Software Quality Journal (2020) 28:595–631
https://doi.org/10.1007/s11219-019-09483-0

An automated approach to assess the similarity of GitHub repositories

Phuong T. Nguyen[1] · Juri Di Rocco[1] · Riccardo Rubei[1] · Davide Di Ruscio[1]

Published online: 15 February 2020

Abstract

Open source software (OSS) allows developers to study, change, and improve the code free of charge. There are several high-quality software projects which deliver stable and well-documented products. Most OSS forges typically sustain active user and expert communities which in turn provide decent levels of support both with respect to answering user questions as well as to repairing reported software bugs. Code reuse is an intrinsic feature of OSS, and developing a new system by leveraging existing open source components can reduce development effort, and thus it can be beneficial to at least two phases of the software life cycle, i.e., implementation and maintenance. However, to improve software quality, it is essential to develop a system by learning from well-defined, mature projects. In this sense, the ability to find similar projects that facilitate the undergoing development activities is of high importance. In this paper, we address the issue of mining open source software repositories to detect similar projects, which can be eventually reused by developers. We propose CrossSim as a novel approach to model the OSS ecosystem and to compute similarities among software projects. An evaluation on a dataset collected from GitHub shows that our proposed approach outperforms three well-established baselines.

Keywords Mining software repositories · Software similarity · Software quality · SimRank

✉ Davide Di Ruscio
davide.diruscio@univaq.it

Phuong T. Nguyen
phuong.nguyen@univaq.it

Juri Di Rocco
juri.dirocco@univaq.it

Riccardo Rubei
riccardo.rubei@univaq.it

[1] Department of Information Engineering, Computer Science and Mathematics, Università degli Studi dell'Aquila, Via Vetoio 2, 67100 L'Aquila, Italy

1 Introduction

Open source software (OSS) repositories contain a large amount of data, which can be of high value when developing new software systems without reimplementing already in place functionalities. The benefits resulting from the reuse of properly selected open source projects are manyfold including the fact that the system being implemented relies on open source code, "which is of higher quality than the custom-developed code's first incarnation" (Spinellis and Szyperski 2004). In addition to source code, also metadata available from different related sources, e.g., communication channels and bug tracking systems, can be beneficial to the development process if properly mined (Ponzanelli et al. 2014).

Mining OSS repositories In recent years, considerable effort has been made in the domain of mining software repositories to conceive techniques and tools to help developers mine and cope with the large amount of data available in OSS repositories. The main goal is to support software developers by providing them with meaningful recommendations. This is synthesized by relying on existing systems and past experiences. For instance, when a developer works on a new project, it is possible to provide her with suggestions about which libraries she should use, based on a comparison with other similar projects (Nguyen et al. 2018b; Thung et al. 2013). Moreover, 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 (Ponzanelli et al. 2014). In the scope of the CROSSMINER project,[1] we aim 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 (Bagnato et al. 2018). We attempt to support software developers by means of an advanced Eclipse-based IDE providing intelligent recommendations that go far beyond the current *code completion-oriented* practice. To this end, metadata is curated from different OSS forges and processed to properly feed the recommendation engines (Nguyen et al. 2019b). By means of the augmented IDE, developers are able to select open source software and get real-time recommendations based on the conceived mining tools (Nguyen et al. 2018a).

Software similarity Copying and pasting is a common practice in software development (Rattan et al. 2013). One of the main countermeasures to deal with software clone is to measure structural similarity among programs, aiming to evaluate their correctness, style, and uniqueness (Gitchell and Tran 1999). The concept of similarity is a key issue in various contexts, such as detecting cloned code (Evans et al. 2009; Ragkhitwetsagul et al. 2018a, b; Tiarks et al. 2011) software plagiarism (Leitão 2004), or reducing test suite in model-based testing (Coutinho et al. 2014; Khan et al. 2016). Nevertheless, a globally exact definition of similarity is hard to come by since depending on the method used to compare items, various types of similarity may be identified. According to Walenstein et al. (2006), a workable common understanding for software similarity is as follows: "the degree to which two distinct programs are similar is related to how precisely they are alike."

In the context of open source software, 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 (McMillan et al. 2012). Given a software system being developed, we are interested in *finding a set of similar OSS projects*

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

with respect to different criteria, such as external dependencies, application domain, or API usage (Nguyen et al. 2018b, c). This type of recommendation is beneficial to the development since it allows developer to learn how similar projects are implemented. Understanding the similarities among open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations (Schafer et al. 2007; Zhang et al. 2017). To aim for software quality (Spinellis and Szyperski 2004), developers normally build their projects by learning from mature OSS projects having comparable functionalities (Spinellis and Szyperski 2004). Furthermore, similarity has been used as a base by both content-based and collaborative-filtering recommender systems to choose the most suitable items for a given user (Schafer et al. 2007). In this sense, it is necessary to equip software developers with suitable machinery which facilitates the similarity search process.

Nevertheless, measuring similarities among software systems has been considered as a daunting task (Chen et al. 2015; McMillan et al. 2012). 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 this sense, choosing the right technique to compute software/projects similarity is a question that may arise at any time.

Goal of the paper The purpose of this paper is twofold. First, we propose CROSSSIM, an approach that allows us to represent different project characteristics belonging to different abstraction layers in a homogeneous manner, then SimRank (Jeh and Widom 2002), a graph algorithm is applied to compute the similarities among nodes. Second, we present a thorough literature review on various techniques for computing software similarities. In this sense, our paper has the following contributions:

– proposing a novel approach to represent the OSS ecosystem by exploiting its mutual relationships;
– developing an extensible and flexible framework for computing similarities among open source software projects; and
– validating the performance of our proposed framework by means of three different existing approaches to computing software similarities, namely MUDABLUE (Garg et al. 2004), CLAN (McMillan et al. 2012), and REPOPAL (Zhang et al. 2017);
– presenting a comprehensive literature review on the field of software similarity measurement.

Structure of the paper The paper is structured into the following sections. Section 2 motivates our work by providing an introduction to three approaches for detecting similar software applications and open source projects. Section 3 brings in our proposed approach for computing similarities between OSS projects. An evaluation on a real GitHub dataset is described in Section 4. Afterwards, Section 5 presents the experimental results in detail. In Section 6, we review the most notable approaches for detecting similar software applications and open source projects. Finally, Section 7 concludes the paper and draws some perspective work.

2 Background

Having access to similar software projects is beneficial to the development process. By looking at a similar OSS project, developers learn how relevant classes are implemented, and in some certain extent, to reuse useful source code (Schafer et al. 2007; Zhang et al. 2017). Also, recommender systems rely heavily on similarity metrics to suggest suitable and meaningful items for a given item (Di Noia et al. 2012; Schafer et al. 2007; Thung et al. 2013). As a result, similarity computation among software and projects has attracted considerable interest from many research groups. In recent years, several approaches have been proposed to solve the problem of software similarity computation. Many of them deal with similarity for software systems, others are designed for computing similarities among open source software projects. Depending on the set of mined features, there are two main types of software similarity computation techniques (Chen et al. 2015):

– *Low-level similarity*: it is calculated by considering low-level data, e.g., source code, byte code, function calls, and API reference,
– *High-level similarity*: it is based on the metadata of the analyzed projects, e.g., similarities in readme files, textual descriptions, and star events. Source code is not taken into account.

This classification is used throughout this paper as a means to distinguish existing approaches with regard to the input information used for similarity computation. In this section, we provide a summary of three techniques for computing similarities among open source projects, i.e., MUDABLUE (Garg et al. 2004), CLAN (McMillan et al. 2012), and REPOPAL (Zhang et al. 2017).

MUDABlue Together with a tool for automatically categorizing open source repositories, Garg et al. (2004) 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. With the application of Latent Semantic Analysis (LSA) (Landauer 2006), software is considered as a document and each identifier is considered as 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, MUDABLUE works in the following steps to compute similarities between software systems:

(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. The outcomes of the evaluation were compared against two existing approaches, namely GURU (Maarek et al. 1991), and the SVM-based method by Ugurel et al. (2002). The evaluation shows that MUDABLUE outperforms these observed algorithms with respect to precision and recall.

CLAN McMillan et al. propose CLAN, an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to package class hierarchies (McMillan et al. 2012). CLAN works based on the document framework for computing similarity, semantic anchors, e.g., those that define the documents' 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.

Using a complete software application as input, CLAN represents source code files as a term-document matrix (TDM). A TDM is used to store the features of a set of document and it is a matrix where a row corresponds to a document and a column represents a term (Collobert et al. 2011). Each cell in the matrix is the frequency that the corresponding term appears in the document. By CLAN, 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 8000 SourceForge[2] applications and shows that it qualifies for the detection of similar applications (McMillan et al. 2012).

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 MUDABLUE as it considers only API calls to represent software systems. As shown later in this paper, CLAN is more efficient than MUDABLUE as it produces recommendations in a much shorter time.

RepoPal In contrast to many previous studies that are generally based on source code (Garg et al. 2004; Liu et al. 2006; McMillan et al. 2012), RepoPal (Zhang et al. 2017) is a high-level similarity metric and takes only repositories metadata as its input. With this approach, two GitHub[3] 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: *(i)* f_i and f_j are the readme files with t being the set of terms in the files; *(ii)* $U(r_i)$ and $U(r_j)$ are the set of users who starred r_i and r_j, respectively; and *(iii)* $R(u_k)$ is the set of repositories that user u_k already starred. There are three similarity indices as follows:

Readme-based similarity The similarity between two readme files is calculated as the cosine similarity between their feature vectors f_i and f_j:

$$\mathrm{sim}_f(r_i, r_j) = \mathrm{CosineSim}(f_i, f_j) \tag{1}$$

Stargazer-based similarity The similarity between a pair of users u_k and u_l is defined as the Jaccard index (Jaccard 1912) of the sets of repositories that u_k and u_l have already starred: $\mathrm{sim}_u(u_k, u_l) = \mathrm{Jaccard}(R(u_k), R(u_l))$. The star-based similarity between two

[2]SourceForge: https://sourceforge.net/
[3]About GitHub: https://github.com/about

repositories r_i and r_j is the average similarity score of all pairs of users who already starred r_i and r_j:

$$\mathrm{sim}_s(r_i, r_j) = \frac{1}{|U(r_i)| \cdot |U(r_j)|} \sum_{\substack{u_k \in U(r_i) \\ u_l \in U(r_j)}} \mathrm{sim}_u(u_k, u_l) \tag{2}$$

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:

$$\mathrm{sim}_t(r_i, r_j) = \frac{1}{|U(r_i) \cap U(r_j)|} \sum_{u_k \in U(r_i) \cap U(r_j)} \frac{1}{|T(u_k, r_i, r_j)|} \tag{3}$$

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

$$\mathrm{sim}(r_i, r_j) = \mathrm{sim}_f(r_i, r_j) \times \mathrm{sim}_s(r_i, r_j) \times \mathrm{sim}_t(r_i, r_j) \tag{4}$$

RepoPal has been evaluated against CLAN using a dataset of 1000 Java repositories (Zhang et al. 2017). 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.

