UNIVERSITY OF L'AQUILA

MASTER THESIS

Automated Approaches to Assess the Similarity of Open Source Software Projects

Author:
Riccardo RUBEI

Supervisor:
Dr. Davide DI RUSCIO
Co-supervisor:
Dr. Phuong T. Nguyen

Corso di Laurea Magistrale in Informatica

Department of Information Engineering Computer Science and Mathematics

Academic Year 2017/2018

Acknowledgements

I would like to express my deepest and sincerest gratitude to my supervisor, Prof. Dr. Davide Di Ruscio, for his expert guidance, constructive criticism, continuous support and encouragement throughout my study. It has been a privilege to work with him and I hope to have the chance to continue working under his supervision in the future.

I would also like to thank my co-supervisor Dr. Phuong T. Nguyen for his supports, as well as for his patience. He has been always available for comments and suggestions.

A special thank to Dr. Juri Di Rocco for his help, in particular during the implementation phase of the software tools. His great expertise is always a good source of inspiration for me.

Also a big thank goes to my dear friends Roberta, Andrea, Claudio, Francesco, Riccardo, Cristina, Rodolfo, who sustained me during my master degree and who never let me go mad.

Finally I would like to thank my mother and my family, without them I could not have been able to complete my study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preamble

Open source software (OSS) repositories contain a large amount of data that has been accumulated along the software development process. Not only source code but also metadata available from different related sources, e.g. communication channels, bug tracking systems, is beneficial to the development process once it is properly mined. Research has been performed to understand and predict software evolution, exploiting the rich metadata available at OSS repositories. This allows for the reduction of effort in knowledge acquisition and quality gain. Developers can leverage the underlying knowledge if they are equipped with suitable tools. For instance, it is possible to empower IDEs by means of tools that continuously monitor the developer's activities and contexts in order to activate dedicated recommendation engines [41].

To aim for software quality, developers normally build their project by learning from mature OSS projects having comparable functionalities. To this end, the ability to search for similar software projects with respect to different criteria such as functionalities and dependencies plays an important role in the development process. Two projects are deemed to be similar if they implement some features being described by the same abstraction, even though they may contain various functionalities for different domains [28]. Understanding the similarities between open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations [46],[54], thereby improving software quality. Meanwhile measuring the similarities between developers and software projects is a critical phase for most types of recommender systems [37],[44]. Similarities are used as a base by both content-based and collaborative-filtering recommender systems to choose the most suitable and meaningful items for a given item [46]. Failing to compute precise similarities means concurrently adding a decline in the overall performance of these systems.

Measuring similarities between software systems has been considered as a daunting task [8],[28]. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail. Given the circumstances, choosing the right tool to compute software similarity is a question that may arise at any time. To this end, the current thesis attempts to address one of the issues in software similarity computation by performing a comprehensive evaluation on various techniques. In particular, we re-implement four software similarity tools and conduct an empirical evaluation using a dataset collected from GitHub.

1.2 The CROSSMINER project

Open source software (OSS) is computer software available in source code form, for which the code and certain other rights are provided under a license that permits users to study, change, and improve the software for free. A report by Standish Group states that adoption of open-source software models has resulted in savings of about 58 billion per year to consumers. Unlike commercial software which is typically developed within the context of a particular organization with a well-established business plan and commitment to the maintenance, documentation and support of the software, OSS is very often developed in a public, collaborative, and loosely-coordinated manner. This has several implications to the level of quality of different OSS software as well as to the level of support that different OSS communities provide to users of the software they produce.

There are several high-quality and mature OSS projects that deliver stable and well-documented products. Such projects typically also foster a vibrant expert and user community, which provides remarkable levels of support both in answering user questions and in repairing reported defects in the provided software. However, there are also many OSS projects that are dysfunctional in one or more of the following ways:

- The development team behind the OSS project invests little time on its development, maintenance and support.
- The development of the project has been altogether discontinued due to lack of commitment or motivation.
- The documentation of the produced software is limited and/or of poor quality.
- The source code contains little or low-quality comments which make studying and maintaining it challenging.
- The community around the project is limited, and questions asked by users receive late/no response and identified defects either get repaired very slowly or are altogether ignored.

Consequently, developing new software systems by reusing existing open source components raises relevant challenges related to the following activities:

- Searching for candidate components.
- Evaluating a set of retrieved candidate components to find the most suitable one.
- Adapting the selected components to fit the specific requirements.

CROSSMINER [1] is a research project funded by the EU Horizon 2020 Research and Innovation Programme, aiming at supporting the development of complex software systems by *i)* enabling monitoring, in-depth analysis and evidence-based selection of open source components, and *ii)* facilitating knowledge extraction from large OSS repositories [1]. In the context of the project, we work towards an advanced Eclipse-based IDE providing intelligent recommendations that go far beyond the current *code completion-oriented* practice. Among others, an indispensable functionality is to find a set of similar OSS projects to a given project with respect to different criteria, such as external dependencies, application domain, or API usage [32],[35].

[1] https://www.crossminer.org

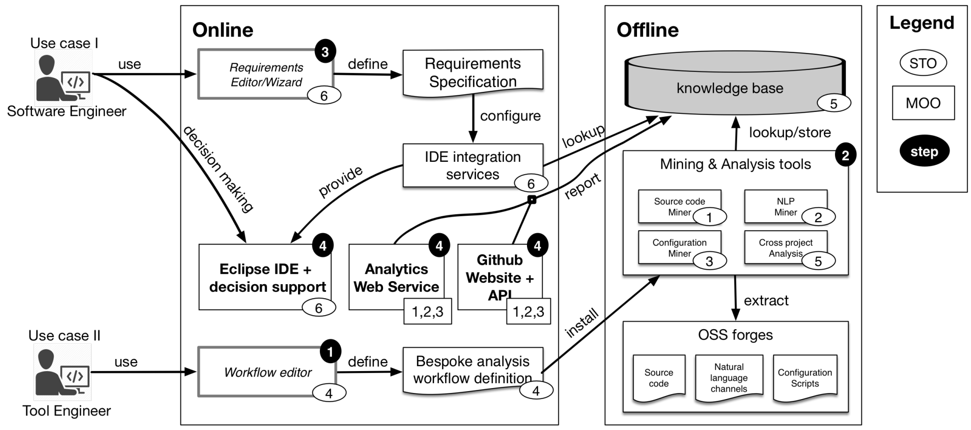

FIGURE 1.1: The CROSSMINER Architecture

CROSSMINER can be seen as a recommendation system aimed at supporting developers while producing new software by integrating existing open source components. Figure 1.1 shows CROSSMINER at a glance and how the project aims at reaching its objectives. Essentially, the *data preprocessing* challenge is addressed by the CROSSMINER components contained in the `Offline` box shown on the right-hand side of Figure 1.1. The developer context is captured by the IDE (see the `Online` box in Figure 1.1), which also produces recommendations that do not require particular and expensive data analysis. For more elaborated recommendations, preprocessed data available in the `Knowledge Base` is used. Both the IDE and Web based dashboards will be used to present the produced recommendations to the developer.

1.3 Problem Statement

The work presented in this thesis is coherently related to the CROSSMINER project and it is dedicated to the problem of recommending similar OSS projects for a given project. We perform a performance evaluation of four software similarity tools, namely MUDABLUE [12], CLAN [28], REPOPAL [54], and CROSSSIM [35] to see how well they perform under common conditions. In this sense, the main research issues that we address in this work are as follows:

- Introduce some state-of-the-art approaches for computing software similarity.
- Re-implement four tools for calculating similarities among OSS projects.
- Compare the performance of the tools using a set of Java projects collected from GitHub.

1.4 Thesis Structure

The thesis is organized in the following chapters:

- Chapter 2 provides a mathematical background related to computing software similarities.
- Some of the most notable approaches for computing software similarity are recalled in Chapter 3.

- Chapter 4 provides a detailed description of the similarity tools, i.e., MUDABLUE [12], CLAN [28], REPOPAL [54], and CROSSSIM [35].
- Chapter 5 presents the evaluation and the experimental results.
- Finally, Chapter 6 sketches out future work and concludes the thesis.

Chapter 2

Mathematical Background

In the field of Information Retrieval [27], there are techniques being widely used in several applications. They can be considered as a basic and indispensable part of a knowledge mining system. Throughout this deliverable some techniques are utilized in different similarity computation algorithms and thus it is worth conducting a review of them. Among others, Term-Document Matrix, Cosine Similarity, Latent Semantic Analysis and Jaccard Index are going to be briefly recalled due to their popularity. Furthermore, later in the chapter we also provide a brief introduction to a graph algorithm [17], which has been exploited to compute the similarity among OSS projects [35].

2.1 Term-Document Matrix

In Natural Language Processing [9], a term-document matrix (TDM) is used to represent the relationships between words and documents [49]. In a TDM, each row corresponds to a document and each column corresponds to a term. A cell in the TDM represents the weight of a term in a document. The most common weighting scheme used in document retrieval is the *term frequency-inverse document frequency (tf-idf)* function [43]. If we consider a set of n documents $D = (d_1, d_2, .., d_n)$ and a set of terms $t = (t_1, t_2, .., t_r)$ then the representation of a document $d \in D$ is vector $\vec{\delta} = (w_1^d, w_2^d, .., w_r^d)$, where the weight w_k^d of term k in document d is computed using the *tf-idf* function [42]:

$$w_k^d = tf \cdot idf(k, d, D) = f_k^d \cdot log\frac{n}{|\{d \in D : t_k \in d\}|} \tag{2.1}$$

where f_k^d is the frequency of term t_k in document d.

Another common weighting scheme uses only the frequency of terms in documents for cells in TDM, i.e., the number of occurrence of a term in a document, instead of *tf-idf*. As an example, we consider a set of two simple documents $D = (doc_a, doc_b)$ as given below:

+ **doc**$_a$: *Julie loves me more than Linda loves me.*

+ **doc**$_b$: *Jane likes me more than Julie loves me.*

The corresponding term-document matrix for D is depicted in Figure 2.1.

TDM has been exploited to characterize software systems and finally to compute similarities between them [12],[20],[28]. In a TDM for software systems, each row represents a package, an API call or a function and each column represents a software system. A cell in the matrix is the number of occurrence of a package/an API/function in each corresponding software system. A TDM for software systems has a similar form to the matrix shown in Figure 2.1 where documents are replaced by software systems and terms are replaced by API calls.

2.2 Cosine Similarity

Cosine similarity is a metric used to compute similarity between two objects using their feature vectors [50]. An object is characterized as a vector, and for a pair of vectors $\vec{\alpha} = (\alpha_1, \alpha_2, .., \alpha_n)$ and $\vec{\beta} = (\beta_1, \beta_2, .., \beta_n)$ there is an angle between them. Intuitively, the cosine similarity metric measures the similarity as the cosine of the corresponding angle between the two vectors and it is computed using the inner product as follows.

$$
\begin{array}{l c}
 & \begin{array}{cc} doc_a & doc_b \end{array} \\
\begin{array}{l} me \\ Jane \\ Julia \\ Linda \\ likes \\ loves \\ more \\ than \end{array} &
\begin{pmatrix} 2 & 2 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 2 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}
\end{array}
$$

FIGURE 2.1: An example of a Term-Document Matrix

$$CosineSim(\vec{\alpha}, \vec{\beta}) = \frac{\sum_{i=1}^{n} \alpha_i \cdot \beta_i}{\sqrt{\sum_{i=1}^{n} (\alpha_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (\beta_i)^2}} \tag{2.2}$$

Figure 2.2 illustrates the cosine similarity between two vectors $\vec{\alpha}$ and $\vec{\beta}$ in a three-dimension space. This can be thought as the similarity between two documents with three terms $t = (t_1, t_2, t_3)$.

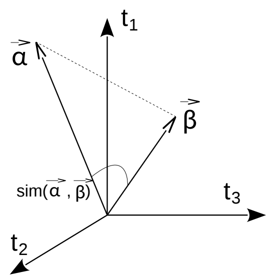

FIGURE 2.2: Cosine similarity between two feature vectors $\vec{\alpha}$ and $\vec{\beta}$

Refer to the example in Section 2.1, the two documents are represented by means of two vectors as follows:

$$\vec{doc_a} = [2, 0, 1, 1, 0, 2, 1, 1]$$

$$\vec{doc_b} = [2, 1, 1, 0, 1, 1, 1, 1]$$

And the similarity between them is computed using Equation 2.2:

$$CosineSim(\vec{doc_a}, \vec{doc_b}) = \frac{9}{\sqrt{12} \cdot \sqrt{10}} = 0.822 \tag{2.3}$$

A similarity score of 0.822 implies that these documents are highly similar.

Cosine similarity has been popularly adopted in many applications that are related to similarity measurement in various domains [14],[15],[21],[26],[29]. Among the similarity metrics being recalled in this deliverable, the prevalence of Cosine Similarity is obvious as it is utilized in almost all of them as follows: *MUDABlue*

[12], *CLAN* [28], *CLANdroid* [20], *LibRec* [48], *SimApp* [8], *WuKong* [52], *TagSim* [23], and *RepoPal* [54].

2.3 Latent Semantic Analysis

The problem with the term-document matrix is that the intrinsic relationships among different terms of a document cannot fully be captured. Furthermore, same words can be used to explain different requirements or the other way around, the same requirements can be described using different words [12]. Latent Semantic Analysis (LSA), also known as Latent Semantic Indexing (LSI), has been proposed to overcome these problems [19]. The technique exploits a mathematical model that can infer latent semantic relationships to compute similarity. LSA represents the contextual usage meaning of words by statistical computations applied to a large corpus of text. It then generates a representation that captures the similarity of words and text passages. To perform LSA on a text, a term-document matrix is created to characterize the text. Afterwards, Singular Value Decomposition (SVD) - a matrix decomposition technique - is used in combination with LSA to reduce matrix dimensionality [4]. SVD takes a highly variable set of data entries as input and transforms to a lower dimensional space but reveals the substructure of the original data. Essentially, it decomposes a rectangular matrix into the product of three other matrices as given in Equation 2.4 [4]. Correspondingly, the decomposition is depicted in Figure 2.3.

$$A_{mn} = U_{mm} S_{mn} V_{mn}^T \tag{2.4}$$

in which

- U_{mm}: Orthogonal matrix.
- S_{mn}: Diagonal matrix.
- V_{mn}^T: The transpose of an orthogonal matrix.

Figure 2.3 depicts the low rank reduction phase. The new matrix is the product of the other three, but reducted, this is a very relevant issue. If the singular values in S_{mn} are ordered by size, the first k largest may be kept and the remaining smaller ones set to zero. The product of the resulting matrices is a matrix X which is only approximately equal to A_{mm} , and is of rank k. It can be shown that the new matrix X is the matrix of rank k which is closest in the least squares sense to A_{mm}. The amount of dimension reduction, i.e., the choice of k, is critical to our work. Ideally, we want a value of k that is large enough to fit all the real structure in the data, but small enough so that we do not also fit the sampling error or unimportant details. The proper way to make such choices is an open issue in the factor analytic literature. In practice, we currently use an operational criterion - a value of k which yields good retrieval performance.

