can utilize this data to spot connections between textual entities.

Assuming we have two similar code fragments and their corresponding textual description, we can say that their descriptions are also likely to be similar. This knowledge can be leveraged as an input to the training phase of text similarity methods or for the enriching of relatedness networks such as WordNet.
Chapter 9

Related

In this chapter, we survey existing work regarding variant related topics.

9.1 Cross-Language Code Similarity

The topic of cross language code similarity has not been widely investigated. However, some research exists. Flores et al. [8], 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 both source codes, using basic NLP techniques, such as \( n \)-grams and tf. To measure the distance between two code representations, they used cosine similarity.

Also, there has been some work that relies on probabilistic models of programs based on large code bases [14]. In this work, Karaivanov et al. suggested an approach for program translation based on mapping between C# and Java grammars.

9.2 NL as a Tool for Programming Tasks

Hindle et al. [11] were the first to apply statistical language models originating from NLP to programming languages. They showed that code is even more repetitive and predictable than natural language and hence can be modeled using statistical language models. Furthermore, they suggest that snippet retrieval can relay on English description, a task we deal with.
Kuhn, Ducasse and Gírba [18] 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. Such that, one can enrich its software analysis with the programmer knowledge that reflects in its vocabulary use. Applying such techniques directly on the code may be the first step of building the textual descriptions for “new” code fragments.

Vinayakarao et al. [38] 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.

There has been a lot of other work on the use of natural language in software engineering tasks. Some work has addressed the problem of inferring specifications from natural language descriptions [24, 45]. Other works have dealt with security policy extraction [40], detection of API documentation errors [44], bug identification [10], code convention modification [1], and source code search [16].

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.

9.3 Code Retrieval

Mishne and De Rijke [21] used conceptual modeling of code to perform the task of code retrieval, avoiding the differences in programming language syntax. They represent each code as conceptual graph (CG) [34] a bipartite, directed, finite graph; each node in the graph is either a concept node or relation node and thus able to capture the content and the structure of the code. They also suggested similarity matric, called Contextual Similarity. They combined their approach with the probabilistic retrieval model Okapi [33] to get the best results. This method is based on a given grammar rules of the language, so it can support variant programming languages.

Ponzelli et al. [30] 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 using the code itself, along with other things (conceptual and community). Each code snippet is transformed to a query that is sent to a search engine in order to find the relevant pages and retrieve the desired information.

9.4 Question Similarity

One of the building blocks of our work is the ability to correctly perform the question similarity task. Jeon, Croft and Lee [13] present a way to find semantically similar questions in community-based question and answer services. They built set of question pairs, such that they are lexically different but semantically similar. With that information they could learn the translation probabilities of words and use them to retrieve similar questions.

9.5 Similarity Learning

Another related area is that of similarity learning. The need to measure the similarity between two elements is crucial for machine learning, bioinformatics and information retrieval [2]. It also has vast implication in the areas of computer vision and medicine [7]. In similarity learning, the goal is to learn similarity function based on labeled examples. In our approach, we classify the similarity of code fragments; we learn new similarity metric over code fragments. To train our tool as a classifier, we searched for the best parameters (threshold and weights) based on a labeled training set, as explained in Section 6.1.2. This step is Statistical Classification, which is a sub-area of similarity learning.
Chapter 10

Discussion

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

10.1 Why 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 between words. The use of more advanced techniques can help us to deal with the randomness associated with human expression. One example can be 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 strong affinity between the descriptions; however if we don’t use ADW or LSA, simple techniques might miss this connection and classify the two as different.

10.2 Why Software Specialization?

Software has many unique words that are not part of “everyday speech”, such as type names (e.g., String, Int, Integer), keywords, etc. It also has words that have a different meaning than what we used to, 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 specific texts, a fact that we would miss without the specialization.

However, we did not observe the specialization to have significant impact on the results in general. This is probably due to the use of mostly consistent software terms in the descriptions from SOF.