The abovementioned approaches are either low-level or high-level similarity. It is evident that each of these similarity tools is able to manage a certain set of features. Thus, they can only be applied in *prescribed contexts* and cannot exploit additional information when this is available for similarity computation. We assume that combining various input information in computing similarities is highly beneficial to the context of OSS repositories. In other words, the ability to compute software similarity in a *flexible* manner is of highly importance. For instance, in the context of the CROSSMINER project, the required project similarity technique should be flexible enough to enable the development of different types of recommendations as introduced in Section 1. Thus, we expect a tool being capable of incorporating new features into the similarity computation without the need of modifying its internal design. To this end, we anticipate a representation model that integrates semantic relationships among various artifacts. The model should be able to consider implicit semantic relationships and intrinsic dependencies among different users, repositories, and source code by enabling similarity applications in different applicative scenarios.

In the next section, we propose a novel approach that attempts to effectively exploit the rich metadata infrastructure provided by the OSS ecosystem to compute software similarities. To validate the performance of the proposed approach, we conduct a thorough evaluation on a real dataset collected from GitHub and we compare our tool with the three similarity metrics introduced above.

3 CrossSim: a novel approach for computing similarities among GitHub repositories

Based on the observations in Section 2, we come to the conclusion that a representation model that incorporates various features and semantic relationships is highly beneficial to similarity computation. We find inspiration from a related field, namely Linked Data and Semantic Web (Bizer et al. 2009), to realize such a model. Linked Data is a representation method that allows for the interlinking and semantic querying of data. The proliferation of Linked Data in recent years has enabled numerous applications. Two prominent examples are Linked Data for building music platform as by BBC Music (Kobilarov et al. 2009) and for developing map application as by OpenStreetMap (Stadler et al. 2012). The core of Linked Data is an RDF[4] graph that is made up several nodes and oriented links to represent the semantic relationships among various artifacts. Thanks to this feature, the representation paves the way for various computations. One of the main applications of RDF is similarity computation for supporting recommender systems (Di Noia et al. 2012).

By considering the analogy of typical applications of RDF graphs and the problem of detecting the similarity of open source projects, we developed CrossSim (**C**ross project **R**elationships for computing **O**pen **S**ource **S**oftware **Sim**ilarity) (Nguyen et al. 2018c), an approach that makes use of graphs for modeling different types of relationships in the OSS ecosystem (Nguyen et al. 2018b). Similar to RDF graphs, the representation model can capture the semantic features and considers the intrinsic connections between various actors. Specifically, the graph model has been chosen since it allows for flexible data integration and facilitates numerous similarity metrics (Blondel et al. 2004). We consider the community of developers together with OSS projects, libraries, source code, etc., and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependencies and implications 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. The graph representation allows for the computation of similarities among nodes by means of several graph algorithms.

The architecture of CrossSim is depicted in Fig. 1. In particular, the approach imports project data from existing OSS repositories ① and represents them into a graph-based representation by means of the *OSS Ecosystem Representor* module ②. Depending on the considered repository (and thus to the information that is available for each project), the 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 Calculator* module ④, depending on the similarity function to be applied, computes similarity on the source graph-based representation of the input ecosystems to generate matrices ⑤ representing the similarity value for each pair of input projects. A detailed description of the proposed graph-based representation of open source projects is given in Section 3.1. Details about the implemented similarity algorithm are given in Section 3.2.

3.1 A knowledge graph for the OSS ecosystem

By means of the graph representation, we transform the relationships among various artifacts in the OSS ecosystem into a mathematically computable format. The representation model considers different artifacts in a united fashion by taking into account their mutual, both direct and indirect, relationships as well as their co-occurrence as a whole.

The following relationships are used to build graphs representing the OSS ecosystems and eventually to calculate similarity exploiting the algorithm presented in the next section.

[4]https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

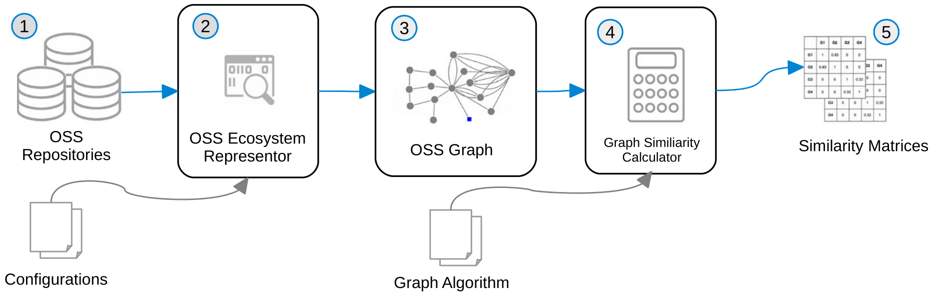


Fig. 1 Overview of the CROSSSIM approach

These vocabularies can also be flexibly augmented upon additional features of input data, even though our current definition seems to cover the most prevailing relationships (Nguyen et al. 2018a).

- *commits* $\subseteq$ *Developer* $\times$ *Project*: this relationship represents the projects or libraries that a user contributes to the development;
- *contains* $\subseteq$ *File* $\times$ *API*: a source file or a communication post with code snippets containing an API function from a third-party library;
- *extends* $\subseteq$ *Class* $\times$ *Class*: a class inherits an abstract class. Two classes extend a same abstract class have a bond since they share a certain number of common functionalities;
- *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*: an OSS project contains a source file;
- *invokes* $\subseteq$ *Class* $\times$ *API*: this is the case when a class calls an API function from a third-party library. API calls can be extracted from source files using suitable code parsers;

In the scope of this paper, we concentrate on studying the performance of CROSSSIM by exploiting high-level information, i.e., *isUsedBy*, *develops*, and *stars*, which are described as follows:

- *isUsedBy* $\subseteq$ *Library* $\times$ *Project*: a project includes a third-party library to make use of the library's functionalities. For the sake of presentation, in the rest of this paper, *library* and *dependency* are used interchangeably to indicate a third-party library;
- *develops* $\subseteq$ *Developer* $\times$ *Class*: a developer contributes to the development of a class in a software project;
- *stars* $\subseteq$ *User* $\times$ *Project*: it represents projects that a given user has starred. This relationship is only applicable to GitHub.

Figure 2 depicts an example of the graph representation for various OSS artifacts. There are several semantic edges to describe the mutual relationship between graph nodes. For example, the edge *includes* describes the relationship between a project and a third-party library, whereas the edge *hasSourceCode* dictates that a project contains a source code file. The graph structure facilitates similarity computation (Blondel et al. 2004). For instance, several existing algorithms are able to compute the similarity between *project#1* and *project#2* as they are indirectly connected by the pair of edges, i.e., *hasSourceCode* and *implements* (Jeh and Widom 2002; Nguyen et al. 2018c). This semantic path reflects the actual relevance of the projects, they contain classes that implement a common interface.

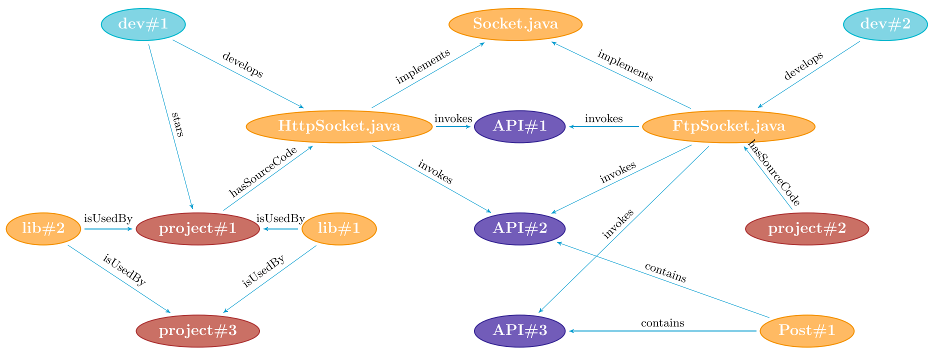


Fig. 2 A Knowledge Graph for the representation of the OSS ecosystem

The similarity is further enforced by another path via *hasSourceCode* and *invokes*, leading to two API functions, i.e., *API#1* and *API#2*. The two projects are a bit more similar since they invoke same APIs. Analogously, the similarity between *project#1* and *project#3* can also be inferred since they both include two third-party libraries *lib#1* and *lib#2*, and projects share similar libraries are considered to be similar (McMillan et al. 2012).

The graph structure allows for similarity computation on different artifacts. For instance, it is possible to compute similarities among developers: we see that developers *dev#1* and *dev#2* share a common activity, they develop two classes, i.e., *HttpSocket.java* and *FtpSocket.java* and these classes are somehow similar. In particular, both *HttpSocket.java* and *FtpSocket.java* implement *Socket.java*; furthermore, they all invoke *API#1* and *API#2* in their code.

To understand how to incorporate existing APIs into current code, a developer normally looks for API documentations that describe the constituent functions. For instance, StackOverflow provides the developer with a broader insight of API usage, and in some cases, with sound code examples (Baltes et al. 2018; Ponzanelli et al. 2014). In Fig. 2, there is a StackOverflow post, i.e., *Post#1* contains code snippets with two function calls *API#2* and *API#3*. In practice, this is a typical scenario when users discuss the usage of libraries containing *API#2* and *API#3*. In this respect, it might be helpful if we recommend *Post#1* to the developer of class *FtpSocket.java*. This is completely feasible since the graph structure allows one to compute the similarity between *Post#1* and *FtpSocket.java*. In the end, a recommendation engine can provide the developer with a list of StackOverflow posts that are relevant to the code being developed.