U_{mm} describes the original row entities as vectors of derived orthogonal factor values. S_{mn} represents the original column entities in the same way, and V_{mn} is a diagonal matrix containing scaling values. With the application of LSA it is possible to find the most relevant features and remove the least important ones by means of the reduced matrix U_{mm}. As a result, an equivalence of A_{mm} can be constructed using the most relevant features. LSA helps reveal the latent relationship among words as well as among passages which cannot be guaranteed by a simple term-document matrix. The similarity measurement by LSA reflects adequately human perception

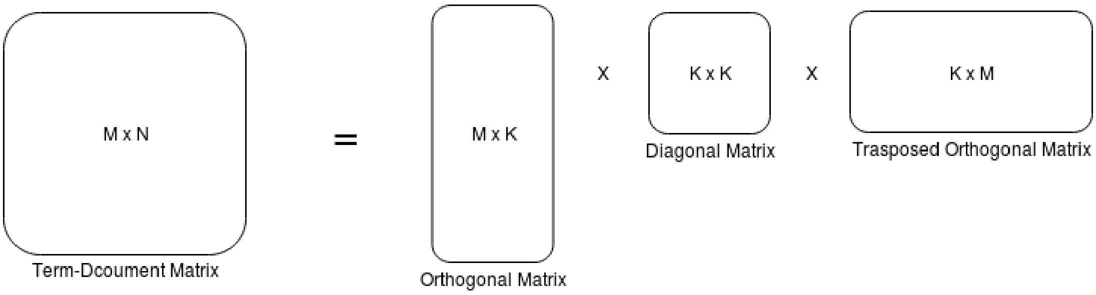

FIGURE 2.3: The decomposition phase

of similarity and association among texts. Using LSA, similarities among documents are measured as the cosine of the angle between their row vectors (see Section 2.2). LSA has been applied in [12],[20],[28] to compute similarities of software systems. The main disadvantage of LSA is that it is computational expensive when a large amount of information is analyzed.

To illustrate how LSA works, we take an example with a set of 9 documents as follows:

- **doc**$_1$: *Human machine interface for ABC computer applications.*
- **doc**$_2$: *A survey of user opinion of computer system response time.*
- **doc**$_3$: *The EPS user interface management system.*
- **doc**$_4$: *System and human system engineering testing of EPS.*
- **doc**$_5$: *Relation of user perceived response time to error measurement.*
- **doc**$_6$: *The generation of random, binary, ordered trees.*
- **doc**$_7$: *The intersection graph of paths in trees.*
- **doc**$_8$: *Graph minors IV: Widths of trees and well-quasi-ordering.*
- **doc**$_9$: *Graph minors: A survey.*

The term-document matrix for the document set is shown in Figure 2.4.

$$
\begin{array}{l}
\\
human \\ interface \\ computer \\ user \\ system \\ response \\ time \\ EPS \\ survey \\ trees \\ graph \\ minors
\end{array}
\begin{array}{c}
\begin{array}{ccccccccc} doc_1 & doc_2 & doc_3 & doc_4 & doc_5 & doc_6 & doc_7 & doc_8 & doc_9 \end{array} \\
\left(\begin{array}{ccccccccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 2 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{array}\right)
\end{array}
$$

FIGURE 2.4: Term-Document Matrix of the example

A computation exploiting an LSA implementation yields the matrices in Figures 2.5, 2.6, and 2.7.

Figure 2.8 depict the result of the decomposition with a rank of 2.

$$\begin{pmatrix} 0.22 & -0.11 & 0.29 & -0.41 & -0.11 & -0.34 & 0.52 & -0.06 & -0.41 \\ 0.20 & -0.07 & 0.14 & -0.55 & 0.28 & 0.50 & -0.07 & -0.01 & -0.11 \\ 0.24 & 0.04 & -0.16 & -0.59 & -0.11 & -0.25 & -0.30 & 0.06 & 0.49 \\ 0.40 & 0.06 & -0.34 & 0.10 & 0.33 & 0.38 & 0.00 & 0.00 & 0.01 \\ 0.64 & -0.17 & 0.36 & 0.33 & -0.16 & -0.21 & -0.17 & 0.03 & 0.27 \\ 0.27 & 0.11 & -0.43 & 0.07 & 0.08 & -0.17 & 0.28 & -0.02 & -0.05 \\ 0.27 & 0.11 & -0.43 & 0.07 & 0.08 & -0.17 & 0.28 & -0.02 & -0.05 \\ 0.30 & -0.14 & 0.33 & 0.19 & 0.11 & 0.27 & 0.03 & -0.02 & -0.17 \\ 0.21 & 0.27 & -0.18 & -0.03 & -0.54 & 0.08 & -0.47 & -0.04 & -0.58 \\ 0.01 & 0.49 & 0.23 & 0.03 & 0.59 & -0.39 & -0.29 & 0.25 & -0.23 \\ 0.04 & 0.62 & 0.22 & 0.00 & -0.07 & 0.11 & 0.16 & -0.68 & 0.23 \\ 0.03 & 0.45 & 0.14 & -0.01 & -0.30 & 0.28 & 0.34 & 0.68 & 0.18 \end{pmatrix}$$

FIGURE 2.5: Matrix U_{mm}x

$$\begin{pmatrix} 3.34 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.54 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.35 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.64 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.50 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.31 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.85 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.56 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.36 \end{pmatrix}$$

FIGURE 2.6: Matrix S_{mn}

$$\begin{pmatrix} 0.20 & 0.61 & 0.46 & 0.54 & 0.28 & 0.00 & 0.01 & 0.02 & 0.08 \\ -0.06 & 0.17 & -0.13 & -0.23 & 0.11 & 0.19 & 0.44 & 0.62 & 0.53 \\ 0.11 & -0.50 & 0.21 & 0.57 & -0.51 & 0.10 & 0.19 & 0.25 & 0.08 \\ -0.95 & -0.03 & 0.04 & 0.27 & 0.15 & 0.02 & 0.02 & 0.01 & -0.03 \\ 0.05 & -0.21 & 0.38 & -0.21 & 0.33 & 0.39 & 0.35 & 0.15 & -0.60 \\ -0.08 & -0.26 & 0.72 & -0.37 & 0.03 & -0.30 & -0.21 & 0.00 & 0.36 \\ 0.18 & -0.43 & -0.24 & 0.26 & 0.67 & -0.34 & -0.15 & 0.25 & 0.04 \\ -0.01 & 0.05 & 0.01 & -0.02 & -0.06 & 0.45 & -0.76 & 0.45 & -0.07 \\ -0.06 & 0.24 & 0.02 & -0.08 & -0.26 & -0.62 & 0.02 & 0.52 & -0.45 \end{pmatrix}$$

FIGURE 2.7: Matrix S_{mn}

2.4 Jaccard Index

Given two objects α and β represented by their corresponding set of elements $O(\alpha)$ and $O(\beta)$, the similarity is computed as the ratio of the cardinality of the intersection and the cardinality of the union of the two sets. The formula is given below:

$$Jaccard(\alpha,\beta) = \frac{|O(\alpha) \bigcap O(\beta)|}{|O(\alpha) \bigcup O(\beta)|} \tag{2.5}$$

The similarity using the Jaccard index is visualized in Figure 2.9. The eclipse on the left hand represents $O(\alpha)$ and the eclipse on the right hand represents $O(\beta)$. The intersection of the two sets is $O(\alpha) \bigcap O(\beta)$ and the larger it is, the closer to 1 is the Jaccard index. Once the two sets completely overlap each other, $Jaccard(\alpha,\beta)$ is equal to 1. Among the similarity tools presented in this thesis, *AnDarwin* [10] and

$$
\begin{array}{l}
 \\
human \\
interface \\
computer \\
user \\
system \\
response \\
time \\
EPS \\
survey \\
trees \\
graph \\
minors
\end{array}
\begin{array}{c}
\begin{array}{ccccccccc}
doc_1 & doc_2 & doc_3 & doc_4 & doc_5 & doc_6 & doc_7 & doc_8 & doc_9
\end{array} \\
\begin{pmatrix}
0.16 & 0.40 & 0.38 & 0.47 & 0.18 & -0.05 & -0.12 & -0.16 & -0.09 \\
0.14 & 0.37 & 0.33 & 0.40 & 0.16 & -0.03 & -0.07 & -0.10 & -0.04 \\
0.15 & 0.51 & 0.36 & 0.41 & 0.24 & 0.02 & 0.06 & 0.09 & 0.12 \\
0.26 & 0.84 & 0.61 & 0.70 & 0.39 & 0.03 & 0.08 & 0.12 & 0.19 \\
0.45 & 1.23 & 1.05 & 1.27 & 0.56 & -0.07 & -0.15 & -0.21 & -0.05 \\
0.16 & 0.58 & 0.38 & 0.42 & 0.28 & 0.06 & 0.13 & 0.19 & 0.22 \\
0.16 & 0.58 & 0.38 & 0.42 & 0.28 & 0.06 & 0.13 & 0.19 & 0.22 \\
0.22 & 0.55 & 0.51 & 0.63 & 0.24 & -0.07 & -0.14 & -0.20 & -0.11 \\
0.10 & 0.53 & 0.23 & 0.21 & 0.27 & 0.14 & 0.31 & 0.44 & 0.42 \\
-0.06 & 0.23 & -0.14 & -0.27 & 0.14 & 0.24 & 0.55 & 0.77 & 0.66 \\
-0.06 & 0.34 & -0.15 & -0.30 & 0.20 & 0.31 & 0.69 & 0.98 & 0.85 \\
-0.04 & 0.25 & -0.10 & -0.21 & 0.15 & 0.22 & 0.50 & 0.71 & 0.62
\end{pmatrix}
\end{array}
$$

FIGURE 2.8: The recovered matrix

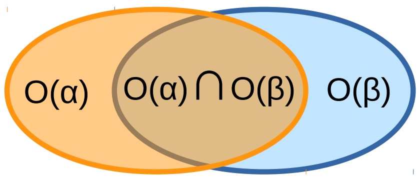

FIGURE 2.9: Jaccard similarity between two sets $O(\alpha)$ and $O(\beta)$

RepoPal [54] employ Jaccard index in their implementation.

2.5 Graph Similarity

Graph similarity is an active research field and receives a significant attention from the research community. In this section, we are going to review the approaches for computing similarity in graph that are beneficial to our context. A directed graph is defined as a tuple $G = (V, E, R)$, where V is the set of vertices, E is the set of edges and R represents the relationship among the nodes [6]. A graph consists of enormous nodes and oriented links with semantic relationships. A triple $<subject, predicate, object>$ with $subject, object \in V$ and $predicate \in E$ states that node *subject* is connected to node *object* by means of the edge labelled with *predicate*. To evaluate the similarity of two nodes in a graph, their intrinsic characteristics like nodes, links, and their mutual interactions are incorporated into the similarity calculation [11],[34],[33]. Among others, feature-based semantic similarity metrics gauge the similarity between graph nodes as a measure of commonality and distinction of their hallmarks.

Tversky provides a deep insight into feature-based similarity in his work [50]. There, objects are represented as a set of common and distinctive features and the similarity between two objects is computed by comparing their features. An object is represented in one of the following forms: *binary values*, *nominal values*, *ordinal values*, and *cardinal values*. Feature-based semantic similarity metrics first attempt to characterize resources in a graph as sets of feature and then perform similarity calculation on them.

SimRank has been designed to calculate similarity based on the mutual relationships between nodes [17]. In a graph, the similarity between two nodes is dependent on their neighbors. Considering two nodes, the more similar nodes point to them, the more similar the two nodes are. For example, in Figure 2.10, the two nodes α

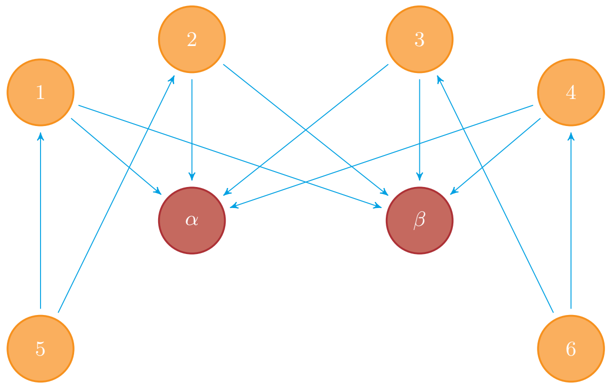

FIGURE 2.10: SimRank similarity

and β are highly similar because they are concurrently pointed by other four nodes in the graph. Also, node 1 is similar to node 2 since both are pointed by node 5. Comparably, the similarity between node 3 and node 4 is high as they are pointed by node 6. In this sense, the similarity between two nodes α and β is computed by using a fixed-point function. Given $k \geq 0$ we have $R^{(k)}(\alpha, \beta) = 1$ with $\alpha = \beta$ and $R^{(k)}(\alpha, \beta) = 0$ with $k = 0$ and $\alpha \neq \beta$. In all the other cases the general formula is:

$$R^{(k+1)}(\alpha, \beta) = \frac{\Delta}{|I(\alpha)| \cdot |I(\beta)|} \sum_{i=1}^{|I(\alpha)|} \sum_{j=1}^{|I(\beta)|} R^{(k)}(I_i(\alpha), I_j(\beta)) \tag{2.6}$$

where Δ is a damping factor ($0 \leq \Delta < 1$); $I(\alpha)$ and $I(\beta)$ are the set of incoming neighbors of α and β, respectively. $|I(\alpha)| \cdot |I(\beta)|$ is the factor used to normalize the sum, thus making $R^{(k)}(\alpha, \beta) \in [0, 1]$. Equation 2.6 implies that the similarity for two nodes is computed by aggregating the similarity of all possible pairs of their neighbors.

SimRank has been used by CROSSSIM [35] as the mechanism to compute the similarity among nodes in a graph representing the OSS ecosystem.

Chapter 3

Literature Review on Software Similarity Measurement

The ability to search for similar software projects with respect to different criteria such as functionalities and dependencies plays an important role in the development process. Two projects are deemed to be similar if they implement some features being described by the same abstraction, even though they may contain various functionalities for different domains [28]. Understanding the similarities between open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations [46],[54], thereby improving software quality. Meanwhile measuring the similarities between developers and software projects is a critical phase for most types of recommender systems [37],[44]. Similarities are used as a base by both content-based and collaborative-filtering recommender systems to choose the most suitable and meaningful items for a given item [46]. Failing to compute precise similarities means concurrently adding a decline in the overall performance of these systems. Nevertheless, measuring similarities between software systems has been considered as a daunting task [8],[28]. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail.

In recent years, several approaches have been proposed to solve the problem of software similarity computation. In this chapter, we review some of the most notable approaches which have been conceived to measure the similarity between software systems or OSS projects. Afterwards, in Section 3.12 we analyze their characteristics. According to [8], depending on the set of mined features, there are two main types of software similarity computation techniques:

- *Low-level Similarity*: it is calculated by considering low-level data, e.g., source code, byte code, function calls, API reference, etc.;
- *High-level Similarity*: detecting the semantic similarity using metadata, such as: topic distribution, readme file, description, star events, etc. Source code is not taken into account.

This classification is used throughout this paper as a means to distinguish between the approaches with regards to the input information used for similarity computation. In particular, we review the following categories of software similarity:

- Detecting similar open source applications (see Sections 3.1, 3.2, 3.3, and 3.10).
- Detecting similar mobile applications (see Sections 3.4, 3.5, and 3.6).
- Detecting software plagiarisms and clones (see Sections 3.7 and 3.8).

- Recommending reusable libraries (see Section 3.9).

3.1 MUDABlue: Automatic Categorization for Open Source Repositories

Together with a tool for automatically categorizing open source repositories, the authors in [12] propose an approach for computing similarity between software projects using source code. A pre-processing stage is performed to extract identifiers such as variable names, function names and to remove unrelated factors such as comment.