10.3 Why Type Signatures?

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 10.1 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 high similarity score (more than 0.8!); however, their associated programs are not the same. The use of type signatures reveals that the two are indeed different and decreases their similarity score; however, usually the type information plays a more minor role. This is the case for a large percentage of the evaluated pairs, hence the difference in the results is negligible.

10.4 Text Similarity vs. Code Similarity

Semantic text similarity is a hard problem in itself due to ontologies and negation. Much research has been conducted on the subject of semantic text similarity. Common techniques are based on knowledge bases, such as WordNet, which help to clarify the randomness created by ontologies and achieve good results in known benchmarks.

On the other hand, the nature of code makes it complicated to use standard text similarity techniques. Code fragments have different syntax, based on the programming language used, different documentation, formatting, identifiers, data types, API’s, 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
```csharp
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);
}

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

(a)

(b)

Figure 10.1: Two code fragments which are the reverse of each other

in that we have to compare texts that originate from the narrow domain of programming questions, written in one specific language – 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.

### 10.5 Limitations

One of the problems with our approach is that it is heavily based on the crowd. We adopted some elimination techniques, which ignore code fragments that originate from posts with a low voting score or remove bad user classifications. However, it might still be biased by a user, like any other crowd based application. Another limitation is associated with the term *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 with *Related* classifications, some of which indeed follow our definition in Section 5.1, but some of which do not. This fact reduced Simon’s performance and we think that the term *Relatedness between Code Fragments* can be a foundation for future work.
Chapter 11

Conclusion

We presented a novel approach for measuring semantic relatedness between code fragments based on their corresponding natural language descriptions and their type signatures. We implemented our approach in a tool called simon, based on the well-known Q&A site, Stackoverflow, and applied it to determine relatedness between a large number of program pairs. We used the crowd to collect labeled data, which may be of interest by itself. We combined an open world approach, text similarity techniques, and lightweight type analysis, and showed that it leads to promising results.
Bibliography


[7] Issam El-Naqa, Yongyi Yang, Nikolas P Galatsanos, Robert M Nishikawa, and Miles N Wernick. A similarity learning approach


בהתבסס על אתרי שאלות SIMON, באומיתות הנספים וbudikeotes התא البريطון בין למלועה Stackoverflow וһותרות רכזת בין למלועהgeber של הקוד. הנסה שלר ידעת להימדד עם הקוד

מモンטנ שמונה, גם כל שישה וחמשת עשרהульт להתקשות בהתקשות הבחרת עימה.

3.-central_FLAGS על הגישה שלenery, הביב同時に SIMON בדקו את הדמיון בין למלועה.

4. פיתוחו מערכות שלchers תקנות התהוגות בכדי לאפשר תויגים של גנים אפוריים

לזכים רמה הדמיון של תויגים הקוד. אספונים ייחות מ-10,000 תויגים בהחס镎

על למוטה מ-40 מונטימי שווים. באומיתות המרעד והכניס קורפוס הכפלת 6500

יתנו הקוד והווג המעדים על רמה הדמיון בוגה. קורפוס זה הוא בלע

シーזניאל על נסיון בבעמות לחקרין בהחשמ 포함

5. השווינו את תוצאות המערכות שלל קורפוס התהוגים הקוסבע כל הגישה שלenery

מסונכת בכל הקשר בין הדמיון ברוב דוגלו של המחירים נוכח השגת דיקוב
3. **Identifying the Similarity Score Between the Codes**

When we quantify the similarity between codes, we seek a method to evaluate the textual similarity scores between pairs of codes. A common approach uses text similarity methods, which produce a vector for each fragment and then uses the cosine function to calculate the similarity score between the text fragments. When calculating the similarity score between code signatures, we create a group of code types (e.g., sentences) and use similar text similarity methods to find pairs of code signatures. Finally, we sort the code similarity scores and use them to determine the level of similarity between the code.