The graph structure allows for the integration of different relationships as depicted in Fig. 2. CrossSim has been designed as a general framework to compute software similarity by exploiting both low-level and high-level features.

3.2 SimRank: computing graph similarity

Graph similarity is an active field and receives a significant attention from the research community. Computing similarity among graph nodes has been applied to solve different problems in Computer Science. For instance, graph algorithms have been used to measure the similarities among social network nodes (Nassar et al. 2018; Wang et al. 2015b), proteins (Kollias et al. 2014), RDF graph nodes (Di Noia et al. 2012), to name a few.

In this paper, we apply graph similarity to solve the problem of computing the similarities among various OSS projects. First, we recall some notations as follows. 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. A graph consists of nodes and oriented links with semantic relationships (Bizer et al. 2009). A triple <*subject*, *predicate*, *object*> with (*subject*, *object* ∈ V) and *predicate* ∈ E states that node *subject* is connected to node *object* by means of the edge labeled 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 (Di Noia et al. 2012). Among others, feature-based semantic similarity metrics gauge the similarity between graph nodes as a measure of commonality and distinction of their hallmarks. By feature-based similarity, objects are represented as a set of common and distinctive features and the similarity between two objects is computed by comparing their features (Tversky 1977).

Many semantic similarity metrics first attempt to characterize resources in graphs representing sets of features and then perform similarity calculation on them. SimRank has been developed to calculate similarities based on mutual relationships between graph nodes (Jeh and Widom 2002). Considering two nodes, the more similar nodes point to them, the more similar the two nodes are. We take an example in Fig. 3 to illustrate how SimRank works in practice. There, node 1 is similar to node 2 since both are pointed by node 5. Comparably, node 3 is similar to node 4 as they are pointed by node 6. As a result, the two nodes α and β are highly similar because they are concurrently pointed by other four nodes in the graph, i.e., 1, 2, 3, and 4, considering that 1 and 2 as well as 3 and 4 are pairwise similar. In this sense, the similarity between α and β is computed by using a fixed-point function, taking into consideration the accumulative similarity by their pointing nodes. Given $k \geq 0$ we have $R^{(k)}(\alpha, \beta) = 1$ with $\alpha = \beta$ and $R^{(k)}(\alpha, \beta) = 0$ with $k = 0$ and $\alpha \neq \beta$, SimRank is computed as follows (Jeh and Widom 2002):

$$R^{(k+1)}(\alpha, \beta) = \frac{\Delta}{|I(\alpha)| \cdot |I(\beta)|} \sum_{i=1}^{|I(\alpha)|} \sum_{j=1}^{|I(\beta)|} R^{(k)}(I_i(\alpha), I_j(\beta)) \quad (5)$$

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 forcing $R^{(k)}(\alpha, \beta) \in [0, 1]$.

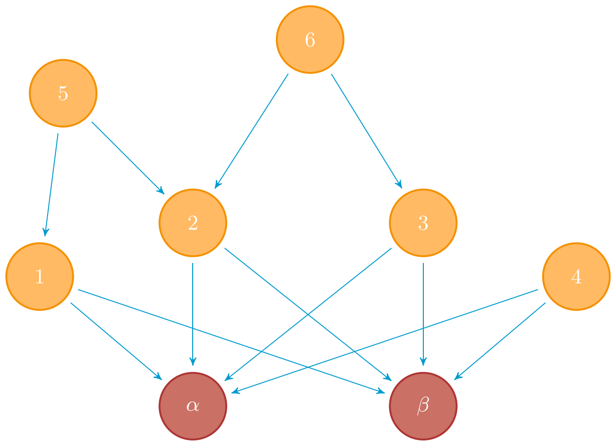


Fig. 3 An example of how SimRank works

By using the graph representation as in Section 3.1, we are able to transform the OSS ecosystem into a mathematically computable format. This graph structure allows for the application of various similarity algorithms. In CrossSim, we adopt SimRank as the mechanism for computing similarities among OSS graph nodes. However, other similarity algorithms can also be flexibly integrated into CrossSim, as long as they are designed for graphs (Nguyen et al. 2015). The utilization of SimRank is convenient and practical also when various relationships are incorporated into the graph. Given the circumstances, the algorithm does not need to be changed since it only works on the basis of nodes and edges. In this sense, CrossSim is a versatile similarity tool as it can accept various input features regardless of their format.

To study the performance of CrossSim, we conducted a comprehensive evaluation using a dataset collected from GitHub. To aim for an unbiased comparison, we opted for existing evaluation methodologies from other studies of the same type (Lo et al. 2012; McMillan et al. 2012; Zhang et al. 2017). Together with other metrics typically used for evaluations, i.e., Success rate, Confidence, and Precision, we decided to use also Ranking to measure the sensitivity of the similarity tools to ranking results. The details of our evaluation are given in the next section.

4 Evaluation

In this section, we describe the process that has been conceived and applied to evaluate the performance of CrossSim compared with some baselines. We opt for MUDABlue, CLAN, and RepoPal to compare with CrossSim. The rationale behind the selection of these approaches is that they are well-established algorithms and have demonstrated their effectiveness in various settings. According to Zhang et al. (2017), by applying the same experiment settings and evaluating on the same dataset, the authors demonstrated that RepoPal outperforms CLAN in terms of Confidence and Precision. Meanwhile, CLAN has a better performance than that of MUDABlue, also with respect to *Confidence* and *Precision* (McMillan et al. 2012). Furthermore, RepoPal works on GitHub Java repositories containing rich metadata that is suitable for building graph by CrossSim. Intuitively, we consider all these tools as a good starting point for a performance comparison. Furthermore, our evaluation aims at comparing two low-level similarity tools, i.e., MUDABlue and CLAN with two high-level similarity ones, i.e., RepoPal and CrossSim.

The evaluation process that has been applied is shown in Fig. 4 and consists of activities and artifacts that are going to be explained later on this section. In particular, a set of Java projects (see Section 4.1) 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. The outcomes are then analyzed using various quality metrics.

Since the implementations of the baselines are no longer available for public use, we reimplemented MUDABlue, CLAN, and RepoPal by strictly following the descriptions in the original papers (Garg et al. 2004; McMillan et al. 2012; Zhang et al. 2017). Such implementations, including the CrossSim one, are available online in GitHub (Nguyen et al. 2018d) to facilitate future research.

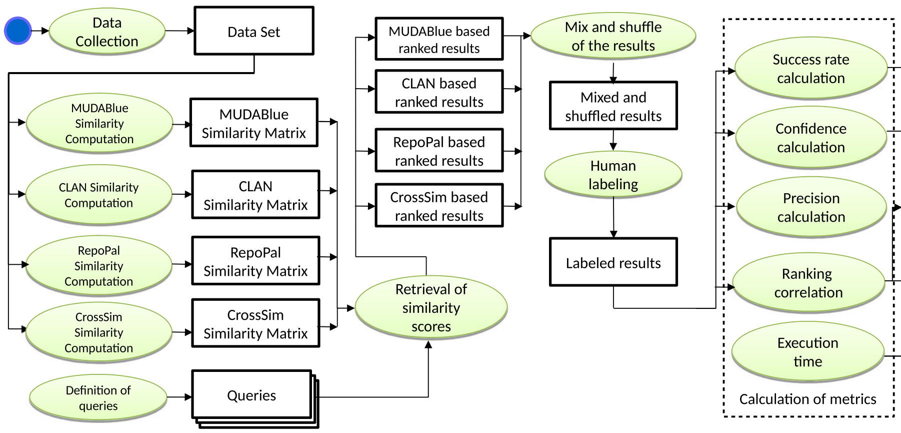

Data Collection
Data Set
MUDABlue Similarity Computation
MUDABlue Similarity Matrix
CLAN Similarity Computation
CLAN Similarity Matrix
RepoPal Similarity Computation
RepoPal Similarity Matrix
CrossSim Similarity Computation
CrossSim Similarity Matrix
Definition of queries
Queries
Retrieval of similarity scores
MUDABlue based ranked results
CLAN based ranked results
RepoPal based ranked results
CrossSim based ranked results
Mix and shuffle of the results
Mixed and shuffled results
Human labeling
Labeled results
Success rate calculation
Confidence calculation
Precision calculation
Ranking correlation
Execution time
Calculation of metrics

Fig. 4 Evaluation process

4.1 Dataset

To compare the performance of CROSSSIM with those of the baselines, it is necessary to execute them on the same set of OSS projects. The collected dataset needs to be suitable as input for all four similarity engines. By MUDABLUE and CLAN, there are no specific requirements since both tools rely solely on source code to function.

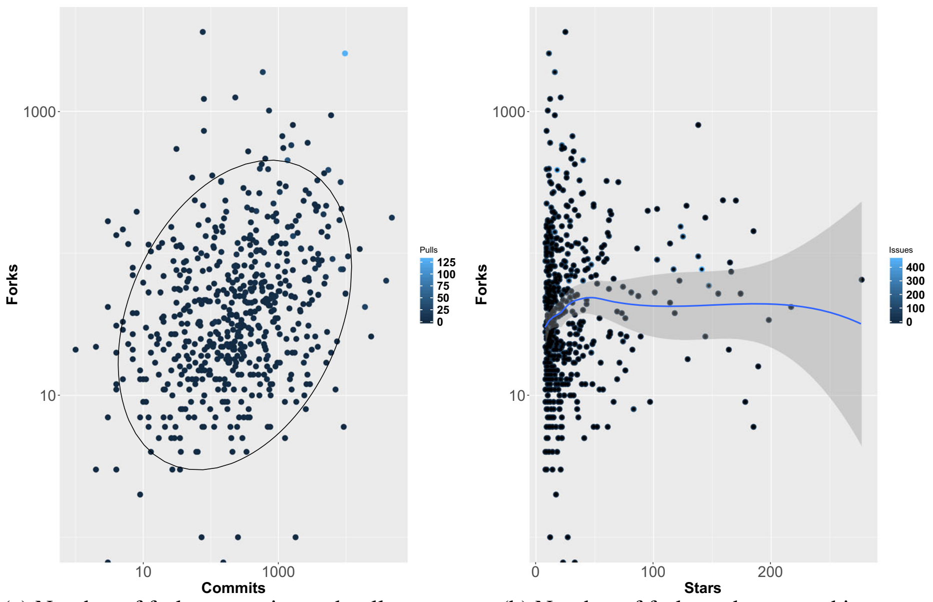
For REPOPAL and CROSSSIM, we can consider only projects that satisfy certain criteria. In particular, the collected projects have to meet the following requirements:

– Providing the specification of their dependencies by means of `code.xml` or `.gradle` files;[5]
– 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—this is needed to enable the application of RepoPal;
– Being starred by at least 20 users as required by REPOPAL to work.