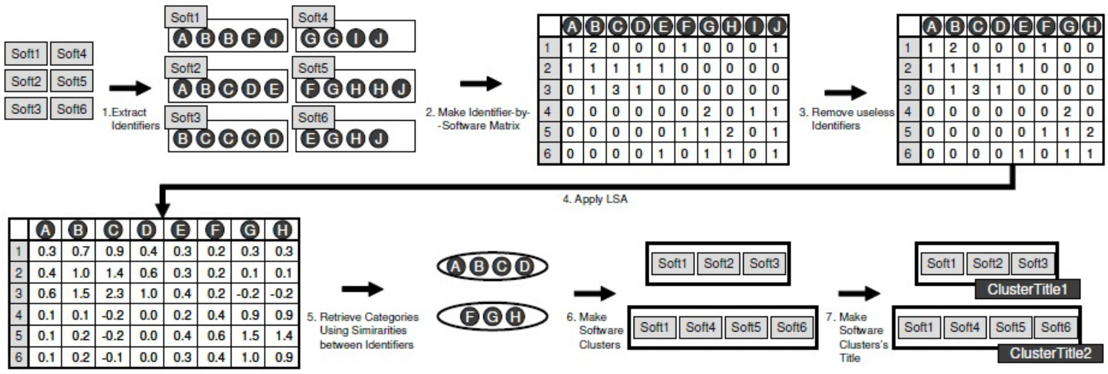

FIGURE 3.1: The MUDABlue working phases [12]

With the application of Latent Semantic Analysis, software is considered as a document and each identifier is a word. LSA is used for extracting and representing the contextual usage meaning of words by statistical computations applied to a large corpus of text. In summary, the process that MUDABLUE applies is depicted in Figure 3.1 and consists of the following steps [12]:

i) Extracts identifiers from source code and removes unrelated content;

ii) Creates an identifier-software matrix with each row corresponds to one identifier and each column corresponds to a software system;

iii) Removes unimportant identifiers, i.e. those that are too rare or too popular;

iv) Performs LSA on the identifier-software matrix and computes similarity on the reduced matrix using cosine similarity;

MUDABlue has been evaluated on a database consisting of software systems written in C [12]. The outcomes of the evaluation were compared against two existing approaches, namely GURU [25], and the SVM based method by *Ugurel et al* [51]. The evaluation shows that MUDABlue outperforms these observed algorithms with regards to precision and recall.

3.2 CLAN: Finding Related Applications

CLAN (Closely reLated ApplicatioNs) [28] is an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to packages class hierarchies. CLAN works based on the document framework for computing similarity, semantic anchors, e.g. those that define the documents' semantic features. Semantic anchors and dependencies help obtain a more precise value for similarity computation between documents. The assumption is that if two applications have API calls implementing requirements described by the same abstraction, then the two applications are more similar than those that do not have common API calls. The approach uses API calls as semantic anchors to compute application similarity since API calls contain precisely defined semantics. The similarity between applications is computed by matching the semantics already expressed in the API calls. The working process applied by CLAN is shown in Figure 3.2.

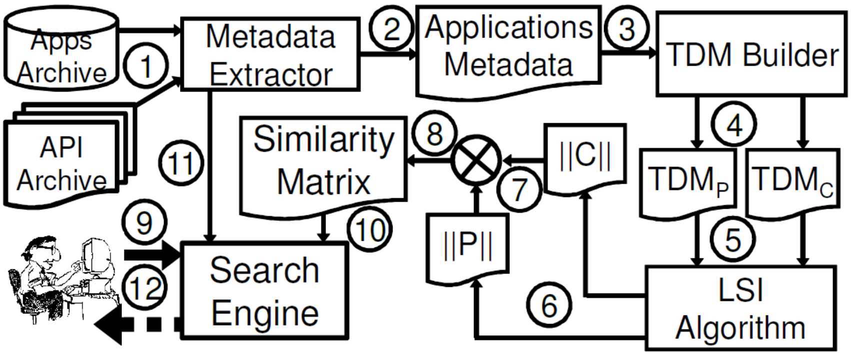

FIGURE 3.2: The CLAN working phases [28]

Using a complete software application as input, CLAN represents source code files as a TDM, in which a row contains a unique class or package and a column corresponds to an application. SVD is then applied to reduce the dimension of the matrix. Similarity between applications is computed as the cosine similarity between vector in the reduced matrix. CLAN has been tested on a dataset with more than 8.000 SourceForge[1] applications and shows that it qualifies for the detection of similar applications [28].

MUDABlue and CLAN are comparable in the way they represent software and source code components like variables, function names or API calls in a term-document matrix and then apply LSA to find the similarity and to category the softwares. However, CLAN has been claimed to help obtain a higher precision than that of MUDABlue as it considers only API calls to represent software systems.

3.3 TagSim: Collaborative Tagging to Detect Similar Applications

In [23] tags are leveraged to characterize applications and then to compute similarity between them. Tags are terms that are used to highlight the most important characteristics of software systems [53] and therefore, they help users narrow down the search scope. Some examples of tags are the category of an app, the license of the system, the programming languages. TagSim[2] can be used to detect similar applications written in different languages. Based on the assumption that tags capture better the intrinsic features of applications compared to textual descriptions, TagSim

[1]SourceForge: https://sourceforge.net/

[2]For the sake of clarity, in this thesis we give a name for the algorithms that have not been named originally

extracts tags attached to an application and computes their weights. This information forms the features of a given software system and can be used to distinguish it from others. The technique also differentiates between important tags and unimportant based on their frequency of appearance in the analyzed software systems. The more popular a tag across the applications is, the less important it is and vice versa, i.e. the weight of tag t is $w(t) = \frac{1}{|App(t)|}$ where $App(t)$ is the set of applications that have t as tag. Each application is characterized by a feature vector, $\vec{Tag}(a)$, with each entry corresponds to the weight of a tag the application has. Eventually, the similarity between two applications is computed as the cosine similarity between the two vectors:

$$sim(a_1, a_2) = CosineSim(\vec{Tag}(a_1), \vec{Tag}(a_2)) \tag{3.1}$$

To evaluate TagSim, the authors in [23] collected and analyzed more than a hundred thousands of projects. A total of 20 queries were used to study the performance of the algorithm in comparison with CLAN. The authors also performed a user study to manually analyze the extent to which two applications are similar. Afterwards, success rate, confidence, and precision were used as evaluation metrics. The experimental results show that TagSim helps achieve better performance in comparison to CLAN with a success rate of 80%.

Similarly, to help users category a new software object, *TagCombine* has been proposed in [53]. *TagCombine* works using three components: a multi-label ranking component, a similarity based ranking component, and a tag-term based ranking component. In this approach, tags are also used to represent a piece of software as a feature vector, and finally to compute similarity. *TagCombine* has been evaluated against the approach proposed by *Al-Kofahi et. al.* in [3] using datasets collected from 2 popular software information sites, StackOverflow[3] and Freecode[4] [53]. Experiment results show that *TagCombine* gains a better performance compared to the approach presented in [3].

3.4 CLANdroid: Detecting Similar Android Applications

Inspired by CLAN, CLANdroid was developed for detecting similar Android applications with the assumption that similar apps share some semantic anchors [20]. Nevertheless, in contrast to CLAN, CLANdroid works also when source code is not available as it exploits other high-level information. By extending the scope of semantic anchors for Android apps, starting from APK (Android Package) CLANdroid extracts quintuple features, i.e. identifiers, intents from source code, API calls and sensors from JAR files and user permissions from the *AndroidManifest.xml*[5] specification. This file is a mandatory component for an Android app and it contains important information about it. For each feature, a feature-Application Matrix is built, resulting in five different matrices. Latent Semantic Indexing is applied to all the matrices to reduce the dimensionality. Afterwards, similarity between a pair of applications is computed as the cosine similarity between their corresponding feature vectors from the matrix. Users can query for similar apps with a given app by specifying which feature is taken into consideration.

[3]StackOverflow: `https://stackoverflow.com/`
[4]Freecode: `https://www.freecodecamp.org/`
[5]`https://developer.android.com/guide/topics/manifest/manifest-intro.html`

Evaluations have been performed in [20] to study which semantic anchors are more effective. The authors also analyze the impact of third-party libraries and obfuscated code when detecting similar apps, since these two factors have been shown to have significant impact on reuse in Android apps and experiments using APKs. The evaluation on a dataset shows that computing similarity based on API helps produce higher recall. According to the experimental results, the feature sensor is ineffective in computing similarity. By comparing with a ground-truth dataset collecting from Google Play, the study suggests the mechanism behind the way Google Play recommends similar apps.

3.5 SimApp: Detecting Similar Mobile Apps by Online Kernel Learning

With the aim of finding apps with similar semantic requirements, SimApp has been proposed in [8]. Unlike other approaches that exploit low-level implementation, e.g. source code, API utilization for similarity calculation, SimApp makes use of high-level metadata collected from apps markets for detecting similar mobile applications. By SimApp, if two apps implement related semantic requirements then they are seen as similar. Each mobile application is modeled by a set of features, so called *modalities* . The following features are incorporated into similarity computation: *Name*, *Category*, *Developer*, *Description*, *Update*, *Permissions*, *Images*, *Content rating*, *Size* and *Reviews*. For each of these features, a function is derived for each of the features to calculate the similarity between applications.

Given a pair of apps (a_i, a_j), a kernel function is defined to compute the similarity for each feature as follows:

- *Name*: It is supposed that two apps are similar if they share common words in their name. A string kernel is exploited to compute the similarity between two app names.

- *Category*: Apps in the same category are more similar to each other than to apps in different categories.

- *Developer*: Each developer is characterized by the set of apps that she is involved in and the similarity between two apps is computed using a kernel function of their corresponding developer vectors.

- *Description*: The description text of an app is considered as a document and a kernel function is used to compute the similarity between two description documents of a_i and a_j.

- *Update*: Developers use update text to describe the changes they made to the new version of the app. Each update is converted to a fixed length vector. The similarity between a_i and a_j based on update is computed by using a kernel function similar to the one used for Description.

- *Permission*: For each app, there is a list of permissions specifying which resources on the phone the app can use. A feature vector is used to characterize the permissions of an app and the similarity between a_i and a_j with regards to permission is computed using a kernel function.

- *Images*: Each app is normally attached with a screenshot image. And SimApp considers two app as similar if they have similar screenshot images. In this way a kernel function is exploited to compute the similarity between two images.

- *Content rating*: Each app has content rating to describe its content and age appropriateness.
- *Size*: It is supposed that two apps whose size is considerably different cannot be similar.
- *User review*: All user reviews for an app is combined in a document and a similar process for other textual contents is applied to compute the similarity between a_i and a_j.

For example, the kernel function for measuring similarity between apps a_i and a_j with names s_i and s_j is as follows:

$$K^{name}(a_i, a_j) = \sum_{u_k \in \Sigma^*} \phi_u(s_i)\phi_u(s_j) \tag{3.2}$$

This kernel function is also applied to other textual contents, i.e. Name, Description, Update, Reviews to compute similarities among apps with regards to these modalities.

The final similarity score for a pair of apps (a_i, a_j) is a linear combination of the multiple kernels with weights. Through the use of a set of training data, the optimal weights are determined by means of online learning techniques.

$$K(a_i, a_j; w) = \sum_{k=1}^{n} w_k K^k(a_i, a_j) \tag{3.3}$$

3.6 AnDarwin: Detecting Similar Android Applications

AnDarwin is an approach that applies Program Dependence Graphs to represent apps [10]. Feature vectors are then clustered to find similar apps. Locality Sensitive Hashing is used to find approximate near-neighbors from a large number of vectors. AnDarwin works in the following stages:

i) It represents each app as a set of vectors computed over the app's Program Dependence Graphs;

ii) Similar code segments are found by clustering all the vectors of all apps;

iii) It eliminates library code based on the frequency of the clusters;

iv) Finally, it detects apps that are similar, considering both full and partial app similarity.

AnDarwin has been applied to find similar apps by different developers (cloned apps) and groups of apps by the same developer with high code reuse (rebranded apps).

3.7 GPLAG: Using Graph for Detecting Software Plagiarism

GPLAG is an approach for detecting software plagiarism using program dependence graph [22]. Using different input information, GPLAG represents source code files as a graph and detect plagiarism by identifying similar graph patterns. The algorithm captures the control flow and data dependencies between the code statement inside code fragments.

A program dependence graph (PDG) is a labelled, directed graph that uses variable declarations, variable assignments, procedure calls to represent the data and control dependencies within one source code procedure. Code statements are represented by vertices and the dependencies of data and control between statements are edges. A PDG represents the data flow between statements as well as the control between statements. Using the representation, PDG encodes the program logic, thereby representing developers' intention. Given an original program P_O, and a plagiarism suspect P_S, plagiarism detection tries to search for duplicate structures. Graph isomorphism is performed to compute the similarity between the PDGs to detect whether two procedures are similar or not.

3.8 WuKong: Detecting Cloned Android Apps

WuKong is a proposed approach to detect Android apps clone [52]. It is based on a two-phase process which first exploits the frequency of Android API calls to filter out external libraries. Afterwards, a fine-grained phase is performed to compare more features on the set of apps coming from the first phase. For each variable, its feature vector is formed by counting the number of occurrence of variables in different contexts (Counting Environments - CE). An m-dimensional Characteristic Vector (CV) is generated using m CEs, where the i-th dimension of the CV is the number of occurrences of the variable in the i-th CE. For each code segment, CVs for all variables are computed. A code segment is represented by an $n \times m$ Characteristic Matrix (CM). For each app, all code segments are modelled using CM, yielding a series of CMs and they are considered as the features for the app. The similarity between two apps is computed as the proportion of similar code segments. The similarity between two variables v_1 and v_2 is computed using cosine similarity between their feature vectors $\vec{V_1}$ and $\vec{V_2}$:

$$sim(v_1, v_2) = CosineSim(\vec{V_1}, \vec{V_2}) \tag{3.4}$$

Evaluations on more than $100,000$ Android apps collected from 5 Chinese app markets show that the approach can effectively detect cloned apps [52].

3.9 LibRec: Automated Library Recommendation

To help developers leverage existing libraries, LibRec is proposed to provide them with library recommendations [48]. LibRec suggests the inclusion of libraries that may be useful for a given project using a combination of rule mining and collaborative filtering techniques. It finds a set of relevant libraries, based on the current set of libraries that a project already uses. Association rule mining is applied to find similar libraries that co-exist in many projects. A collaborative filtering technique is applied to search for top most similar projects and recommends libraries used by these projects to a given project [48].

- *Association rule*: the common co-occurrence of libraries in an application. The association rule mining component extracts libraries that are commonly used together. The component then rates each of the libraries based on their likelihood to appear together with the currently used libraries.

- *Collaborative Filtering*: Given a project, similarity is computed against all projects and top similar projects are selected. The libraries used by the top similar projects are used as recommendations based on a score computed according to their popularity.

Considering a set of projects $R = (p_1, p_2, ...p_m)$ and a set of libraries $L = (l_1, l_2, ...l_n)$, each project is characterized by a feature vector using the set of libraries it includes, i.e. $\vec{P_i} = (I_i(l_1), I_i(l_2), ..I_i(l_n))$, where $I_i(l_r)$ is the inclusion of library l_r in project p_i. $I_i(l_r) = 1$ if l_r is used in p_i, otherwise $I_i(l_r) = 0$. The similarity between two projects is the cosine similarity between their feature vectors as follows:

$$sim(p_i, p_j) = CosineSim(\vec{P_i}, \vec{P_j}) \tag{3.5}$$

Ten-fold cross validation is applied on a dataset of 500 GitHub projects that use at least 10 third-party libraries to evaluate the performance of LibRec [48]. The dataset is divided into 10 equal parts, so-called *sub-samples*. The validation was conducted for ten times and for each time, nine sub-samples are used as training data and the remaining sub-sample is used as test data. For each testing project, a half of its libraries is taken out and used as ground-truth data and the other half is used to compute the similarities to all projects in the training set to get library recommendation. The experiments show that the libraries recommended by LibRec match the ones that are already stored in the ground-truth data with high recall rate.