Defining the similarity score, we use the cosine function for calculating the similarity between text fragments. We construct a similarity matrix for each pair of code types and use the cosine function to calculate the similarity score between the code types. The similarity score is calculated as the cosine of the angle between the two vectors representing the code types. We then sort the similarity scores and use them to determine the level of similarity between the code.

In conclusion, the main contributions of this research are as follows:

1. **Deployment of a New Textual Similarity Score**: We developed a method to calculate a new textual similarity score that is applicable to code types.

2. **Evaluation of the New Score**: We tested the new score on a set of code types and found that it outperformed existing methods.

3. **Application of the New Score**: We applied the new score to a set of real-world code types and found that it accurately identified similar code types.

4. **Implementation of a New System**: We implemented a new system that uses the new similarity score to identify similar code types.

5. **Validation of the New System**: We validated the new system on a set of real-world code types and found that it accurately identified similar code types.

In summary, this research has developed a new text similarity score that is applicable to code types and has demonstrated its effectiveness in identifying similar code types.
We analyze the text similarity by using ad-hoc techniques for various purposes, such as language modeling and automatic document classification. Among these techniques, LSA and tf.idf are the most commonly used. We also use the ADW technique for this purpose.

To find the best matching text, we divide the problem into two main steps. The first step involves finding the code text and then analyzing the text similarity using text mining methods. In the second step, we perform a similarity analysis of the text code. In this step, we use standard techniques after which we focus on the parts of the text code. The first method, which is language independent, is related to the code text. The second method uses a language-dependent technique for the code text. In both methods, we consider the code text for the analysis.

We also use the LSTP algorithm to find the best matching text. The LSTP algorithm is a method for finding the best matching text in a given code. The algorithm works as follows:

1. For each code text, we find the best matching text. We consider the best matching text as the text that has the highest similarity score.
2. For each code text, we find the best matching code text. We consider the best matching code text as the code text that has the highest similarity score.

Finally, we use the LSTP algorithm to find the best matching text. The LSTP algorithm works as follows:

1. For each code text, we find the best matching text. We consider the best matching text as the text that has the highest similarity score.
2. For each code text, we find the best matching code text. We consider the best matching code text as the code text that has the highest similarity score.

At the end, we present the final results, which are the best matching text and code text.
Diminutive code is a key problem faced by developers of large codebases, such as in searching code, translation between different programming languages, or understanding the code. In this research, we present a novel approach to determining code similarity by relying on text-based indexes. In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase. In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.

In this research, we present a novel approach to determining code similarity by relying on text-based indexes. We use a method where code is associated with text-based relations and uses these relations to identify code snippets. We then use these code snippets in conjunction with a set of additional resources such as Stackoverflow. In addition, we extend the approach to cover other languages, particularly those not supported by the codebase.
המחקר נעשתה בתנאיי פורום על ידי בפקולטה למדעי המחשב.

אני מודה למנחת שלי, פרופ' ערן צ'בר, על כל תמיכה ו.Db על נוחותי במדעי המחשב. על עניינות, שיתוף פעולה כל יום שבוע חדש של אחרון העבידה את התוכנית. בכל יום עבידה זו בגרותנו, מתמתא. אני מודה לрабатываו, פרופ' ערן צ'בר, שיר דייט, שחברתי עם בורסלי יוסי אחéri, ושלושת מחקראי קשב. המחקר ממחזרים המ biçimתיmarried כי בgréות, ובמחוז התהליך, וה⊂ה המוניטזיה,𝑔ידורטורת ביבאנד,cל מחזורים,文化艺术 בניהלוון ביבן.
דמיוון קופר בامجיצות הקשורים

טקטוסאלים

היבר על מחקר

לשם מילוי חקיק של הדירויות לקבצל היות
מניסיון למיעום למidine המחשב

מיטל צילברשטיין (בן סיני)

הנהל סקט המכלן – מכון טכנולוגי לישראל
נוכז לכתם
הfait
אפריל 2015
ד미ון קוד באמעעות הקשרים
tekstoraliem

מתיו גולברשטיין (בן סיני)