Furthermore, we realized that the final outcomes of a similarity algorithm have 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 manually 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, we crawled projects to provide input for the evaluation. Though the number of projects that fulfill the requirements of a single approach is high, the number of projects that meet the requirements of all approaches is considerably lower. For example, a project contains both `pom.xml` and `README.md`, albeit having only 5 dependencies,

[5]The files `pom.xml` and with the extension `.gradle` are related to management of dependencies by means of Maven (https://maven.apache.org/) and Gradle (https://gradle.org/), respectively.



(a) Number of forks, commits, and pull requests

(b) Number of forks and stars, and issues

Fig. 5 The projects and their number of forks, commits, pull requests, stars, and issues

does not meet the constraints and must be discarded. The scraping is time consuming as for each project, at least 6 queries must be sent to get the relevant data. As a matter of fact, GitHub already sets a rate limit for an ordinary account[6], with a total number of 5000 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 is published together with all tools for public usage (Nguyen et al. 2018d).

Figure 5 a and b provide a summary on the projects and the number of forks, commits, stars, pull requests, and issues. The number of pull requests for most of the projects is considerably low, i.e., lower than 100; however, their number of forks and commits is high. Forking is a means to contribute to the original repositories (Jiang et al. 2017). Furthermore, there is a strong correlation between forks and stars (Borges et al. 2016), as it is further witnessed in Fig. 5b. A project with a high number of forks means that it can be considered as a sign of a well-maintained and well-received project. Similarly, as commits have a significant influence on the source code (Behnamghader et al. 2017), the number of commits is also a good indicator of how a project has been developed.

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 randomly added 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 1. This is an approximate classification since a project might belong to more than one category.

[6]GitHub Rate Limit: https://developer.github.com/v3/rate_limit/

Table 1 List of software categories

No.	Name	No. 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

As can be seen in Table 1, among 580 considered projects, 309 of them belong to some specific categories, such as SPARQL, RDF, Jena Apache, Selenium Test, Elastic Search, and Spring MVC. 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 full list of the categories in this paper.

4.2 Similarity computation

To explain how the graph representation is exploited in CrossSim, Fig. 6 sketches the sub-graph for representing the relationships between two projects AskNowQA/AutoSPARQL and AKSW/SPARQL2NL. The orange nodes are dependencies and their real names are depicted in Table 2. The turquoise nodes are developers who already starred the repositories. Every node is encoded using a unique number across the whole graph. To compute similarity

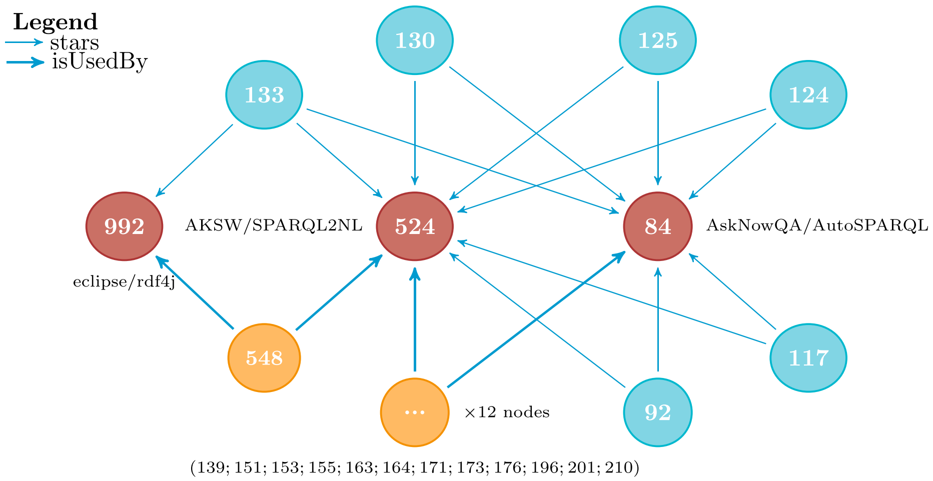


Fig. 6 Sub-graph showing a fragment of the representation for three projects, i.e., AskNowQA/AutoSPARQL, AKSW/SPARQL2NL, and eclipse/rdf4j

Table 2 Shared dependencies in Fig. 6

ID	Name
139	org.apache.jena:jena-arq
151	org.dllearner:components-core
153	net.didion.jwnl:jwnl
155	net.sourceforge.owlapi:owlapi-distribution
163	net.sf.jopt-simple:jopt-simple
164	jaws:core
171	com.aliasi:lingpipe
173	org.dllearner:components-ext
176	org.apache.opennlp:opennlp-tools
196	org.apache.solr:solr-solrj
201	org.apache.commons:commons-lang3
210	javax.servlet:servlet-api
548	org.slf4j:log4j-over-slf4j

between the two projects, SimRank is applied following (5) and the damping factor Δ is empirically set to 0.85 according to some existing studies (Jeh and Widom 2002; Nguyen et al. 2015). The final result that lies between 0 and 1 is the similarity between two projects AskNowQA/AutoSPARQL and AKSW/SPARQL2NL.

In the evaluation, we also investigate the effect of highly frequent libraries on the prediction performance. By analyzing the dataset in Section 4.1 and counting the number of all libraries, we obtained a ranked list of libraries, sorted according to the frequency of occurrence in the projects. Table 3 depicts the list of the six most frequent libraries in the dataset. For example, **junit:unit** is the most popular library and it is included by 447 projects. The second item on the list is **org.slf4j:slf4j-api** whose frequency is much lower, it is found in 217 projects. The least popular one among the libraries in Table 3 is **org.slf4j:slf4j-log4j12**, and it appears in 129 projects.

Query definition Among 580 projects in the dataset, 50 have been selected as queries. We did not sample them randomly but selected those coming from some specific categories. This is due to the fact that the dataset is rather small and if we randomly selected queries which do not belong to any categories, we may end up retrieving irrelevant projects, and this is not useful for the validation process. We assume that the random selection can be done only when more projects available for training. 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 as listed in Table 4.

Table 3 Most frequent dependencies in the considered dataset

Dependency	Frequency
junit:junit	447
org.slf4j:slf4j-api	217
com.google.guava:guava	171
log4j:log4j	156
commons-io:commons-io	151
org.slf4j:slf4j-log4j12	129

Table 4 List of queries for evaluation (Nguyen et al. 2018d)

No.	Project name	No.	Project 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	pranab/beymani	30	huangchen007/elasticsearch-rest-command
6	sayems/java.webdriver	31	pitchpoint-solutions/sfs
7	psaravan/JamsMusicPlayer	32	javanna/elasticsearch-river-solr
8	webdriverextensions/webdriverextensions	33	mesos/hadoop
9	dadoonet/spring-elasticsearch	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	pyvandenbussche/sparqles
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	testIT-WebTester/webtester-core	46	jt6211/hadoop-dns-mining
22	elastic/elasticsearch-metrics-reporter-java	47	xebia/Xebium
23	elastic/elasticsearch-support-diagnostics	48	TheAndroidMaster/Pasta-Music
24	SpringDataElasticsearchDevs/spring-data-elasticsearch	49	SubstanceMobile/GEM
25	javanna/elasticshell	50	markzhai/LyricHere

Retrieval of similarity scores Our evaluation has been conducted in line with some other existing studies (Lo et al. 2012; McMillan et al. 2012; Zhang et al. 2017). In particular, for each query in the set of the 50 projects defined in the previous step, similarity is computed against all the remaining projects in the dataset using the SimRank algorithm discussed in Section 3.2. From the retrieved projects, only top 5 are selected for the subsequent evaluation steps. For every query, similarity is also computed using MUDABLUE, CLAN, and REPOPAL to get the top-5 most similar retrieved projects.

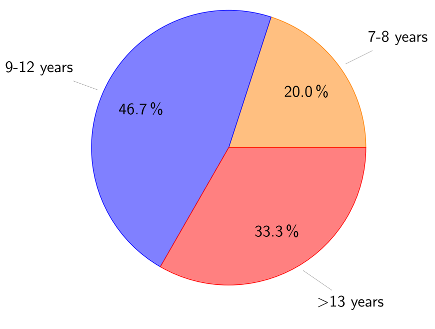
4.3 User evaluation

For each similarity tool, the outcomes of the computation are a ranked list of similar projects. It is necessary to evaluate how relevant the projects are, compared with the query project. Since a user evaluation is the only way to evaluate the outcome (McMillan et al. 2012; Zhang et al. 2017), we involved a group of 15 software developers to participate in the manual evaluation. Some of the participants are master students, and most of them work as software developers or researchers in academic and industry. Before the evaluation, we sent each evaluator a tutorial on how to conduct the scoring process. Furthermore, to get information about the participants related to their development background, we sent them a questionnaire similar to the one presented in CLAN evaluation dataset (2018) and McMillan et al. (2012). According to the survey, all the participants are capable of at least 2 different programming languages, and their favorite code platform is StackOverflow, where they normally search for posts that are useful for their current development tasks. In addition, most of them tend to re-use code fragments collected from external sources quite often.