3.10 RepoPal: A tool to detect similar GitHub projects

In contrast to many previous studies that are generally based on source code [12],[22],[28], RepoPal [54] is a high-level similarity metric and takes only repositories metadata as its input. With this approach, two GitHub[6] repositories are considered to be similar if:

i) They contain similar `README.MD` files;

ii) They are starred by users of similar interests;

iii) They are starred together by the same users within a short period of time.

Thus, the similarities between GitHub repositories are computed by using three inputs: readme file, stars and the time gap that a user stars two repositories. Considering two repositories r_i and r_j, the following notations are defined:

- f_i and f_j are the readme files with t being the set of terms in the files;
- $U(r_i)$ and $U(r_j)$ are the set of users who starred r_i and r_j, respectively;
- $R(u_k)$ is the set of repositories that user u_k already starred.

There are three similarity indices as follows:

[6]About GitHub: https://github.com/about

Readme-based similarity The similarity between two readme files is calculated as the cosine similarity between their feature vectors $\vec{f}_i$ and $\vec{f}_j$:

$$sim_f(r_i, r_j) = CosineSim(\vec{f}_i, \vec{f}_j) \tag{3.6}$$

Stargazer-based similarity The similarity between a pair of users u_k and u_l is defined as the Jaccard index [16] of the sets of repositories that u_k and u_l have already starred:

$$sim_u(u_k, u_l) = Jaccard(R(u_k), R(u_l)) \tag{3.7}$$

The star-based similarity between two repositories r_i and r_j is the average similarity score of all pairs of users who already starred r_i and r_j:

$$sim_s(r_i, r_j) = \frac{1}{|U(r_i)| \cdot |U(r_j)|} \sum_{\substack{u_k \in U(r_i) \\ u_l \in U(r_j)}} sim_u(u_k, u_l) \tag{3.8}$$

Time-based similarity It is supposed that if a user stars two repositories during a relative short period of time, then the two repositories are considered to be similar. Based on this assumption, given that $T(u_k, r_i, r_j)$ is the time gap that user u_k stars repositories r_i and r_j, the time-based similarity is computed as follows:

$$sim_t(r_i, r_j) = \frac{1}{|U(r_i) \cap U(r_j)|} \sum_{u_k \in U(r_i) \cap U(r_j)} \frac{1}{|T(u_k, r_i, r_j)|} \tag{3.9}$$

Finally, the similarity between two projects is the product of the three similarity indices:

$$sim(r_i, r_j) = sim_f(r_i, r_j) \times sim_s(r_i, r_j) \times sim_t(r_i, r_j) \tag{3.10}$$

RepoPal has been evaluated against CLAN using a dataset of 1,000 Java repositories [54]. Among them, 50 were chosen as queries. *Success Rate*, *Confidence* and *Precision* were used as the evaluation metrics. Experimental results in the paper show that RepoPal produces better quality metrics than those of CLAN.

3.11 CROSSSIM: Detecting similar OSS projects using graph

In recent years, considerable effort has been made to provide automated assistance to developers in navigating large information spaces and giving recommendations. Though remarkable progress can be seen in this field, there is still room for improvement. To the best of our knowledge, most of the existing approaches consider the constituent components of the OSS ecosystem separately, without paying much attention to their mutual connections. There is a lack of a proper scheme that facilitates a unified consideration of various OSS artifacts and recommendations. CROSSSIM [35],[31],[32] has been proposed as a novel approach to compute similarity.

The architecture of CrossSim is depicted in Figure 3.3: the rectangles represent artifacts, whereas the ovals represent activities that are automatically performed by the developed CrossSim tooling. In particular, the approach imports project data from existing OSS repositories and represents them into a graph-based representation by means of the *OSS Ecosystem Representation* module. Depending on the considered repository (and thus to the information that is available for each project) the

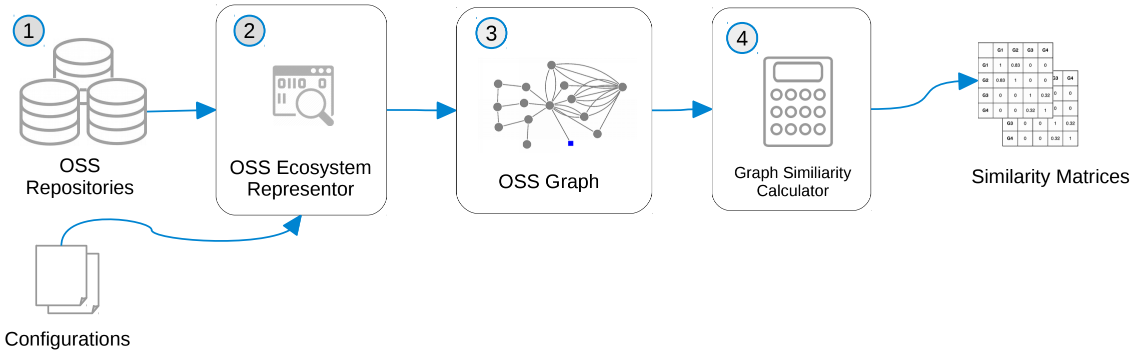

FIGURE 3.3: The CROSSSIM Architecture

graph structure to be generated has to be properly configured. For instance in case of GitHub, specific configurations have to be specified in order to enable the representation in the target graphs of the stars assigned to each project. Such a configuration is "forge" specific and specified once, e.g., SourceForge does not provide the star based system available in GitHub. The *Graph similarity* module implements the SimRank algorithm [17] that is applied on the source graph-based representation of the input ecosystems generates matrices representing the similarity value for each pair of input projects.

We consider the community of developers together with OSS projects, libraries and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependency and implication on the others. There, several connections and interactions prevail, such as developers commit to repositories, users star repositories, or projects contain source code files, just to name a few. We propose a solution that makes use of graphs for representing relationships in OSS ecosystems. Specifically, the graph model has been chosen since it allows for flexible data integration and facilitates numerous similarity metrics and clustering techniques [7],[24],[45]. All the playing actors and their communications are transformed into a directed graph. Humans and non-human artifacts are represented as nodes and there is a directed edge between a pair of nodes if they interact with each others. The representation model considers different artifacts in a united fashion, taking into account their mutual, both direct and indirect relationships as well as their co-occurrence as a whole. The representation is twofold: First, it incorporates semantic relationships into the graph. Second, it helps combine both low-level and high-level information into a homogeneous representation.

To demonstrate the utilization of graphs in an OSS ecosystem, we consider an excerpt of the dependencies for two OSS projects, namely `project#1` and `project#2` in Figure 3.4. Using dependency information extracted from source code and the corresponding metadata (e.g. coming from the tools developed in by Work Package 2), this graph can be properly built to represent the two projects as a whole. In this figure, `project#1` contains code file `HttpSocket.java` and `project#2` contains `FtpSocket.java` with the corresponding edges being marked with the semantic predicate `hasSourceCode`. Both source code files implement `interface#1` being marked by the semantic predicate `implements`. `Project#1` and `project#2` are also connected via other semantic paths, such as API `isUsedBy` highlighted in Figure 3.5. In practice, an OSS graph is much larger with numerous nodes and edges, and the relationship between two projects can be thought as a sub-graph.

Based on the graph structure, one can exploit nodes, links and the mutual relationships to compute similarity using existing graph similarity algorithms. To the

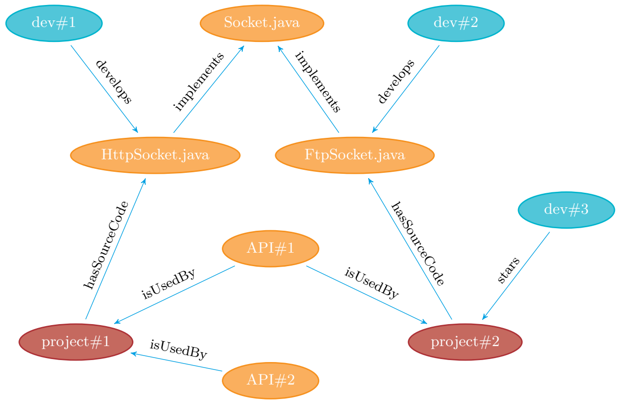

FIGURE 3.4: Sample graph-based representation of OSS ecosystems

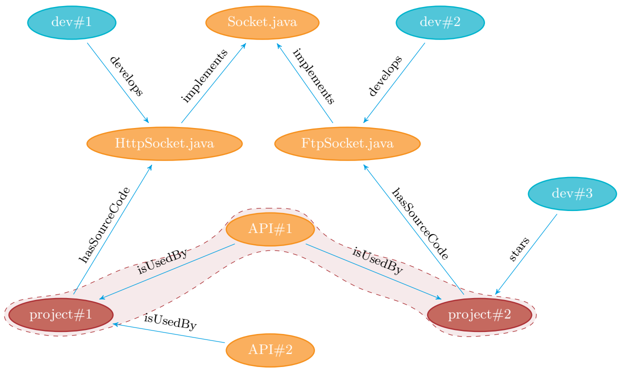

FIGURE 3.5: Similarity between OSS projects with respect to API usage

best of our knowledge, there exist several metrics for computing similarity in graph [7],[34],[33]. The graph structure also allows for graph kernel methods, which are an effective way to compute similarity [38]. Considering Figure 3.4, we can compute the similarity between `project#1` and `project#2` with regards to the semantic paths between them, e.g. the two-hop path using `hasSourceCode` and `implements` (Figure 3.6), or the one-hop path using API `isUsedBy`. For example, concerning `isUsedBy`, the two projects are considered to be similar since with the predicate both projects originate from `API#1`. The hypothesis is based on the fact that the projects are aiming at creating common functionalities by using common libraries [28],[48].

The representation allows us to compute similarity between other graph components, e.g. developers. Back to Figure 3.4, though there is no direct connection between `dev#1` and `dev#2`, their similarity can still be inferred from indirect semantic paths, such as `develops` and `implements` which are highlighted in Figure **??**. If we consider other semantic paths, we see that the two developers have more in common as they both take part in `project#1` and `projects#2` represented by `commits`. To a certain extent, the two developers are considered to be similar, although they are not directly connected. In reality, the connection between `developer#1` and `developer#2` is enforced by further semantic paths and as a result their similarity can be more precisely computed. The similarities between developers can serve as input for a collaborative filtering recommendation system, with which a developer is recommended a list of projects or libraries that similar developers already worked with

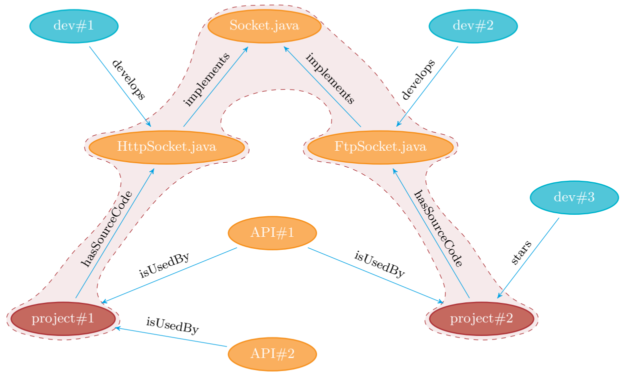

FIGURE 3.6: Similarity between OSS projects with respect to source implementation

[39],[46]. This is an invaluable tool in the context of the CROSSMINER project since it is essential to equip developers with recommendation functionalities to help them increase reusability and productivity.

The relationships underpinning the graph-based representations of the simple ecosystems shown in Fig. 3.4–3.5 are shown in Table 3.1. For the first implementation of CROSSSIM the relationships *isUsedBy*, *develops*, and *stars* have been considered.

3.12 Analysis

In this section we present a review on the above mentioned similarity metrics. The review is twofold as follows. First, it helps determine the features that contribute effectively towards similarity computation in OSS projects. Second, it aims at evaluating and identifying the strength as well as the shortcomings of the approaches. Table 3.2 shows a summary of the previously outlined approaches with respect to the features, which are exploited by each technique. In particular, the features shown in Table 3.2 are:

- *LOC*: the number of lines of code.
- *Dep.*: the third-party libraries a project includes.
- *API calls*: API function calls appear in source code. They are used to build term-document matrices and then to calculate similarities among applications.
- *Function*: functions and procedures defined in the source code.
- *Star*: star events in GitHub. Different from the concept of stars or bubbles used in other rating systems like TripAdvisor[7] or Facebook[8], stars in GitHub are not used to rate a repository. A developer stars a repository as a way to keep track of it for future reference. In addition, stars are used as a means to thank the repository maintainers for their contribution[9].
- *Timestamp*: the point of time when a user stars a repository.
- *Statement*: source code statement.

[7] https://www.tripadvisor.com/TripAdvisorInsights/n2640/all-about-your-tripadvisor-bubble-rating
[8] https://www.facebook.com/help/548274415377576/
[9] https://help.github.com/articles/about-stars

Relationship	Description
$isUsedBy \subseteq Dependency \times Project$	this relationship depicts the reliance of a project on a dependency (e.g., a third-party library). The project needs to include the dependency in order to function. According to [28, 48] the similarity between two considered projects relies on the dependencies they have in common because they aim at implementing similar functionalities.
$develops \subseteq Developer \times Project$	we suppose that there is a certain level of similarity between two projects if they are built by same developers, as already hypothesized by [8]. Thus, this relationship is used to represent the projects that a given user contributes in terms of source code development.
$stars \subseteq User \times Project$	This relationship is inspired by the star event in RepoPal [54] to represent GitHub projects that a given user has starred. However, we consider the star event in a broader scope in the sense that not only direct but also indirect connections between two developers is taken into account.
$implements \subseteq File \times File$	It represents a specific relation that can occur between the source code given in two different files, e.g. a class specified in one file implementing an interface given in another file.
$hasSourceCode \subseteq Project \times File$	It represents the source files contained in a given project.

TABLE 3.1: Graph Representation of OSS projects

- *Readme*: Readme.md or description file, used to describe the functionalities of an open source project.
- *Tag*: tags are used by OSS platforms, e.g. SourceForge to classify and characterize an open source project.
- *Update*: the newest changes made to the app.
- *Permission*: this feature is available only by mobile apps. It specifies the permission of an app to handle data in a smartphone.
- *Screenshot*: this feature is available by mobile apps. It is used to compare different apps.

As shown in Table 3.2 the techniques that mainly underpin the outlined similarity approaches are:

- *TDM & LSA*: Term-Document Matrix [9] and Latent Semantic Analysis [19] are generally used in combination to model the relationships between API calls/identifiers and software systems and to compute the similarities between them.

- *COS*: Cosine Similarity, this technique is widely used in several algorithms for computing similarities among vectors.
- *JCS*: Jaccard index used for computing similarity between two sets of elements [16].

Most low-level similarity algorithms (shown as *L* in Table 3.2) attempt to represent source code (and API calls) in a term-document matrix and then apply SVD to reduce dimensionality. The similarity is then computed as the cosine similarity between feature vectors. Among others, MUDABlue [12], CLAN [28], and CLANdroid [20] belong to this category. CLAN includes API calls for computing similarity, whereas, by MUDABlue, every word appearing in source code files is integrated into the term-document matrix. This makes the difference in the performance of the two algorithms in a way that the similarity scores of CLAN reflect better the perception of humans of similarity than those of MUDABlue.