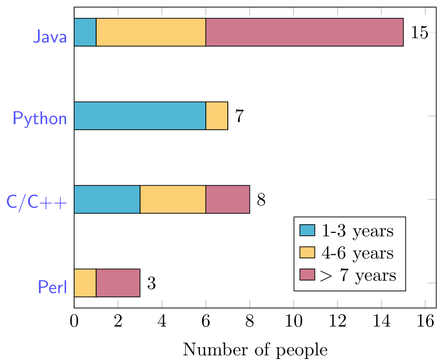
Figure 7 a depicts the number of years that the developers have spent for software development activities. All the participants have at least 7 years of programming experience, two of them have more than 20 years. In Fig. 7b, we show the number of people and their corresponding number of years of programming experience for different languages. Among others, Java is the programming language that all developers are knowledgeable about, with at least 2 years of experience. Nine of the developers have spent more than 7 years working with Java. This is highly advantageous for our user evaluation since all projects included in the dataset introduced in Section 4.1 are written in Java, and we assume that skillful developers shall have a better judgment about the similarities among projects. The knowledge of different programming languages, i.e., Perl, Python, C/C++, is also a plus for the evaluation process.

By the user evaluation, in order to have a fair evaluation, for each query, we mixed and shuffled the top-5 results generated from the computation by each similarity metric in a single Google form and presented them to the evaluators who then inspected and given a score to every pair. 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 (Ghose and Lowengart 2001; Pettigrew and Charters 2008). This aims at eliminating any bias or prejudice against a specific similarity metric. In particular, given a query, a manual labeling process is performed to evaluate the similarity between the query and the corresponding retrieved projects. The participants are asked to label the similarity for each pair of projects (i.e., <*query, retrieved project*>) with regard to their application domains and functionalities using the scales listed in Table 5 (McMillan et al. 2012).

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) People and years of experience



(b) Languages and years of experience

Fig. 7 **a**, **b** A summary of the participants' software development experience

Table 5 Similarity scales

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

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* or *Highly similar*. A good similarity approach should produce as much true positives as possible.

To aim for a reliable comparison, we followed the same procedures utilized by related work. For overlapping pairs, i.e., those that appear in the ranked lists of two or more algorithms, we chose just one of them and presented it to one of the participants. This aims to avoid having one pair with different scores. Furthermore, the labeling results by a participant were then double-checked by another one to aim for soundness of the outcomes. By carefully investigating the results, we realized that in most cases, the evaluators agree on the evaluation scores. In case there is any disagreement about a final score between any two evaluators, a senior researcher is involved to inspect the pair again to reach a consensus.

4.4 Evaluation metrics

To evaluate the outcomes of the algorithms with respect to the user evaluation, the following metrics have been considered as typically done in related work (Lo et al. 2012; McMillan et al. 2012; Zhang et al. 2017):

- *Success rate*: if at least one of the top-5 retrieved projects is labeled *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 labeled as *Similar* or *Highly similar* by humans.

Further than the previous metrics, we propose 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 analyzed 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 (Spearman 1904) 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 follows: $r_s = 1 - \frac{6\sum_{i=1}^{n}(\rho_i - \sigma_i)^2}{n(n^2-1)}$. Because of the large number of ties, we also used *Kendall's tau* (1938) coefficient, which is used to measure the ordinal association between two considered quantities. Both r_s and τ range from -1 (perfect negative correlation) to +1 (perfect positive correlation); $r_s = 0$ or $\tau = 0$ implies that the two variables are not correlated.

Finally, we consider also the *execution time* related to the application of the four approaches on the dataset to obtain the corresponding similarity matrices.

4.5 Research questions

To study the performance of the considered tools in detecting similar projects for the set of queries, the following research questions are considered:

– ***RQ*$_1$*: Which graph configuration brings the best performance to* CROSSSIM?** Our proposed approach allows for a flexible computation by incorporating different features in a graph. We investigate which types of edges sustain similarity computation by considering different test configurations. In this way, we identify the configuration that fosters the best prediction outcome for CROSSSIM.
– ***RQ*$_2$*: Which similarity approach between* MUDABLUE, CLAN, REPOPAL, *and* CROSSSIM *yields a better performance in terms of Success rate, Confidence, Precision, and Ranking?*** By this question, we compare the performance of the approaches concerning the ability to produce accurate recommendations. In the context of software development, providing relevant results is of highly importance since a developer would expect a set of similar projects to the project being developed.
– ***RQ*$_3$*: Which similarity approach is more efficient with respect to execution time?*** An important factor for a similarity tool is the ability to compute within an acceptable amount of time. This research question aims at measuring the time needed for a tool to produce a final recommendation.

5 Experimental results

In Section 5.1, the data that has been obtained as discussed in the previous section is analyzed to answer the research questions, i.e., **RQ**$_1$, **RQ**$_2$, and **RQ**$_3$. Afterwards, Section 5.2 presents discussions related to the experimental outcomes. Finally, threats to the validity of the evaluation are also discussed in Section 5.3.

5.1 Data analysis

***RQ*$_1$*: Which graph configuration brings the best performance to* CROSSSIM?**

We investigate the implication of graph structure on the outcome of CROSSSIM by considering various types of edges. This aims at identifying the set of features that contribute to the performance gain. First, only star events are used to build the graph with the *stars* edges

Table 6 CrossSim test configurations

Configuration	Stars	isUsedBy	Develops	Fre. deps.
CrossSim$_1$	✓	×	×	×
CrossSim$_2$	×	✓	×	×
CrossSim$_3$	✓	✓	×	×
CrossSim$_4$	✓	✓	✓	×
CrossSim$_5$	✓	✓	×	✓
CrossSim$_6$	✓	✓	✓	✓

(see Fig. 2), and this configuration is called CrossSim$_1$. Correspondingly, in CrossSim$_2$, the graph is built with only the *isUsedBy* edges by using only dependencies. By CrossSim$_3$, we consider both the relationships *stars* and *isUsedBy* together to compute similarity. Afterwards, we extend CrossSim$_3$ by incorporating also committers, i.e., the *develops* relationship, and this yields configuration CrossSim$_4$. Similarly, with CrossSim$_5$, we investigate the effect of frequent dependencies by adding them to CrossSim$_3$. Finally, we take into account all the abovementioned edges, resulting in CrossSim$_6$. Table 6 gives a detailed description of the test configurations used to internally compare CrossSim.

Since the manual evaluation is a time-consuming process, we decided to exploit only a subset of the queries to address this research question. In particular, we selected the first 20 queries in Table 4, i.e., from number 1 to 20 and provided as input for CrossSim and the results are depicted in Table 7.

Among the configurations, CrossSim$_2$ gains the lowest prediction performance. In particular, it gets 0.90 as success rate and 0.51 as precision. This implies that using only dependencies as features does not contribute to a good performance. Compared with CrossSim$_2$, CrossSim$_1$ has a slightly better performance as its success rate and precision are 0.95 and 0.61, respectively. By referring back to the RepoPal approach (see Section 2), where information related to the star event such as stargazers and star time gap is used to compute similarity, we come to the conclusion that stars are useful for the detection of similar GitHub repositories. However, we assume that more investigations are needed to understand better the effect of stars on similarity computation. This issue remains as a future work.

We consider CrossSim$_5$ in combination with CrossSim$_6$ to observe the effect of the adoption of committers. According to Table 7, CrossSim$_5$ gains a success rate of 100%, with a precision of 0.75. The number of false positives by CrossSim$_6$ goes up, thereby worsening the overall performance considerably with 0.70 being as the precision. The performance degradation is further witnessed by considering CrossSim$_3$ and CrossSim$_4$ together. Both get 100% as success rate; however, CrossSim$_3$ obtains a better precision, i.e., 0.80 compared with 0.76 by CrossSim$_4$. 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

Table 7 Comparison of different CrossSim configurations

	CrossSim$_1$	CrossSim$_2$	CrossSim$_3$	CrossSim$_4$	CrossSim$_5$	CrossSim$_6$
Success rate (%)	95	90	100	100	100	100
Precision	0.62	0.51	0.80	0.76	0.75	0.70

Table 8 Comparison of the similarity approaches

	MUDABLUE	CLAN	REPOPAL	CROSSSIM3
Success rate (%)	60	60	*100*	*100*
Precision	0.22	0.22	0.71	*0.78*
Execution time (min)	380	22	240	*12*
Spearman's (r_s)	−0.01	−0.08	−0.132	*−0.230*
Kendall's tau (τ)	−0.01	−0.07	−0.108	*−0.214*

Numbers in italic represent higher values

deployment of a weighting scheme for developers may help counteract the degradation in performance.

Next, CROSSSIM3 and CROSSSIM5 are considered together to analyze the effect of the removal of the most frequent dependencies. CROSSSIM3 outperforms CROSSSIM5 as it gains a precision of 0.80, the highest value among all, compared with 0.75 by CROSSSIM5. The removal of the most frequent dependencies helps also improve the performance of CROSSSIM4 in comparison with CROSSSIM6. 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 presented in Section 3.2. 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 as in Table 3, this characteristic may not hold anymore. For example, two projects are pointed by junit:junit because they use JUnit[7] for testing. Since testing is a common functionality of many software projects, it does not help contribute towards the characterization of a project and thus, needs to be removed from the considered graph.

Among the considered experimental configurations, CROSSSIM3, where star events and dependencies are used together to build the graph, obtains the best prediction performance. The graph structure considerably affects the outcome of the similarity computation. In this sense, finding a graph structure that facilitates similarity computation is of paramount importance.

RQ*$_2$*: Which similarity approach between* MUDABLUE, CLAN, REPOPAL, *and* CROSSSIM *yields a better performance in terms of Success rate, Confidence, Precision, and Ranking? For this research question, we compared the best configuration CROSSSIM3 with the baselines. The experimental results are shown in Table 8 and Fig. 8. In particular, Table 8 depicts Success rate, Precision, Execution time, Spearman's (r_s), and Kendall's tau (τ) for all similarity tools. The obtained results demonstrate that REPOPAL is a good choice for computing similarity among OSS projects. Its success rate and precision are superior to those of MUDABLUE and CLAN. Both MUDABLUE and CLAN obtain a success rate of 60%; however, REPOPAL gets a success rate of 100%. Furthermore, the precision of MUDABLUE and CLAN is 0.22 which is considerably lower than 0.71, the corresponding value for REPOPAL.