	MUDABlue	**CLAN**	**CLANdroid**	**GPLAG**	**LibRec**	**SimApp**	**AnDarwin**	**WuKong**	**TagSim**	**RepoPal**	**CrossSim**
References	[12]	[28]	[20]	[22]	[48]	[8]	[10]	[52]	[23]	[54]	[35]
Features (Modalities)											
LOC				$\times$							$\times$
Dep.		$\times$									$\times$
API Calls		$\times$	$\times$		$\times$			$\times$			$\times$
Function				$\times$							$\times$
Star										$\times$	$\times$
Timestamp										$\times$	
Statement	$\times$			$\times$							
Identifier	$\times$		$\times$								
App.Name						$\times$					
Topic						$\times$					
Developer						$\times$					$\times$
Readme.md					$\times$		$\times$		$\times$	$\times$	
Tag									$\times$		
Update						$\times$					
Permissons			$\times$			$\times$					
Screenshot						$\times$					
Content						$\times$					
Size						$\times$					
Reviews						$\times$					
Intent			$\times$								
Sensors			$\times$								
Used Techniques											
TDM&LSA	$\times$	$\times$	$\times$								
COS	$\times$	$\times$	$\times$		$\times$	$\times$		$\times$	$\times$	$\times$	
JCS							$\times$			$\times$	
Category											
High/Low Sim	L	L	L	L	L	H	H	L	H	H	L & H

TABLE 3.2: Summary of the similarity algorithms and their features

In contrast, high-level similarity techniques (shown as *H* in Table 3.2) do not consider source code for similarity computation. They characterize software by exploiting available features such as descriptions, user reviews, and `README.MD` file. The similarity is computed as the cosine similarity of the corresponding feature vectors. For computing similarity between mobile applications, other specific features such as images and permissions are also incorporated. A current trend in these techniques is to exploit textual content to compute similarity, e.g. in AppRec [5], SimApp [8], TagSim [23]. A main drawback with this approach is that, same words can be used to explain different requirements or the other way around, the same requirements can be described using different words [12]. So it might be the case that two textual contents with different vocabularies still have a similar description or two files with similar vocabularies contain different descriptions. The matching of words in the descriptions as well as source code to compute similarity is considered to be ineffective as already stated in [28]. To overcome this problem, the application of a synonym dictionary like WordNet [30] is beneficial. Furthermore, the utilization TDM and LSA in textual contents is proven to be effective as LSA helps consider latent semantic relationships. Nevertheless, there is still a problem with the approaches like RepoPal where readme file is used for similarity computation, since in general the descriptions for software projects are written in different languages. According to our observation in GitHub, `README.MD` files are written in various languages, e.g., not only English but also Japanese, Korean, or Chinese. And the comparison of a readme file in Japanese with one in English should yield dissimilarity, even though two projects may be similar. SimApp [8] is the only technique that attempts to combine several high-level information into similarity computation. It eventually applies a machine learning algorithm to learn optimal weights. The approach is promising, nevertheless it is only applicable in the presence of a decent training dataset, which is hard to come by in practice.

CROSSSIM is an approach that attempts to combine both low-level and high-level information in computing similarities. The approach integrates implicit semantic relationships and intrinsic dependencies among different users, repositories, source code. Thus, it is able to incorporate new features, on the fly, into the similarity computation without modifying the internal design. CROSSSIM is expected to improve the overall performance of the similarity computation and thus the quality of the eventual recommendations. In the next chapters, we are going to introduce our implementation and evaluation of CROSSSIM with respect to some notable tools, i.e., MUDABLUE, CLAN, and REPOPAL.

Chapter 4

Implementation

We select MUDABLUE, CLAN, REPOPAL and CROSSSIM as the tools to be investigated in this chapter. The rationale behind the selection of these tools for comparison is that they are well-established approaches for detecting similar OSS projects. According to *Zhang et al.* [54], by applying the same experiment settings and evaluating on the same dataset, the authors demonstrated that REPOPAL outperforms CLAN with respect to *Confidence* and *Precision*. Whereas CLAN has a better performance than that of MUDABLUE, also with respect to *Confidence* and *Precision* [28]. In addition, REPOPAL works on GitHub Java repositories containing rich metadata that is suitable for building graph by CROSSSIM. Intuitively, we consider MUDABLUE, CLAN, and REPOPAL as a good starting point for a performance comparison with CROSSSIM.

We try to exploit the original implementation of the tools. The CROSSSIM source code and data are already available at the CROSSMINER 's GitHub repository [36]. Similarly, the REPOPAL implementation can be found from one of its author's repository [2]. Unfortunately, the public implementations of MUDABLUE and CLAN are no longer available. Thus, we had to re-implement the tools by strictly following the description in the corresponding papers [12],[28]. In the following sections, we are going to provide a description on the implementation of MUDABLUE and CLAN.

4.1 System Description

The use cases concerning the functionality of finding similar OSS projects are depicted in Figure 4.1. These use cases are directly extracted from the CROSSMINER project document. The Component Diagram is shown in Figure 4.2 and there are the following components:

- Project Repository manager: this is the component that provides the repositories and manages the file system.
- Parser: this component analyzes all the *.java* files in order to retrieve the keywords to create the term-document matrix. As stated before we search for the *JDK* related imports and methods for CLAN and any imports, method, variables and field variables for MUDABLUE.
- Matrix Manager: this is the central component, it manages the creation of the term-document matrix, and coordinates all the matrices "roaming" during the process.
- LSA Manager: by this component all the operations concerning the Latent Semantic Analysis occur, from the low-rank matrix reduction to the Singular Value Decomposition.

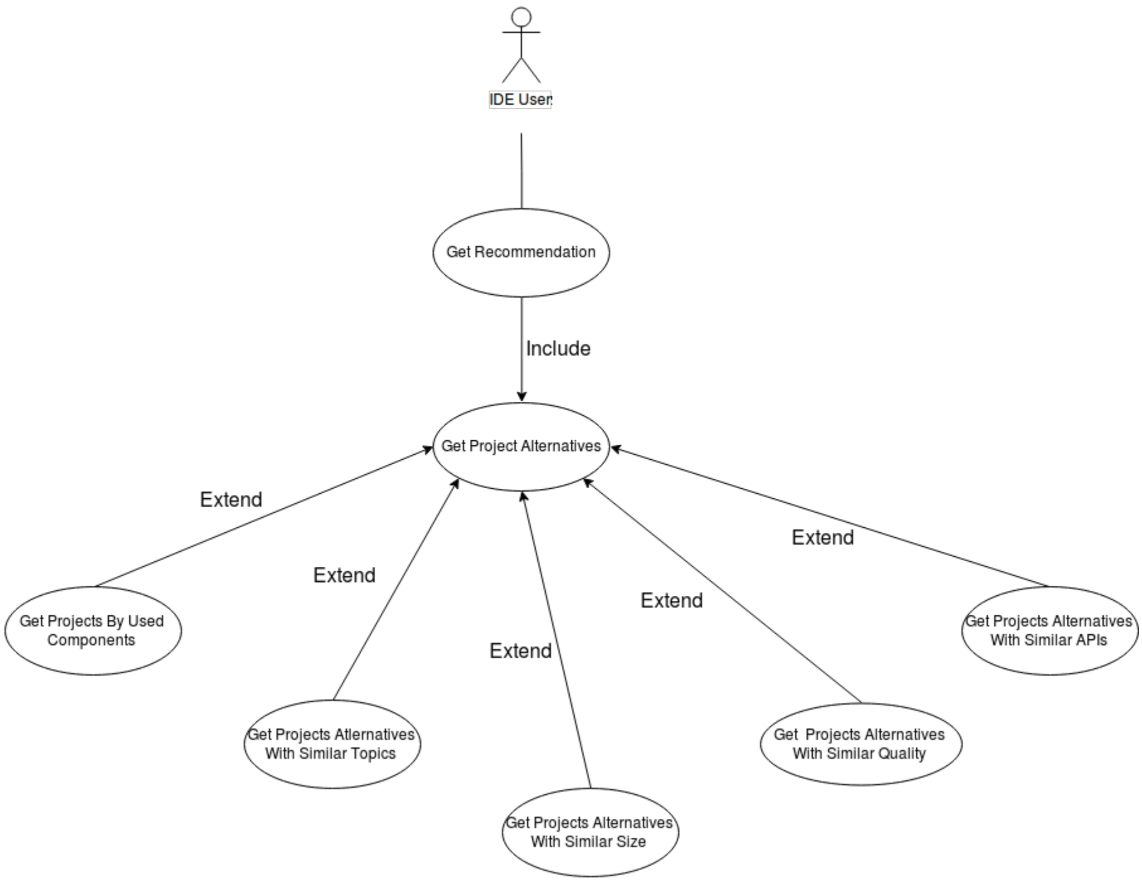

FIGURE 4.1: The Use Case Diagram

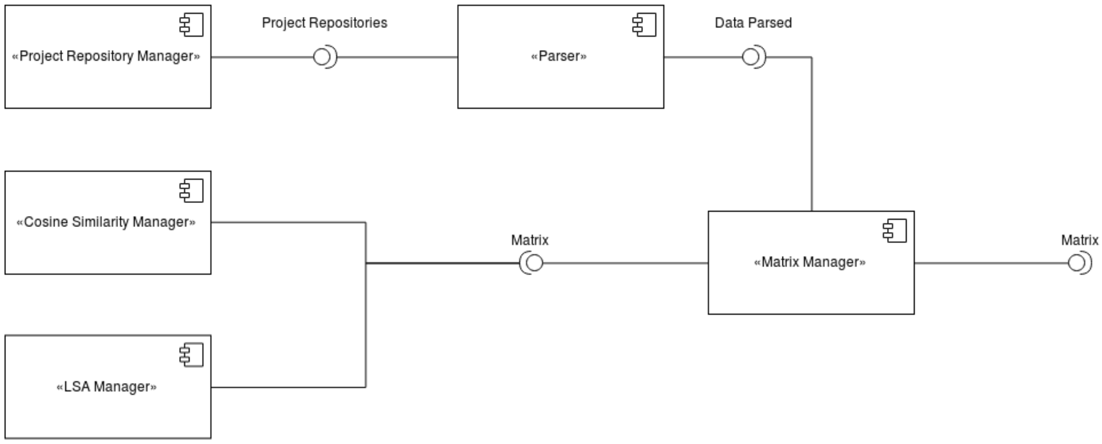

FIGURE 4.2: Component Diagram

- Cosine Similarity Manager: once the LSA completes its work, cosine similarity is then applied to get the final version of the matrix.

Figure 4.3 depicts the sequence diagram. When the process starts, the Project Repository Manager analyzes its file system in order to provide all the repositories to be analyzed. It also checks if the parsing has already occurred, this is due to the extremely high consumption of memory, thus the phases have been split in two moments. Once the repositories to be analyzed are known, the process can start. As already explained before, for MUDABLUE and CLAN the terms are different,

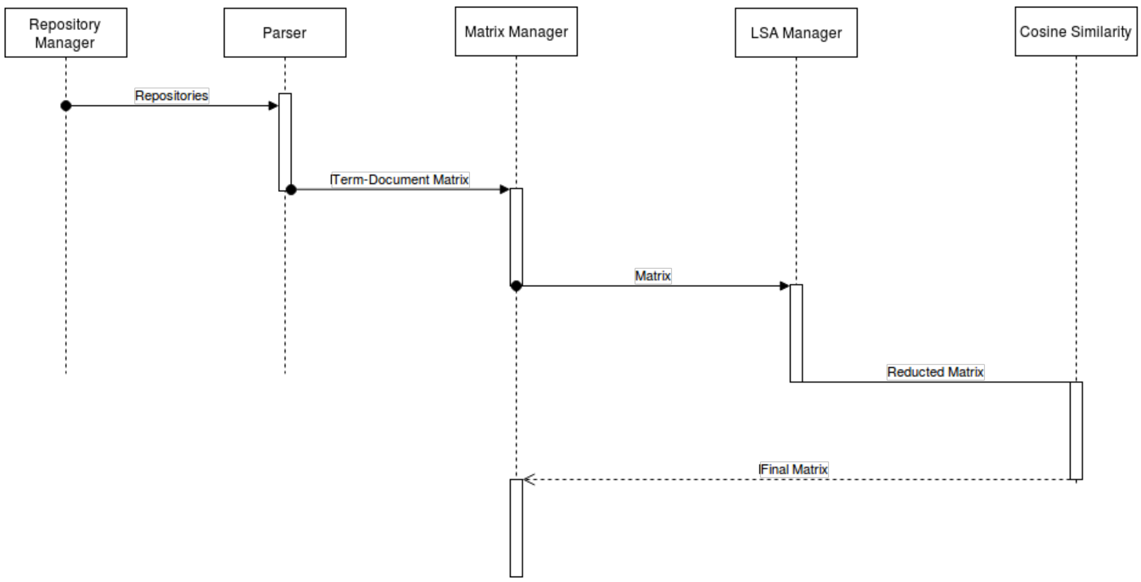

FIGURE 4.3: Sequence Diagram

however the same library *Java Parser* is used in both cases. The outcome is a term-document matrix which is processed by the Latent Semantic Analysis manager. The *commons math* component is used to decompose the matrix. Afterwards, the LSA matrix is obtained by multiplying the matrices. At this stage, it is necessary to take the matrix and then apply the cosine similarity. For each vector of the matrix, we calculate the cosine with all the others vectors. In this way, we get a final matrix of 580×580 which can be fed as input for further computations.

4.2 System Structure

Figure 4.3 shows the general architecture of the implemented systems. The process consists of the following steps.

- Retrieving the dataset, in this case a folder with all 580 repositories.
- All the `.java` files are parsed.
- Each repository is represented as a vector that contains all the frequencies.
- The SVD procedure is applied to decompose the matrix into 3 matrices.
- The matrices are multiplied back to realize the LSA procedure.
- Similarity between every vector is computed against all the remaining vectors.
- Eventually, the final similarity matrix is created.

The parsing step is conducted on all `.java` files of the 580 repositories using Java Parser. The main components of the files (import and method invocation for CLAN, import, method declaration, variables and field variables for MUDABLUE). For each repository we created a relative `.txt` file containing the frequencies. For CLAN such terms are filtered by searching only the terms belonging to the Java JDK. All these terms are merged to create another file, called `mainlist.txt` which is used to avoid reps. The idea is to parse the files and compare with the `mainlist.txt` to add new

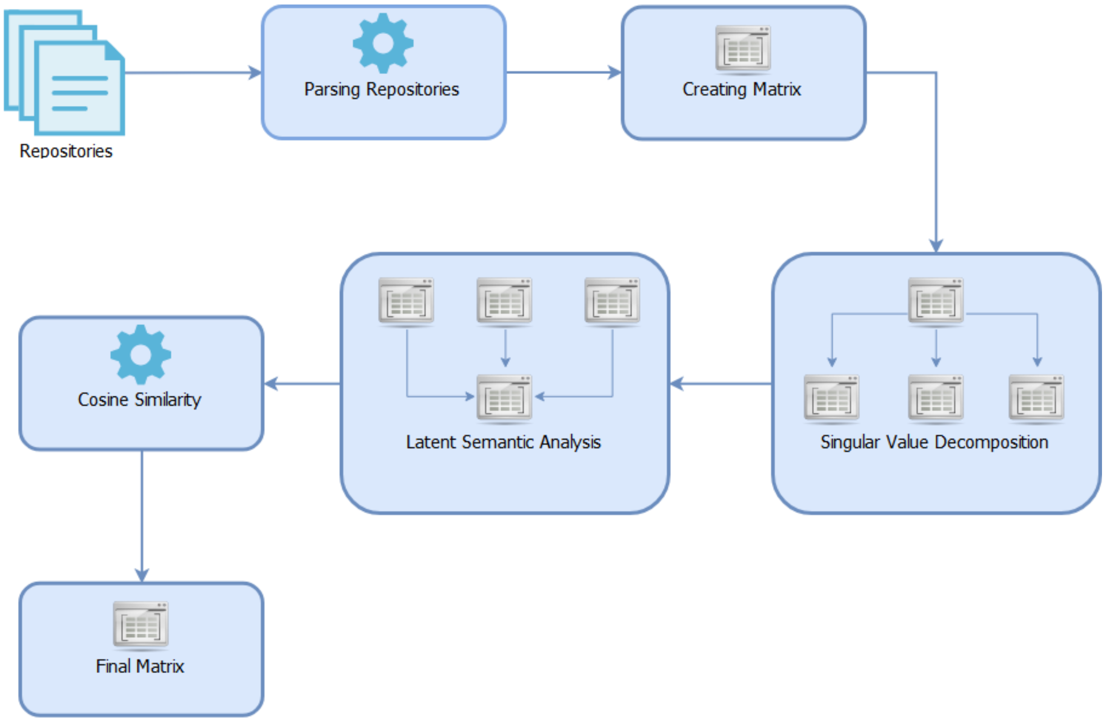

FIGURE 4.4: System Structure

terms, and then count, for each term how many times it appears inside the files. The final result is a vector of numbers.

Matrix Creation: Once all the repositories are analyzed, the term-document matrix can be created using the *apache commons math3* library, in particular the following components:

- ArrayRealVector.
- RealMatrix.
- RealVector.

Each file contains only its own terms, so the idea is, once the parsing process is done, to count how many terms we have and then, to add fill the missing terms with zeros. For example, considering a set of 3 documents A,B,C for 10 different terms, document contains 4 terms, this means that the other 6 terms are missing, so they are represented as 0.

SVD: As stated before the SVD operation is used to decompose the main matrix into 3 different matrices using *math3 linear SingularValueDecomposition*. So we invoke the methods and pass as parameter the term-document matrix.

LSA: Since there is no implementation of Latent Semantic Analysis available, we had to re-implement it from scratch. Among others, the most important issue is to identify a suitable value k for the reduced rank. We empirically selected a value of total $\frac{repository}{2}$. The computation complexity is a key issue since a total of memory needed for a matrix is as follows:

$$Memory\ in\ gigabytes = \frac{(columns * rows * 8)}{(1024 * 1024 * 1024)} \tag{4.1}$$

For MUDABLUE, we got 700,000 distinct terms for a total of 3GB of dedicated memory just for storing the matrix, without considering any kind of operations. This is due to the fact that MUDABLUE takes many different terms from a file into consideration. In contrast, CLAN focuses only on the import and method that belong to the *JDK* and this helps greatly reduce the number of distinct terms. A possible solution is to increase the available memory for Eclipse up to 8GB. Even then, many crashes can be seen. Thus, we need to perform some refactoring in the code in order to save memory, e.g. by deleting unused data structure, or using more light structures, etc.

By cosine similarity we mean a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. As for *LSA*, so the method takes as input two vectors and performs the operation. Since in the final matrix we have the similarity between *repo1 - repo2* and *repo2 - repo1*, the computation is done only for one pair in order to cut half of the calculation.

At this stage the matrix is of 580×580 in size, and with values ranging between *0.0* and *1.0*. This matrix is actually a collection of vectors, representing the similarity of a project with all the other projects. The final matrix $\|M\|$ is square matrix whose rows and columns represent projects.

$$M_{i,j} = \begin{cases} 0 \leq M \leq 1 & \text{if i} \neq \text{j} \\ \\ 1 & \text{if i = j.} \end{cases} \tag{4.2}$$

4.3 Tools and Libraries

The implementations have been conducted using Eclipse IDE Oxygen .2 and the following libraries:

- **org.eclipse.jdt.core 3.10.0**: this is the core part of Eclipse's Java development tools. It contains the non-UI support for compiling and working with Java code, including the following tools:
  - An incremental or batch Java compiler that can run standalone or as part of the Eclipse IDE.
  - Java source and class file indexer and search infrastructure.
  - A Java source code formatter.
  - APIs for code assist, access to the AST and structured manipulation of Java source.
- **eclipse-astparser 8.1**: this is used to analyze the AST at runtime on Eclipse.
- **commons-math3 3.6.1**: it is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang. In particular it is used to compute the SVD, singular value decomposition.
- **commons-text 1.2**: Apache Commons Text is a library focused on algorithms working on strings.
- **javaparser-core 3.5.14**: This is a library for parsing `.java` files.

- **ejml-0.33**: Efficient Java Matrix Library (EJML) is a linear algebra library for manipulating real/complex/dense/sparse matrices. The design goals are: 1) to be as computationally and memory efficient as possible for both small and large matrices, and 2) to be accessible to both novices and experts. These goals are accomplished by dynamically selecting the best algorithms to use at run-time, clean API, and multiple interfaces.

Chapter 5

Evaluation

In this section we discuss the process that has been conceived and applied to evaluate the performance the four approaches introduced in Chapter 4. To this end, the evaluation process that has been applied is shown in Figure 5.1 and consists of activities and artifacts that are going to be explained later on this chapter. In particular, a set of Java projects (Section 5.1) has been crawled to feed as input for the computation by all approaches, i.e., MUDABLUE, CLAN, REPOPAL, and CROSSSIM. Afterwards, a set of projects is selected as queries to compute similarities against all the remaining OSS projects. Once the scores have been computed, for each similarity tool, some of the top similar projects are chosen, mixed with results by the other tools, and eventually evaluated by humans (Section 5.2). The outcomes are then evaluated using various quality metrics (Section 5.3). Finally, the experimental results are discussed (Section 5.4).

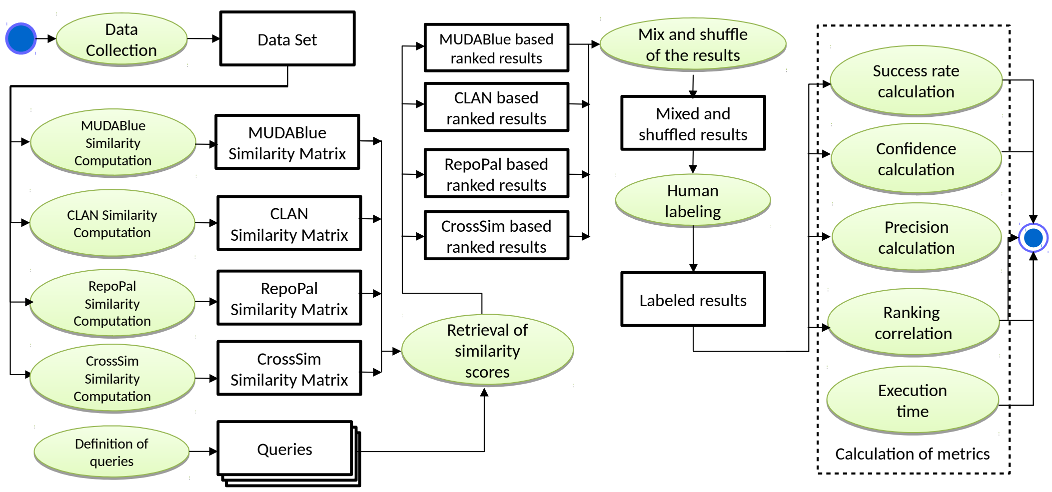

FIGURE 5.1: Evaluation process

5.1 Dataset & Queries

To serve as input for the evaluation, it is necessary to populate a dataset that meets the requirements by all four approaches. By MUDABlue and CLAN, there are no specific requirements since both metrics rely solely on source code to function. However, for CrossSim, we consider only projects that satisfy certain criteria. In particular, we collected projects that meet the following requirements:

- Being GitHub Java projects;

- Providing the specification of their dependencies by means of `code.xml` or `gradle` files;
- Including at least 9 dependencies. A project with no or little information about dependencies may adversely affect the performance of CROSSSIM;
- Having the `README.md` file available.

Furthermore, we realized that the final outcomes of a similarity algorithm are to be validated by human beings, and in case the projects are irrelevant by their very nature, the perception given by human evaluators would also be *dissimilar* in the end. This is valueless for the evaluation of similarity. Thus, to facilitate the analysis, instead of crawling projects in a random manner, we first observed projects in some specific categories (e.g. PDF processors, JSON parsers, Object Relational Mapping projects, and Spring MVC related tools). Once a certain number of projects for each category had been obtained, we also started collecting randomly to get projects from various categories.

Using the GitHub API[1], we crawled projects to provide input for the evaluation. Though the number of projects that fulfill the requirements of a single approach, i.e. either RepoPal or CrossSim, is high, the number of projects that meet the requirements of both approaches is considerably lower. For example, a project contains both `pom.xml` and `README.md`, albeit having only 5 dependencies, does not meet the constraints and must be discarded. The crawling is time consuming as for each project, at least 6 queries must be sent to get the relevant data. GitHub already sets a rate limit for an ordinary account[2], with a total number of $5,000$ API calls per hour being allowed. And for the search operation, the rate is limited to 30 queries per minute. Due to these reasons, we ended up getting a dataset of 580 projects that are eligible for the evaluation. The dataset we collected and the CrossSim tool are already published online for public usage [36].

No.	Name	# of Projects
1	SPARQL, RDF, Jena Apache	21
2	PDF Processor	8
3	Selenium Web Test	26
4	ORM	13
5	Spring MVC	51
6	Music Player	25
7	Boilerplate	38
8	Elastic Search	55
9	Hadoop, MapReduce	52
10	JSON	20
11	Miscellaneous Categories	271

TABLE 5.1: List of software categories

Further than collecting projects for each category, we also started collecting random projects. These projects serve as a means to test the stability of the algorithms. If the algorithms work well, they will not perceive newly added random projects as similar to projects of some other specific categories. To this end, the categories and their corresponding cardinality to be studied in our evaluation are listed in Table 5.1.

[1] GitHub API: `https://developer.github.com/v3/`
[2] GitHub Rate Limit: `https://developer.github.com/v3/rate_limit/`

This is an approximate classification since a project might belong to more than one category.

As can be seen in Table 5.1, among 580 considered projects, 309 of them belong to some specific categories, such as *SPARQL, RDF, Jena Apache*, *Selenium Test*, *Elastic Search*, *Spring MVC*, etc. The other 271 projects being selected randomly belong to *Miscellaneous Categories*. These categories disperse in several domains and sometimes it happens that there is only one project in a category. For the sake of clarity, we do not introduce the list of the categories in this thesis, interested readers are referred to our GitHub repository for more details [36].

Among 580 projects in the dataset, 50 have been selected as queries and they are listed in Table 5.2. To aim for variety, the queries have been chosen to cover different categories, e.g.: SPARQL and RDF, Selenium Test, Elastic Search, Spring MVC, Hadoop, Music Player.

No.	**Name**	**No.**	**Name**
1	neo4j-contrib/sparql-plugin	26	mariamhakobyan/elasticsearch-river-kafka
2	AskNowQA/AutoSPARQL	27	OpenTSDB/opentsdb-elasticsearch
3	AKSW/Sparqlify	28	codelibs/elasticsearch-cluster-runner
4	AKSW/SPARQL2NL	29	opendatasoft/elasticsearch-plugin-geoshape
5	pyvandenbussche/sparqles	30	huangchen007/elasticsearch-rest-command
6	sayems/java.webdriver	31	pitchpoint-solutions/sfs
7	xebia/Xebium	32	javanna/elasticsearch-river-solr
8	webdriverextensions/webdriverextensions	33	mesos/hadoop
9	testIT-WebTester/webtester-core	34	pentaho/big-data-plugin
10	seleniumQuery/seleniumQuery	35	asakusafw/asakusafw
11	bonigarcia/webdrivermanager	36	klarna/HiveRunner
12	selenium-cucumber/selenium-cucumber-java	37	sonalgoyal/hiho
13	conductor-framework/conductor	38	pranab/beymani
14	caelum/vraptor	39	lintool/Ivory
15	caelum/vraptor4	40	GoogleCloudPlatform/bigdata-interop
16	KEN-LJQ/WMS	41	Conductor/kangaroo
17	white-cat/jeeweb	42	datasalt/pangool
18	livrospringmvc/lojacasadocodigo	43	laserson/avro2parquet
19	spring-projects/spring-mvc-showcase	44	Knewton/KassandraMRHelper
20	sonian/elasticsearch-jetty	45	blackberry/KaBoom
21	dadoonet/spring-elasticsearch	46	jt6211/hadoop-dns-mining
22	elastic/elasticsearch-metrics-reporter-java	47	psaravan/JamsMusicPlayer
23	elastic/elasticsearch-support-diagnostics	48	TheAndroidMaster/Pasta-Music
24	SpringDataElasticsearchDevs/spring-data-elasticsearch	49	SubstanceMobile/GEM
25	javanna/elasticshell	50	markzhai/LyricHere

TABLE 5.2: List of queries for evaluation

Configuration	**Star events**	**Dependencies**	**Committers**	**Frequent Deps**
CROSSSIM$_1$	×	×		×
CROSSSIM$_2$	×	×	×	×
CROSSSIM$_3$	×	×		
CROSSSIM$_4$	×	×	×	

TABLE 5.3: CROSSSIM test configurations

Furthermore, in order to investigate the implication of graph structure on the performance of SimRank, different types of graph are considered for the evaluation. By the first configuration, only star events and dependencies are used to built the graph and this is named as CROSSSIM_1. In the second configuration CROSSSIM_2, all developers who have performed at least a push to update a project are added into the graph representing the project. Next, we study the influence of the most frequent dependencies on similarity computation. From the graph in CROSSSIM_1, all nodes and edges being derived from these dependencies are removed, and this configuration is denoted as CROSSSIM_3. Finally, the most frequent dependencies are removed from CROSSSIM_2, resulting in CROSSSIM_4. The test configurations are detailed in Table 5.3.

5.2 User Study

We performed a user study following the descriptions in [23],[28],[54] to evaluate the similarity between query projects and their corresponding retrieved projects. A group of 15 software developers who have at least 5 years of experience took part in the experiments. In order to have a fair evaluation, for each query we mixed and shuffled the top-5 results generated from the computation by all similarity metrics in a single file and present them to the evaluators. This mimics a *taste test* where users are asked to evaluate a product, e.g., food or drink, without having a priori knowledge about what is being addressed [13],[40]. This aims at eliminating any bias or prejudice against a specific similarity metric. The participants are asked to label the similarity for each pair of projects (i.e., <*query, retrieved project*>) with regards to their application domains and functionalities using the scales listed in Table 5.4 [28].

Scale	**Description**	**Score**
Dissimilar	The functionalities of the retrieved project are completely different from those of the query project	1
Neutral	The query and the retrieved projects share a few functionalities in common	2
Similar	The two projects share a large number of tasks and functionalities in common	3
Highly similar	The two projects share many tasks and functionalities in common and can be considered the same	4

TABLE 5.4: Similarity scales

For example, an OSS project p_1 that performs the sending of files across a TCP/IP network is somehow similar to an OSS project p_2 that exchanges text messages between two users, i.e., $Score(p_1, p_2) = 3$. However, an OSS project p_3 with the functionalities of a pure text editor is dissimilar to both p_1 and p_2, i.e., $Score(p_1, p_2) = Score(p_1, p_3) = 1$. Given a query, a retrieved project is considered as a *false positive* if its similarity to the query is labeled as `Dissimilar` (1) or `Neutral` (2). In contrast, *true positives* are those retrieved projects that have a score of 3 or 4, i.e.,`Similar` of `Highly similar`. A good similarity metric should produce as much true positives as possible.