In comparison with the other tools, CROSSSIM3 has a better performance concerning Success rate and Precision. Both REPOPAL and CROSSSIM3 gain a Success rate of 100%. However, CROSSSIM3 gets 0.78 as precision which is higher than 0.71, the corresponding

[7] JUnit: http://junit.org/junit5/

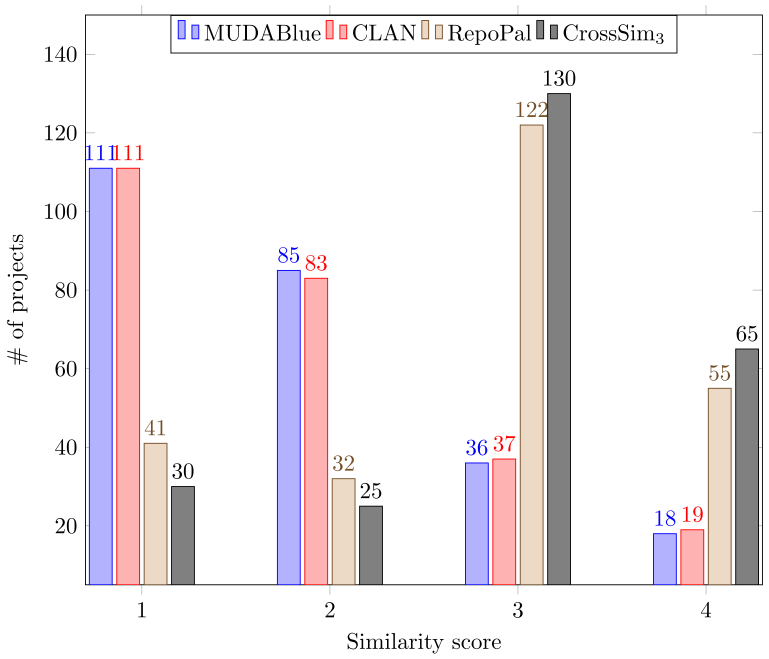


Fig. 8 Confidence

value by RepoPal. In this sense, CrossSim outperforms the baselines with respect to success rate and precision.

The Confidence for the similarity tools is shown in Fig. 8. MUDABlue has more false negatives than CLAN does, i.e., pairs that are scored with 1 or 2. In particular, the number of project pairs that have been assigned the score of 1 is 111 for both MUDABlue and CLAN. Meanwhile, the corresponding number of pairs that have the score of 2 is 85 and 83 for MUDABlue and CLAN, respectively. In addition, CLAN has more true positives, i.e., scores of either 3 or 4. In this sense, we conclude that CLAN has a slightly better performance in comparison with MUDABlue. By this index, i.e., Confidence, CrossSim3 yields a better outcome as it has more scores that are either 3 or 4 and less scores that are 1 or 2. For the similarity score of 3, CrossSim3 finds 130 relevant projects, whereas RepoPal returns 122 projects. The same trend can also be witnessed for the similarity score of 4: the number of projects that RepoPal and CrossSim3 retrieve is 55 and 65, respectively. It is evident that CrossSim3 achieves a better Confidence compared with that of RepoPal.

In addition to the conventional quality indexes, we investigated the ranking produced by the 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 developers, which are already sorted in descending order. In this way, a lower r_s (τ) means a better ranking. Both indexes r_s and τ are computed for all 50 queries and related first five results. The value of r_s is −0.230 for CrossSim3 and −0.132 for RepoPal. The value of τ is −0.214 for CrossSim3 and −0.108 for RepoPal. By this quality index, CrossSim3 performs slightly better than RepoPal. Compared with MUDABlue, CLAN obtains a superior ranking with respect to both r_s and τ. In particular, CLAN obtains −0.08 and −0.07 for r_s and τ, respectively; and MUDABlue gets −0.01 for both r_s and τ.

The experimental results confirm the claim made by the authors of RepoPal (Zhang et al. 2017): The system obtains a better recommendation performance than CLAN in terms of Success rate, Precision, and Confidence. Furthermore, the ranking by RepoPal is also better to those of MUDABlue and CLAN.

Compared to MUDABlue, CLAN, and RepoPal, CrossSim gains a better performance in terms of Success rate, Precision, Confidence, and Ranking. The obtained results confirm our hypothesis that the incorporation of various features, e.g., dependencies and star events into graph facilitates similarity computation. Furthermore, CrossSim is more flexible as it can include other artifacts in similarity computation without affecting the internal design.

RQ_3: Which similarity approach is more efficient with respect to execution time? We measure the time needed to produce recommendations for all the four similarity approaches. The execution time related to the application of the analyzed techniques is shown in the third row of Table 8. For the experiments, a laptop with Intel Core i5-7200U CPU @ 2.50GHz $\times$ 4, 8GB RAM, Ubuntu 16.04 was used. In such a configuration, RepoPal takes $\approx$4 h to generate the similarity matrix, whereas the execution of CrossSim$_3$, including both the time for generating the input graph and that for generating the similarity matrix, takes $\approx$12 min. Such an important time difference is due to the time needed to calculate the similarity between `README.md` files, on which RepoPal relies. MUDABlue takes 380 min to complete the computation, even though most of the time (310 min) is devoted to the creation of the internal matrices needed to perform the similarity calculation. Meanwhile, CLAN needs only 22 min to do the complete computation. Also in this case the creation of the internal structures is the most computational demanding phase, it takes 21 min. Such execution times make evident the distinction between high-level and low-level similarity approaches. In particular, MUDABlue takes into consideration all source code artifacts for computation, such as variable names, method names, the matrix used to represent the input may become huge. CLAN considers only API calls for computation. This is the reason why the computation times for MUDABlue and CLAN are much higher than those of RepoPal and CrossSim that instead consider just project metadata.

CrossSim is the most efficient tool as it generates the similarity matrix in a short time as shown in Table 8.

5.2 Discussions

To aim for a reliable evaluation, the experiments in this paper have been performed in line with existing studies (McMillan et al. 2012; Zhang et al. 2017). As can be seen, CrossSim has a better performance than that of MUDABlue, CLAN, and RepoPal with respect to different quality indicators. The gain in CrossSim's performance demonstrates that the consideration of different artifacts, e.g., projects, libraries in a mutual relationship, instead of individual items brings a substantial benefit. Referring back to the RepoPal paper (Zhang et al. 2017), we see that the success rate of RepoPal in our evaluation is considerably higher than that in the original experiments. This can be explained as follows:

According to our investigation on the dataset considered by REPOPAL,[8] the chosen projects scatter in several categories and the number of projects belonging to certain categories is rather low. That means, the similarities among the projects are low by their origin. However, in our dataset, projects have been deliberately selected so as to converge on some specific categories, thus increasing their mutual similarities. This makes the possibility that a query gets a relevant project which comes from the same category become superior to that by the original REPOPAL dataset. As a result, the success rate in our experiments increases considerably.

In our evaluation, compared with MUDABLUE, CLAN gains a better confidence: it returns fewer false negatives and more true positives. This partly confirms the findings by McMillan et al. in their work (2012). However, both MUDABLUE and CLAN obtain comparable success rate and precision. This is not completely consistent with their claim since they demonstrated that CLAN gained a better performance compared with that of MUDABLUE. Our intuition is that the discrepancy between our work and that of McMillan et al. (2012) may attribute to following reasons: *(i)* although we attempted to strictly follow descriptions by the related papers to implement the tools, the final implementations might not be exactly identical to the original ones. Particularly, the selection of various parameters for the LSA implementation used in MUDABLUE and CLAN may considerably contribute towards the difference; *(ii)* the dataset in our evaluation is different from the one used for evaluating CLAN. Altogether, this should introduce some fluctuations in the performance of both approaches.

Currently, CROSSSIM supports the incorporation of *isUsedBy*, *develops*, and *stars* to compute similarities (Nguyen et al. 2018d), and thus being a high-level similarity metric. However, it is feasible to consider also low-level features, such as package names, class names, or API function calls as partly shown in Fig. 1. In a recent work Nguyen et al. (2019a) and Nguyen et al. (2019b), we exploited CROSSSIM to compute similarities by considering low-level information, i.e., API function calls as features. In this way, CROSSSIM becomes a *hybrid* similarity metric as it deals with both metadata (high-level) and source code (low-level). Furthermore, since CROSSSIM allows for the integration of various graph algorithms for calculating similarity, the deployment of different techniques rather than SimRank might possibly improve the overall recommendation performance, depending on the set of features. To this end, the selection of a proper similarity technique for each type of graphs can be considered as an open research topic.

We assume that the graph structure may have a dramatic influence on its performance. Thus, finding a suitable graph that facilitates the computation is an interesting research topic and needs further investigations. Moreover, since very frequent nodes are not useful for similarity computation, it is necessary to define a threshold at which a node is considered to be frequent.

5.3 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.

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

[8] https://github.com/yunzhang28/RepoPal/blob/master/1000repo.xlsx

15 developers with decent programming experience in the user evaluation. (*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 strictly followed the descriptions in the original papers to re-implement the tools. Furthermore, we have run several manual assessments and counter-checks to validate the evaluation outcomes.

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 all the tools is low and thus required a prolonged scraping. 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.

Construct validity is related to the experimental settings used to evaluate the similarity approaches. We addressed the issue seriously and attempted to simulate a real deployment scenario where the tools are used to search for similar GitHub repositories. In this way, we were able to investigate if the tools are really applicable to authentic usage.

Conclusion validity is whether the exploited experiment methodology is intrinsically related to the obtained outcome, or there are also other factors that have an impact on it. The evaluation metrics, i.e., Success rate, Confidence, Precision, Ranking, and Execution time might cause a threat to conclusion validity. To confront the issue, we employed the same metrics for evaluating all the similarity approaches.

6 Related work

In this section, we review some of the most notable approaches that have been developed to measure the similarity between software systems or OSS projects. These approaches deal with the detection of: *(i)* similar open source applications, *(ii)* similar mobile applications, *(iii)* software plagiarisms and clones, and *(iv)* relevant third-party libraries.

To detect clone among Android apps, Wang et al. propose WuKong (2015a) which employs a two-phase process as follows. The first phase 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 occurrences of variables in different contexts (Counting Environments - CE). An m-dimensional Characteristic Vector (CV) is generated using m CEs, where the ith dimension of the CV is the number of occurrences of the variable in the ith 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 modeled 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 (Turney and Pantel 2010; Tversky 1977) between their feature vectors. Evaluations on more than 100,000 Android apps collected from 5 Chinese app markets show that the approach can effectively detect cloned apps (Wang et al. 2015a). CrossSim is also able to deal with low-level features as by WuKong if such features are integrated into the graph presented in Section 3.

Lo et al. develop TagSim,[9] a tool that leverages tags to characterize applications and then to compute similarity between them (Lo et al. 2012). Tags are terms that are used to highlight the most important characteristics of software systems (Xia et al. 2013) and therefore, they help users narrow down the search scope. TagSim 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 with textual descriptions, TagSim 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 ones 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. Each application is characterized by a feature vector, and each entry corresponds to the weight of a tag the application has. Eventually, the similarity between two applications is computed as the cosine similarity (Turney and Pantel 2010; Tversky 1977) between the two vectors. To evaluate TagSim, more than a hundred thousands of projects have been collected and analyzed (Lo et al. 2012). A total of 20 queries were used to study the performance of the algorithm in comparison with CLAN. The experimental results show that TagSim helps achieve better performance in comparison with CLAN.

Being inspired by CLAN, Linares-Vásquez et al. develop CLANdroid for detecting similar Android applications with the assumption that similar apps share some semantic anchors (Linares-Vasquez et al. 2016). 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*[10] specification. This file is a mandatory component for any 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. Evaluations have been performed to study which semantic anchors are more effective (Linares-Vasquez et al. 2016). 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 gives some hints on the mechanism behind the way Google Play recommends similar apps. CROSSSIM is relevant to CLANdroid since it can work with low-level features by representing function calls, API calls in the graph as we already demonstrated in our recent work (Nguyen et al. 2019a, b).

With the aim of finding apps with similar semantic requirements, SimApp has been developed to exploit high-level metadata collected from apps markets (Chen et al. 2015). If two apps implement related semantic requirements then they are seen as similar. Each

[9]For the sake of clarity, in this paper, we give a name for the algorithms that have not been originally named
[10]https://developer.android.com/guide/topics/manifest/manifest-intro.html

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. The final similarity score for a pair of apps 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.

AnDarwin is an approach that applies Program Dependence Graphs to represent apps (Crussell et al. 2013), and feature vectors are then clustered to find similar apps. Locality Sensitive Hashing (Baltes et al. 2018) is used to find approximate near-neighbors from a large number of vectors. AnDarwin works according to 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). An evaluation using more than 200,000 apps from different Android markets demonstrated that the system can effectively detect cloned apps. LibRec is a tool that assists developers in leveraging existing libraries (Thung et al. 2013). It suggests the inclusion of libraries that may be useful for a given project using a combination of rule mining and collaborative filtering techniques. *Association rule* mining is applied to find similar libraries that co-exist in many projects to extract 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. A *collaborative filtering* technique is applied to search for top most similar projects and recommends libraries used by these projects. The libraries included by these projects are used as recommendations based on a score computed according to their popularity.

Considering a set of projects and a set of libraries, each project is characterized by a feature vector using the set of libraries it includes. The similarity between two projects is the cosine similarity between their feature vectors. In a previous work Nguyen et al. (2018b), we introduced CrossRec, a recommender system for providing software developers with third-party libraries. In this setting, CROSSSIM performs its computation using third-party libraries as the input features. By considering LibRec as baseline, we demonstrated that CrossRec obtains a superior performance with respect to various quality metrics.

A summary of all the similarity metrics introduced in Sections 2 and 6 is depicted in Table 9. 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, CLAN, and CLANdroid belong to this category.

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. A current trend in these techniques is to exploit textual content to compute similarity, e.g., in AppRec (Bhandari et al. 2013), SimApp (Chen et al. 2015), and TagSim (Lo et al. 2012).

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 (Garg et al. 2004). So it might be the case that two textual contents with different

Table 9 Summary of the similarity algorithms and their features

	Source code			Metadata						
	MUDABlue (Garg et al. 2004)	CLAN (McMillan et al. 2012)	CLANdroid (Linares-Vasquez et al. 2016)	WuKong (Wang et al. 2015a)	SimApp (Chen et al. 2015)	AnDarwin (Crussell et al. 2013)	TagSim (Lo et al. 2012)	LibRec (Thung et al. 2013)	RepoPal (Zhang et al. 2017)	
Considered features										Descriptions
Dependencies	–	–	–	✓	–	–	–	✓	–	Set of third-party libraries that a project includes
API calls	✓	✓	✓	✓	–	✓	–	–	–	API function calls that appear in the source code of the analyzed projects. They are used to build term-document matrices and then to calculate similarities among applications
Functions	✓	–	–	–	–	✓	–	–	–	Functions defined in a project's source code
Stars	–	–	–	–	–	–	–	–	✓	The GitHub star events occurred for each analyzed projects
Timestamps	–	–	–	–	–	–	–	–	✓	The point of time when a user stars a repository
Statements	✓	–	–	–	–	–	–	–	–	Source code statements
Identifiers	✓	–	✓	✓	–	–	–	–	–	All artifacts related to source code, such as variable names, function names, and package names
App name	–	–	–	–	✓	–	–	–	–	Name of a mobile app may reveal its functionalities

Table 9 (continued)

	Source code			Metadata						
	MUDABlue (Garg et al. 2004)	CLAN (McMillan et al. 2012)	CLANdroid (Linares-Vasquez et al. 2016)	WuKong (Wang et al. 2015a)	SimApp (Chen et al. 2015)	AnDarwin (Crussell et al. 2013)	TagSim (Lo et al. 2012)	LibRec (Thung et al. 2013)	RepoPal (Zhang et al. 2017)	
Descriptions	–	–	–	–	✓	–	–	–	–	The description text of an app
Developers	–	–	–	–	✓	–	–	–	–	All developers who contribute to the development of a software/an app
Readme	–	–	–	–	✓	–	✓	–	–	Descriptions or *README.MD* files, used to describe the functionalities of an open source project
Tags	–	–	–	–	–	–	✓	–	–	The tags that are used by OSS platforms, e.g. SourceForge to classify and characterize an OSS project
Updates	–	–	–	–	✓	–	–	–	–	The newest changes made to the considered applications
Permissions	–	–	✓	–	✓	–	–	–	–	This feature is available by mobile apps. It specifies the permission of an app to handle data in a smartphone
Screenshots	–	–	–	–	✓	–	–	–	–	This feature is available by mobile apps. It is a picture representing an app

Table 9 (continued)

	Source code			Metadata						
	MUDABlue (Garg et al. 2004)	CLAN (McMillan et al. 2012)	CLANdroid (Linares-Vasquez et al. 2016)	WuKong (Wang et al. 2015a)	SimApp (Chen et al. 2015)	AnDarwin (Crussell et al. 2013)	TagSim (Lo et al. 2012)	LibRec (Thung et al. 2013)	RepoPal (Zhang et al. 2017)	
Contents	–	–	–	–	✓	–	–	–	–	Each app has content rating to describe its content and age appropriateness
Size	–	–	–	–	✓	–	–	–	–	Some similarity metrics assume that two apps whose size is considerably different cannot be similar
Reviews	–	–	–	–	✓	–	–	–	–	All user reviews for an app are combined in a document
Intents	–	–	✓	–	–	–	–	–	–	For a mobile app, an intent is description for an operation to be performed
Sensors	–	–	✓	–	–	–	–	–	–	In mobile devices, sensors can provide raw data to monitor 3-D device movement or positioning, or changes in the environment. A set of features can be built from sensors to characterize an app

Table 9 (continued)

	Source code			Metadata						
	MUDABlue (Garg et al. 2004)	CLAN (McMillan et al. 2012)	CLANdroid (Linares-Vasquez et al. 2016)	WuKong (Wang et al. 2015a)	SimApp (Chen et al. 2015)	AnDarwin (Crussell et al. 2013)	TagSim (Lo et al. 2012)	LibRec (Thung et al. 2013)	RepoPal (Zhang et al. 2017)	
Used techniques										
TDM and LSA	✓	✓	✓	–	–	–	–	–	–	TDM (Collobert et al. 2011) and LSA (Landauer et al. 1998) 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 (Turney and Pantel 2010; Tversky 1977) is widely used in several algorithms for computing similarities among vectors
JCS	–	–	–	–	–	✓	–	–	✓	Jaccard index (Jaccard 1912) is used for computing similarity between two sets of elements

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 McMillan et al. (2012). To overcome this problem, the application of a synonym dictionary like WordNet (Miller 1995) is beneficial. Nevertheless, there is an issue with the approaches like RepoPal where readme files are used for similarity computation. Since in general the descriptions for software projects are written in different languages, the comparison of readme files in different languages should yield dissimilarity, even though two projects may be similar. SimApp (Chen et al. 2015) 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.

7 Conclusions

In this paper, we presented CrossSim, a framework for computing similarities among OSS projects. Through a review on some of the most notable methods for detecting similarity in software applications and open source projects, we came to the conclusion that a representation model that flexibly incorporates various features and semantic relationships is highly beneficial to similarity computation in the context of an OSS ecosystem. We considered the community of developers together with OSS projects, libraries, and various artifacts and their mutual interactions as whole by using the graph representation. In the graph, either humans or non-human factors have mutual dependency and implication on the others. By means of a graph, we are able to transform the relationships among various artifacts, e.g., developers, API utilizations, source code, interactions, into a mathematically computable format. The implementation of CrossSim exploits SimRank to compute similarity in graphs and it can handle the following relationships: *isUsedBy*, *develops*, and *stars*. CrossSim is a versatile similarity tool as it can accept various input features regardless of their format.