5.3 Evaluation Metrics

To evaluate the outcomes of the algorithms with respect to the user study, the following metrics have been considered as typically done in related work [23, 28, 54]:

- *Success rate*: if at least one of the top-5 retrieved projects is labelled *Similar* or *Highly similar*, the query is considered to be successful. *Success rate* is the ratio of successful queries to the total number of queries;
- *Confidence*: Given a pair of <*query, retrieved project*> the confidence of an evaluator is the score she assigns to the similarity between the projects;
- *Precision*: The precision for each query is the proportion of projects in the top-5 list that are labelled as *Similar* or *Highly similar* by humans.

Further than the previous metrics, we introduce an additional one to measure the ranking produced by the similarity tools. For a query, a similarity tool is deemed to be good if all top-5 retrieved projects are relevant. In case there are false positives, i.e. those that are labeled *Dissimilar* and *Neutral*, it is expected that these will be ranked lower than the true positives. In case an irrelevant project has a higher rank than that of a relevant project, we suppose that the similarity tool is generating an improper recommendation. The *Ranking* metric presented below is a means to evaluate whether a similarity metric produces properly ranked recommendations.

- *Ranking*: the obtained human evaluation has been analysed to check the correlations among the ranking calculated by the similarity tools and the scores given by the human evaluation. To this end the Spearman's rank correlation coefficient r_s [47] is used to measure how well a similarity metric ranks the retrieved projects given a query. Considering two ranked variables $r_1 = (\rho_1, \rho_2, .., \rho_n)$ and $r_2 = (\sigma_1, \sigma_2, .., \sigma_n)$, r_s is defined as: $r_s = 1 - \frac{6\sum_{i=1}^{n}(\rho_i - \sigma_i)^2}{n(n^2-1)}$; r_s ranges from -1.00 (perfect negative correlation) and $+1.00$ (perfect positive correlation); $r_s = 0$ implies that the two variables are not correlated. Because of the large number of ties, we also used *Kendall's tau* [18] coefficient, which is used to measure the ordinal association between two considered quantities. Similar to Spearman's correlation coefficient, values of Kendall's tau correlation coefficient τ range from -1 (perfect negative correlation) to +1 (perfect positive correlation).

5.4 Results

To study the performance of the metrics in detecting similar projects for the set of queries, we consider the following research questions:

RQ*$_1$*: Which similarity metric yields a better performance in terms of Success rate and Precision?

The experimental results suggest that RepoPal is a good choice for computing similarity among OSS projects. This indeed confirms the claim made by the authors of RepoPal in [54]. In comparison with the other metrics, i.e., MUDABLUE, CLAN and REPOPAL, three CROSSSIM configurations gain a superior performance, with CROSSSIM3 overtaking all. As can be seen in and Figure 5.2, CROSSSIM3 outperforms RepoPal with respect to *Precision*. Both gain a *success rate* of 100%, however CROSSSIM3 has a better precision. CROSSSIM3 obtains a precision of 0.78 and RepoPal gets 0.71. The *Confidence* for both metrics is shown in Figure 5.4. Also by this index, CROSSSIM3 yields a better outcome as it has more scores that are either 3 or 4 and less scores that are 1 or 2. In addition to the conventional quality indexes, we investigated the ranking produced by the two metrics using the Spearman's (r_s) and Kendall's tau (τ) correlation indexes. The aim is to see how good is the correlation between the rank generated by each metric and the scores given by the users, which are already sorted in descending order. In this way, a lower r_s (τ) means a better ranking. r_s and τ are computed for all 50 queries and related first five results. The value of r_s is 0.250 for CROSSSIM3 and -0.193 for *RepoPal*. The value of τ is -0.214 for CROSSSIM3 and -0.163 for REPOPAL. By this quality index, CROSSSIM3 performs slightly better than REPOPAL.

The results obtained by CROSSSIM confirm our hypothesis that the incorporation of various features, e.g. dependencies and star events into graph is beneficial to similarity computation. To compute similarity between two projects, RepoPal considers the relationship between the projects per se, whereas CROSSSIM takes also the cross relationships among other projects into account by means of graphs. Furthermore, CROSSSIM is more flexible as it can include other artifacts in similarity computation, on the fly, without affecting the internal design. Last but not least, the ratio between the overall performance of CROSSSIM and its execution time is very encouraging.

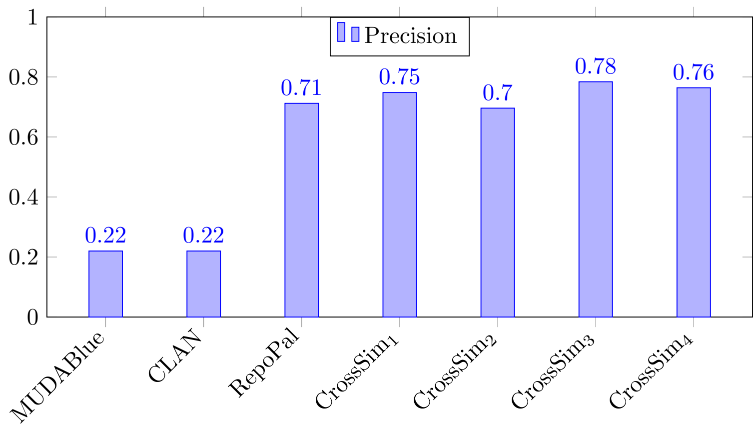

FIGURE 5.2: Precision Comparison

Experimental results suggests that CROSSSIM approach outperforms all the other approaches, in particular MUDABLUE and CLAN. REPOPAL got a good score, this means that is still a valid choice for similarity in the OSS environment. The precision, as the figure 5.2 depicts, shows that CrossSim and Repopal got a score *greater than* 70%. Clan and MudaBlue instead, reported a very low score, *about 20%*, on 10 queries evaluted, just 2 got a score ≥ 3.

Concerning the success rate, the results of CrossSim and Repopal are quite impressive, about *100%* of queries got score high, the situation is lower for Clan and MudaBlue that achieved just the *60%* of the queries. In order to calculate this values, we counted for each query how many votes were ≥ 3 divided then by 25, which is the number of queries.

The confidence confirms what stated so far, the mojority of the votes for MudaBlue and Clan are between 1 and 2,that is, users evaluated as dissimilar most of the projects. For CrossSim the result is quite more nice, with 130 rank 3 votes and 60 rank 4 votes, so more than half results are good. Repopal also got a good evaluation, close to CrossSim but a bit lower.

RQ_2: Which similarity metric is more efficient?

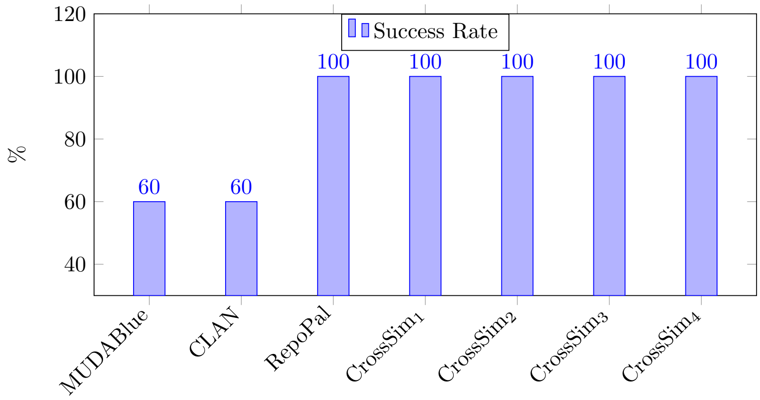

FIGURE 5.3: Success Rate Comparison

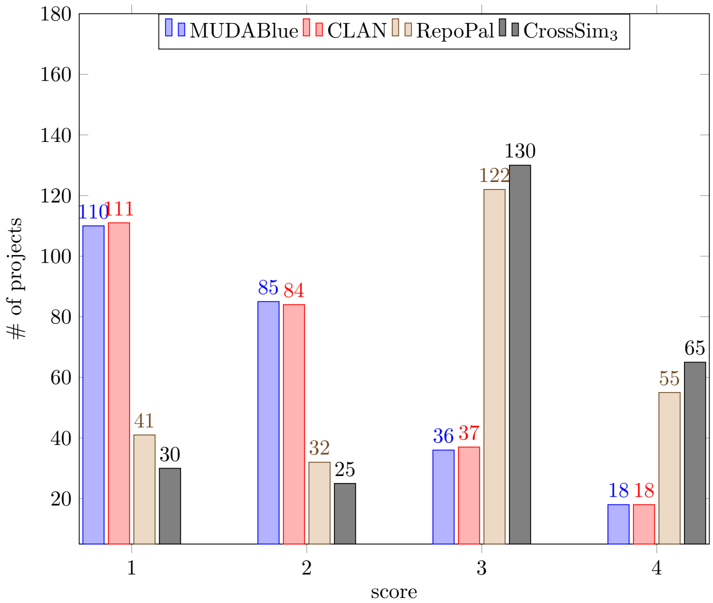

FIGURE 5.4: Confidence Comparison

An important factor for a similarity metric is the ability to compute within an acceptable amount of time. The execution time related to the application of REPOPAL and CROSSSIM$_3$ is shown in Figure 5.5. For the experiments on the dataset using a laptop with Intel Core i5-7200U CPU @ 2.50GHz $\times$ 4, 8GB RAM, Ubuntu 16.04, RepoPal takes $\approx$4 hours to generate the similarity matrix, whereas the execution of CROSSSIM$_3$, including both the time for generating the input graph and that for generating the similarity matrix, takes $\approx$16 minutes. Such an important time difference is due to the time needed to calculate the similarity between `README.md` files, on which *RepoPal* relies.

RQ*$_3$*: How does the graph structure affect the performance of* CROSSSIM*?

When we consider CROSSSIM$_1$ in combination with CROSSSIM$_2$, the effect of the adoption of committers can be observed. CROSSSIM$_1$ gains a success rate of 100%, with a precision of 0.748 and 63 false positives. Whereas, the number of false positives by CROSSSIM$_2$ goes up to 76, thereby worsening the overall performance considerably with 0.696 being as the precision. The precision of CROSSSIM$_2$ is higher than those of *Readme*, *Dependency*, *Compound*, *Weighted RepoPal*, but lower than

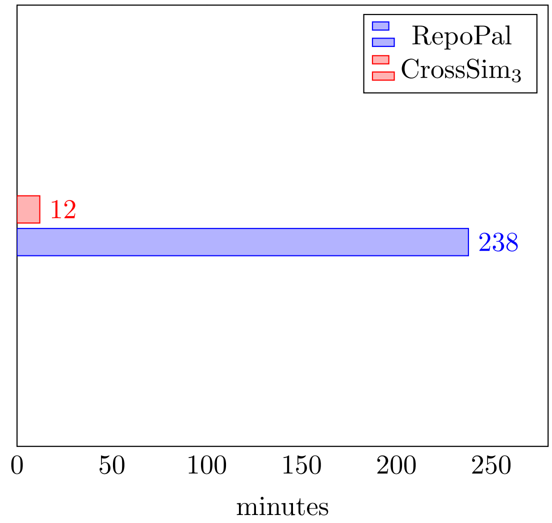

FIGURE 5.5: Execution Time Comparison

those of *RepoPal* and all of its CROSSSIM counterparts. The performance degradation is further witnessed by considering CROSSSIM$_3$ and CROSSSIM$_4$ together. With respect to CROSSSIM$_3$, the number of false positives by CROSSSIM$_4$ increases by 5 projects. We come to the conclusion that the inclusion of all developers who have committed updates at least once to a project in the graph is counterproductive as it adds a decline in precision. In this sense, we make an assumption that the deployment of a weighting scheme for developers may help counteract the degradation in performance. We consider the issue as our future work.

Next, CROSSSIM$_1$ and CROSSSIM$_3$ are studied together to analyze the effect of the removal of the most frequent dependencies. CROSSSIM$_3$ outperforms CROSSSIM$_1$ as it gains a precision of 0.784, the highest value among all, compared to 0.748 by CROSSSIM$_1$. The removal of the most frequent dependencies helps also improve the performance of CROSSSIM$_4$ in comparison to CROSSSIM$_2$, which is a similar configuration, except that all dependencies are taken into account. Together, this implies that the elimination of too popular dependencies in the original graph is a profitable amendment. This is understandable once we get a deeper insight into the design of SimRank as already presented in Section 2.5 and Figure 2.10. There, two projects are deemed to be similar if they share a same dependency, or in other words their corresponding nodes in the graph are pointed by a common node. However, with frequent dependencies this characteristic may not hold anymore. Take as an example, two projects are pointed by a frequent dependency, e.g. junit:junit because they use JUnit[3] for testing. And since testing is a common functionality of many software projects, it does not help contribute towards the characterization of a project and as a result, needs to be removed from similarity computation.

5.5 Threats to Validity

In this section, we investigate the threats that may affect the validity of the experiments as well as how we have tried to minimize them. In particular, we focus on the following threats to validity as discussed below.

[3]JUnit: Testing Framework for Java 8: http://junit.org/junit5/

Internal validity concerns any confounding factor that could influence our results. We attempted to avoid any bias in the evaluation and assessment phases: (*i*) by involving three participants in the user study. In particular, the labeling results by one user were then double-checked by other two users to make sure that the outcomes were sound; (*ii*) by completely automating the evaluation of the defined metrics without any manual intervention. Indeed, the implemented tools could be defective. To contrast and mitigate this threat, we have run several manual assessments and counter-checks.

External validity refers to the generalizability of obtained results and findings. Concerning the generalizability of our approach, we were able to consider a dataset of 580 projects, due to the fact that the number of projects that meet the requirements of both RepoPal and CrossSim is low and thus required a prolonged crawling. During the data collection, we crawled both projects in some specific categories as well as random projects. The random projects served as a means to test the generalizability of our algorithm. If the algorithm works well, it will not perceive newly added random projects as similar to projects of the specific categories.

Reliable validity is related to the reproducibility of our experiments. To allow anyone to seamlessly replicate the evaluation, we made available the source code implementation of MUDABLUE, CLAN, REPOPAL, and CROSSSIM as also the dataset exploited in the paper in our GitHub repository [36].

Chapter 6

Conclusions

Measuring similarities between software systems has been considered as a daunting task. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail. Thus, choosing the right tool to compute software similarity is a question that may arise at any time. The current thesis attempts to address one of the issues in software similarity computation by performing a comprehensive evaluation on various techniques. We performed a literature review on different approaches for computing software similarity. We see that depending on the set of mined features, there are two main types of software similarity computation techniques. The first type is *Low-level Similarity* where only low-level data, e.g., source code, byte code, function calls, API reference, etc. is considered. The second type is *High-level Similarity* and it detects the semantic similarity using metadata, such as: topic distribution, readme file, description, star events, etc. Source code is not taken into account.

Most low-level similarity algorithms attempt to represent source code (and API calls) in a term-document matrix and then apply SVD to reduce dimensionality. The similarity is then computed as the cosine similarity between feature vectors. Among others, MUDABLUE [12], CLAN [28], and CLANdroid [20] belong to this category. CLAN includes API calls for computing similarity, whereas, by MUDABLUE, every word appearing in source code files is integrated into the term-document matrix. This makes the difference in the performance of the two algorithms in a way that the similarity scores of CLAN reflect better the perception of humans of similarity than those of MUDABLUE. In contrast, high-level similarity techniques do not consider source code for similarity computation. They characterize software by exploiting available features such as descriptions, user reviews, and `README.MD` file. The similarity is computed as the cosine similarity of the corresponding feature vectors. For computing similarity between mobile applications, other specific features such as images and permissions are also incorporated.

We re-implement four software similarity tools and conduct an empirical evaluation using a dataset of 580 GitHub Java projects collected from GitHub. The obtained results are promising: by considering MUDABLUE, CLAN, and REPOPAL as baseline, we demonstrated that CROSSSIM is considered as a good candidate for computing similarities among open source software projects. CROSSSIM is an extensible and flexible approach to calculate the similarity of open source projects. It can deal with various types of input project information that is represented in a homogeneous manner by means of graphs. By means of the proposed graph representation, it is possible to transform the relationships among various artifacts, e.g. developers, API utilizations, source code, interactions, into a mathematically computable format. In this sense, CROSSSIM is a versatile similarity tool as it can accept various input features regardless of their format.

Bibliography

[1] Alessandra Bagnato et. al. "Developer-Centric Knowledge Mining from Large Open-Source Software Repositories (CROSSMINER)". In: *Software Technologies: Applications and Foundations*. Springer International Publishing, 2018, pp. 375–384. ISBN: 978-3-319-74730-9.

[2] Zhang Yun et al. *RepoPal Implementation*. https://github.com/yunzhang28/RepoPal. 2018.

[3] Jafar M. Al-Kofahi et al. "Fuzzy Set Approach for Automatic Tagging in Evolving Software". In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-8630-4. DOI: 10.1109/ICSM.2010.5609751. URL: http://dx.doi.org/10.1109/ICSM.2010.5609751.

[4] Kirk Baker. "Singular Value Decomposition Tutorial". 2005.

[5] Upasna Bhandari et al. "Serendipitous Recommendation for Mobile Apps Using Item-Item Similarity Graph." In: *AIRS*. Ed. by Rafael E. Banchs et al. Vol. 8281. Lecture Notes in Computer Science. Springer, 2013, pp. 440–451. ISBN: 978-3-642-45067-9. URL: http://dblp.uni-trier.de/db/conf/airs/airs2013.html#BhandariSDJ13.

[6] C. Bizer, T. Heath, and T. Berners-Lee. "Linked data - the story so far". In: *Int. J. Semantic Web Inf. Syst.* 5.3 (2009), 1–22.

[7] Vincent D. Blondel et al. "A Measure of Similarity Between Graph Vertices: Applications to Synonym Extraction and Web Searching". In: *SIAM Rev.* 46.4 (Apr. 2004), pp. 647–666. ISSN: 0036-1445. DOI: 10.1137/S0036144502415960. URL: http://dx.doi.org/10.1137/S0036144502415960.

[8] Ning Chen et al. "SimApp: A Framework for Detecting Similar Mobile Applications by Online Kernel Learning". In: *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. WSDM '15. Shanghai, China: ACM, 2015, pp. 305–314. ISBN: 978-1-4503-3317-7. DOI: 10.1145/2684822.2685305. URL: http://doi.acm.org/10.1145/2684822.2685305.

[9] Ronan Collobert et al. "Natural Language Processing (Almost) from Scratch". In: *J. Mach. Learn. Res.* 12 (Nov. 2011), pp. 2493–2537. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2078186.

[10] Jonathan Crussell, Clint Gibler, and Hao Chen. "AnDarwin: Scalable Detection of Semantically Similar Android Applications". In: *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*. Ed. by Jason Crampton, Sushil Jajodia, and Keith Mayes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 182–199. ISBN: 978-3-642-40203-6. DOI: 10.1007/978-3-642-40203-6_11. URL: https://doi.org/10.1007/978-3-642-40203-6_11.

[11] Tommaso Di Noia et al. "Linked Open Data to Support Content-based Recommender Systems". In: *Proceedings of the 8th International Conference on Semantic Systems*. I-SEMANTICS '12. Graz, Austria: ACM, 2012, pp. 1–8. ISBN: 978-1-4503-1112-0. DOI: 10.1145/2362499.2362501. URL: http://doi.acm.org/10.1145/2362499.2362501.

[12] Pankaj K. Garg et al. "MUDABlue: An Automatic Categorization System for Open Source Repositories". In: *2013 20th Asia-Pacific Software Engineering Conference (APSEC)* (2004), pp. 184–193. ISSN: 1530-1362. DOI: doi.ieeecomputersociety.org/10.1109/APSEC.2004.69.

[13] Sanjoy Ghose and Oded Lowengart. "Taste tests: Impacts of consumer perceptions and preferences on brand positioning strategies". In: *Journal of Targeting, Measurement and Analysis for Marketing* 10.1 (2001), pp. 26–41. ISSN: 1479-1862. DOI: 10.1057/palgrave.jt.5740031. URL: https://doi.org/10.1057/palgrave.jt.5740031.

[14] Lan Huang et al. "Learning a Concept-based Document Similarity Measure". In: *J. Am. Soc. Inf. Sci. Technol.* 63.8 (Aug. 2012), pp. 1593–1608. ISSN: 1532-2882. DOI: 10.1002/asi.22689. URL: http://dx.doi.org/10.1002/asi.22689.

[15] Aminul Islam and Diana Inkpen. "Semantic Text Similarity Using Corpus-based Word Similarity and String Similarity". In: *ACM Trans. Knowl. Discov. Data* 2.2 (July 2008), 10:1–10:25. ISSN: 1556-4681. DOI: 10.1145/1376815.1376819. URL: http://doi.acm.org/10.1145/1376815.1376819.

[16] Paul Jaccard. "The Distribution of the Flora in the Alpine Zone". In: *New Phytologist* 11.2 (1912), pp. 37–50. DOI: 10.1111/j.1469-8137.1912.tb05611.x.

[17] Glen Jeh and Jennifer Widom. "SimRank: A Measure of Structural-context Similarity". In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '02. Edmonton, Alberta, Canada: ACM, 2002, pp. 538–543. ISBN: 1-58113-567-X. DOI: 10.1145/775047.775126. URL: http://doi.acm.org/10.1145/775047.775126.

[18] Maurice George Kendall. "Rank correlation methods." In: (1948).

[19] T.K. Landauer, P.W. Foltz, and D. Laham. "An introduction to latent semantic analysis". In: *Discourse processes* 25 (1998), pp. 259–284.

[20] Mario Linares-Vasquez, Andrew Holtzhauer, and Denys Poshyvanyk. "On automatically detecting similar Android apps". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* 00 (2016), pp. 1–10. DOI: doi.ieeecomputersociety.org/10.1109ICPC.2016.7503721.

[21] Greg Linden, Brent Smith, and Jeremy York. "Amazon.Com Recommendations: Item-to-Item Collaborative Filtering". In: *IEEE Internet Computing* 7.1 (Jan. 2003), pp. 76–80. ISSN: 1089-7801. DOI: 10.1109/MIC.2003.1167344. URL: http://dx.doi.org/10.1109/MIC.2003.1167344.

[22] Chao Liu et al. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: ACM, 2006, pp. 872–881. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150522. URL: http://doi.acm.org/10.1145/1150402.1150522.

[23] David Lo, Lingxiao Jiang, and Ferdian Thung. "Detecting Similar Applications with Collaborative Tagging". In: *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 600–603. ISBN: 978-1-4673-2313-0. DOI: 10.1109/ICSM.2012.6405331. URL: http://dx.doi.org/10.1109/ICSM.2012.6405331.

[24] Wangzhong Lu et al. "Node similarity in the citation graph". In: *Knowledge and Information Systems* 11.1 (2007), pp. 105–129. ISSN: 0219-3116. DOI: 10.1007/s10115-006-0023-9. URL: https://doi.org/10.1007/s10115-006-0023-9.

[25] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. "An Information Retrieval Approach for Automatically Constructing Software Libraries". In: *IEEE Trans. Softw. Eng.* 17.8 (Aug. 1991), pp. 800–813. ISSN: 0098-5589. DOI: 10.1109/32.83915. URL: http://dx.doi.org/10.1109/32.83915.

[26] Ainura Madylova and Sule Gündüz Ögüducü. "A taxonomy based semantic similarity of documents using the cosine measure." In: *ISCIS*. IEEE, Dec. 30, 2009, pp. 129–134. URL: http://dblp.uni-trier.de/db/conf/iscis/iscis2009.html#Madylova009.

[27] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521865719, 9780521865715.

[28] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. "Detecting Similar Software Applications". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 364–374. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337267.

[29] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. "Corpus-based and Knowledge-based Measures of Text Semantic Similarity". In: *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*. AAAI'06. Boston, Massachusetts: AAAI Press, 2006, pp. 775–780. ISBN: 978-1-57735-281-5. URL: http://dl.acm.org/citation.cfm?id=1597538.1597662.

[30] George A. Miller. "WordNet: A Lexical Database for English". In: *Commun. ACM* 38.11 (Nov. 1995), pp. 39–41. ISSN: 0001-0782. DOI: 10.1145/219717.219748. URL: http://doi.acm.org/10.1145/219717.219748.

[31] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. "Knowledge-aware Recommender System for Software Development". In: *Proceedings of the 1st Workshop on Knowledge-aware and Conversational Recommender System*. KaRS 2018. Vancouver, Canada: ACM, 2018.

[32] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. "Mining Software repositories to support OSS developers: A recommender systems approach". In: *Proceedings of the 9th Italian Information Retrieval Workshop, Roma, Italy, May 28-30, 2018.* 2018.

[33] Phuong T. Nguyen et al. "An Evaluation of SimRank and Personalized PageRank to Build a Recommender System for the Web of Data". In: *Proceedings of the 24th International Conference on World Wide Web*. WWW '15 Companion. Florence, Italy: ACM, 2015, pp. 1477–1482. ISBN: 978-1-4503-3473-0. DOI: 10.1145/2740908.2742141. URL: http://doi.acm.org/10.1145/2740908.2742141.

[34] Phuong T. Nguyen et al. "Content-Based Recommendations via DBpedia and Freebase: A Case Study in the Music Domain". In: *Proceedings of the 14th International Conference on The Semantic Web - ISWC 2015 - Volume 9366*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 605–621. ISBN: 978-3-319-25006-9. DOI: 10.1007/978-3-319-25007-6_35. URL: http://dx.doi.org/10.1007/978-3-319-25007-6_35.

[35] Phuong T. Nguyen et al. "CrossSim: exploiting mutual relationships to detect similar OSS projects". In: *Procs. of 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) - to appear*. 2018.

[36] Phuong T. Nguyen et al. *CrossSim tool and evaluation data*. https://doi.org/10.5281/zenodo.1252866. 2018.

[37] Tommaso Di Noia and Vito Claudio Ostuni. "Recommender Systems and Linked Open Data". In: *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*. 2015, pp. 88–113. DOI: 10.1007/978-3-319-21768-0_4. URL: https://doi.org/10.1007/978-3-319-21768-0_4.

[38] Vito Claudio Ostuni et al. "A Linked Data Recommender System using a Neighborhood-based Graph Kernel". In: *The 15th International Conference on Electronic Commerce and Web Technologies*. Lecture Notes in Business Information Processing. Springer, 2014. URL: http://sisinflab.poliba.it/sisinflab/publications/2014/ODMD14a.

[39] Michael J. Pazzani and Daniel Billsus. "Content-Based Recommendation Systems". In: *The Adaptive Web: Methods and Strategies of Web Personalization*. Ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 325–341. ISBN: 978-3-540-72079-9. DOI: 10.1007/978-3-540-72079-9_10. URL: https://doi.org/10.1007/978-3-540-72079-9_10.

[40] Simone Pettigrew and Stephen Charters. "Tasting as a projective technique". In: *Qualitative Market Research: An International Journal* 11.3 (2008), pp. 331–343. DOI: 10.1108/13522750810879048. eprint: https://doi.org/10.1108/13522750810879048. URL: https://doi.org/10.1108/13522750810879048.

[41] Luca Ponzanelli et al. "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter". In: *Proceedings of MSR 2014*. Hyderabad, India: ACM, 2014, pp. 102–111. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597077. URL: http://doi.acm.org/10.1145/2597073.2597077.

[42] Juan Ramos. *Using TF-IDF to Determine Word Relevance in Document Queries*. 1999.

[43] Joel W. Reed et al. "TF-ICF: A New Term Weighting Scheme for Clustering Dynamic Data Streams". In: *Proceedings of the 5th International Conference on Machine Learning and Applications*. ICMLA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 258–263. ISBN: 0-7695-2735-3. DOI: 10.1109/ICMLA.2006.50. URL: http://dx.doi.org/10.1109/ICMLA.2006.50.

[44] Badrul Sarwar et al. "Item-based Collaborative Filtering Recommendation Algorithms". In: *Proceedings of the 10th International Conference on World Wide Web*. WWW '01. Hong Kong, Hong Kong: ACM, 2001, pp. 285–295. ISBN: 1-58113-348-0. DOI: 10.1145/371920.372071. URL: http://doi.acm.org/10.1145/371920.372071.

[45] Satu Elisa Schaeffer. “Survey: Graph Clustering”. In: *Comput. Sci. Rev.* 1.1 (Aug. 2007), pp. 27–64. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2007.05.001. URL: http://dx.doi.org/10.1016/j.cosrev.2007.05.001.

[46] J. Ben Schafer et al. “The Adaptive Web”. In: ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Berlin, Heidelberg: Springer-Verlag, 2007. Chap. Collaborative Filtering Recommender Systems, pp. 291–324. ISBN: 978-3-540-72078-2. URL: http://dl.acm.org/citation.cfm?id=1768197.1768208.

[47] Charles Spearman. “The proof and measurement of association between two things”. In: *The American journal of psychology* 15.1 (1904), pp. 72–101.

[48] F. Thung, D. Lo, and J. Lawall. “Automated library recommendation”. In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 182–191. DOI: 10.1109/WCRE.2013.6671293.

[49] Peter D. Turney and Patrick Pantel. “From Frequency to Meaning: Vector Space Models of Semantics”. In: *J. Artif. Int. Res.* 37.1 (Jan. 2010), pp. 141–188. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1861751.1861756.

[50] Amos Tversky. “Features of similarity”. In: *Psychological Review* 84.4 (1977), pp. 327–352. ISSN: 19391471. DOI: 10.1037/0033-295X.84.4.327.

[51] Secil Ugurel, Robert Krovetz, and C. Lee Giles. “What’s the Code?: Automatic Classification of Source Code Archives”. In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’02. Edmonton, Alberta, Canada: ACM, 2002, pp. 632–638. ISBN: 1-58113-567-X. DOI: 10.1145/775047.775141. URL: http://doi.acm.org/10.1145/775047.775141.

[52] Haoyu Wang et al. “WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 71–82. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771795. URL: http://doi.acm.org/10.1145/2771783.2771795.

[53] Xin Xia et al. “Tag Recommendation in Software Information Sites”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 287–296. ISBN: 978-1-4673-2936-1. URL: http://dl.acm.org/citation.cfm?id=2487085.2487140.

[54] Yun Zhang et al. “Detecting similar repositories on GitHub”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* 00 (2017), pp. 13–23. DOI: doi.ieeecomputersociety.org/10.1109/SANER.2017.7884605.