We conducted an evaluation of computing similarities among OSS projects on a dataset of 580 GitHub Java projects. Using MUDABlue, CLAN, RepoPal as baselines, we studied the performance of our approach. The obtained results are promising, among the test configurations, CrossSim3 has the best performance, where dependencies and star events are considered as features for the similarity computation. It is our belief that CrossSim is a good candidate for computing similarities among OSS projects. In order to enable the reproducibility of the performed experiments, we made available the source code implementation of MUDABlue, CLAN, RepoPal, and CrossSim as also the dataset exploited and the corresponding user evaluation in our GitHub repository (Nguyen et al. 2018d). For future work, we are going to investigate in more detail the influence of graph structure on similarity computation. In addition, we plan to incorporate also low-level similarity features such as API function calls, and package names into the graph to see if the recommendation performance can be improved. Last but not least, we are going to exploit CrossSim to automatically cluster OSS projects.

Acknowledgements The research described in this paper has been carried out as part of the CROSSMINER Project, EU Horizon 2020 Research and Innovation Programme, grant agreement No. 732223. We thank our project partners for the help with the user evaluation presented in this paper. Furthermore, we thank the anonymous reviewers for their valuable comments and suggestions that help us improve our paper.

Appendix

Questionnaire

This is the questionnaire sent to the developers who took part in our user evaluation. We adopted most of the content proposed by CLAN evaluation dataset (2018) and McMillan et al. (2012)

1. How many years of programming experience do you have?
2. Which programming languages are you capable of? And how many years have you worked with them?

Language	**Number of years**
Java	
Python	
Perl	
C++/C#	
Others	

3. How often do you use code search engines?
4. What code search engines have you used and for how long?
5. How often do you reuse source code snippets collected from the search engines in your work?
6. According to your experience, what is the main obstacle to using code search engines?

Materials

We uploaded the materials created from the user evaluation in GitHub for future reference.[11]

References

Bagnato, A., Barmpis, K., Bessis, N., Cabrera-Diego, L.A., Di Rocco, J., Di Ruscio, D., Gergely, T., Hansen, S., Kolovos, D., Krief, P., Korkontzelos, I., Laurière, S., Lopez de la Fuente, J.M., Maló, P., Paige, R.F., Spinellis, D., Thomas, C., Vinju, J. (2018). Developer-centric knowledge mining from large open-source software repositories (crossminer). In Seidl, M., & Zschaler, S. (Eds.) *Software technologies: applications and foundations* (pp. 375–384). Cham: Springer International Publishing.

Baltes, S., Dumani, L., Treude, C., Diehl, S. (2018). SOTorrent: reconstructing and analyzing the evolution of stack overflow posts. In: MSR.

Behnamghader, P., Alfayez, R., Srisopha, K., Boehm, B. (2017). Towards better understanding of software quality evolution through commit-impact analysis. In *2017 IEEE International conference on software quality, reliability and security (QRS)* (pp. 251–262).

Bhandari, U., Sugiyama, K., Datta, A., Jindal, R. (2013). Serendipitous recommendation for mobile apps using item-item similarity graph. In Banchs, R.E., Silvestri, F., Liu, T.-Y., Zhang, M., Gao, S., Lang, J. (Eds.) *AIRS, volume 8281 of lecture notes in computer science* (pp. 440–451): Springer.

Bizer, C., Heath, T., Berners-Lee, T. (2009). Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, *5*(3), 1–22.

[11] https://github.com/crossminer/CrossSim/blob/master/user-study/

Blondel, V.D., Gajardo, A., Heymans, M., Senellart, P., Dooren, P.V. (2004). A measure of similarity between graph vertices: applications to synonym extraction and web searching. *SIAM Review*, *46*(4), 647–666.

Borges, H., Hora, A., Valente, M.T. (2016). Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International conference on software maintenance and evolution (ICSME)* (pp. 334–344).

Chen, N., Hoi, S.C., Li, S., Xiao, X. (2015). 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* (pp. 305–314). New York: ACM.

CLAN evaluation dataset (2018). http://www.cs.wm.edu/semeru/clan/CaseStudyMaterials.zip. Last access 16.10.2018.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, *12*, 2493–2537.

Coutinho, A.E.V.B., Cartaxo, E.G., de Lima Machado, P.D. (2014). Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal*, *24*, 407–445.

Crussell, J., Gibler, C., Chen, H. (2013). 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* (pp. 182–199).

Di Noia, T., Mirizzi, R., Ostuni, V.C., Romito, D., Zanker, M. (2012). Linked open data to support content-based recommender systems. In *Proceedings of the 8th international conference on semantic systems, I-SEMANTICS '12* (pp. 1–8). New York: ACM.

Evans, W.S., Fraser, C.W., Ma, F. (2009). Clone detection via structural abstraction. *Software Quality Journal*, *17*(4), 309–330.

Garg, P.K., Kawaguchi, S., Matsushita, M., Inoue, K. (2004). MUDABlue: an automatic categorization system for open source repositories. In *2013 20th Asia-Pacific software engineering conference (APSEC)* (pp. 184–193).

Ghose, S., & Lowengart, O. (2001). Taste tests: impacts of consumer perceptions and preferences on brand positioning strategies. *Journal of Targeting, Measurement and Analysis for Marketing*, *10*(1), 26–41.

Gitchell, D., & Tran, N. (1999). Sim: a utility for detecting similarity in computer programs. In *The proceedings of the thirtieth SIGCSE technical symposium on computer science education, SIGCSE '99* (pp. 266–270). New York: ACM.

Jaccard, P. (1912). The distribution of the flora in the alpine zone. *New Phytologist*, *11*(2), 37–50.

Jeh, G., & Widom, J. (2002). Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining, KDD '02* (pp. 538–543). New York: ACM.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P.S., Zhang, L. (2017). Why and how developers fork what from whom in github. *Empirical Software Engineering*, *22*(1), 547–578.

Kendall, M.G. (1938). A new measure of rank correlation. *Biometrika*, *30*(1/2), 81–93.

Khan, S.U.R., Lee, S.P., Ahmad, R.W., Akhunzada, A., Chang, V. (2016). A survey on test suite reduction frameworks and tools. *International Journal of Information Management*, *36*(6), 963–975.

Kobilarov, G., Scott, T., Raimond, Y., Oliver, S., Sizemore, C., Smethurst, M., Bizer, C., Lee, R. (2009). Media meets semantic web – how the bbc uses dbpedia and linked data to make connections. In Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (Eds.) *The semantic web: research and applications* (pp. 723–737). Berlin: Springer.

Kollias, G., Sathe, M., Schenk, O., Grama, A. (2014). Fast parallel algorithms for graph similarity and matching. *Journal of Parallel and Distributed Computing*, *74*(5), 2400–2410.

Landauer, T.K. (2006). *Latent semantic analysis*. Wiley Online Library.

Landauer, T., Foltz, P., Laham, D. (1998). An introduction to latent semantic analysis. *Discourse Processes*, *25*, 259–284.

Leitão, A.M. (2004). Detection of redundant code using r2d2. *Software Quality Journal*, *12*(4), 361–382.

Linares-Vasquez, M., Holtzhauer, A., Poshyvanyk, D. (2016). On automatically detecting similar android apps. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, *00*, 1–10.

Liu, C., Chen, C., Han, J., Yu, P.S. (2006). 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* (pp. 872–881). New York: ACM.

Lo, D., Jiang, L., Thung, F. (2012). Detecting similar applications with collaborative tagging. In *Proceedings of the 2012 IEEE international conference on software maintenance (ICSM), ICSM '12* (pp. 600–603). Washington, DC: IEEE Computer Society.

Maarek, Y.S., Berry, D.M., Kaiser, G.E. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, *17*(8), 800–813.

McMillan, C., Grechanik, M., Poshyvanyk, D. (2012). Detecting similar software applications. In *Proceedings of the 34th international conference on software engineering, ICSE '12* (pp. 364–374). Piscataway: IEEE Press.

Miller, G.A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, *38*(11), 39–41.

Nassar, H., Veldt, N., Mohammadi, S., Grama, A., Gleich, D.F. (2018). Low rank spectral network alignment. In *Proceedings of the 2018 World Wide Web conference, WWW '18* (pp. 619–628). Republic and Canton of Geneva: International World Wide Web Conferences Steering Committee.

Nguyen, P.T., Tomeo, P., Di Noia, T., Di Sciascio, E. (2015). 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* (pp. 1477–1482). New York: ACM.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D. (2018a). Knowledge-aware recommender system for software development. In *Proceedings of the 1st Workshop on Knowledge-aware and Conversational Recommender System, KaRS*, Vol. 2018. New York: ACM.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D. (2018b). Mining software repositories to support OSS developers: a recommender systems approach. In *Proceedings of the 9th Italian information retrieval workshop, Rome, Italy, May, 28-30, 2018.*

Nguyen, P.T., Di Rocco, J., Rubei, R., Di Ruscio, D. (2018c). CrossSim: exploiting mutual relationships to detect similar OSS projects. In *2018 44th Euromicro conference on software engineering and advanced applications (SEAA)* (pp. 388–395).

Nguyen, P.T., Di Rocco, J., Rubei, R., Di Ruscio, D. (2018d). CrossSim tool and evaluation data. https://github.com/crossminer/CrossSim.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D. (2019a). Enabling heterogeneous recommendations in OSS development: what's done and what's next in CROSSMINER. In *Proceedings of the evaluation and assessment on software engineering, EASE '19* (pp. 326–331). New York: ACM.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., Di Penta, M. (2019b). FOCUS: a recommender system for mining API function calls and usage patterns. In *Proceedings of the 41st international conference on software engineering, ICSE '19* (pp. 1050–1060). Piscataway: IEEE Press.

Pettigrew, S., & Charters, S. (2008). Tasting as a projective technique. *Qualitative Market Research: An International Journal*, *11*(3), 331–343.

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M. (2014). Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of MSR 2014* (pp. 102–111): ACM.

Ragkhitwetsagul, C., Krinke, J., Clark, D. (2018a). A comparison of code similarity analysers. *Empirical Software Engineering*, *23*(4), 2464–2519.

Ragkhitwetsagul, C., Krinke, J., Marnette, B. (2018b). A picture is worth a thousand words: code clone detection based on image similarity. In *2018 IEEE 12th International workshop on software clones (IWSC)* (pp. 44–50).

Rattan, D., Bhatia, R., Singh, M. (2013). Software clone detection: a systematic review. *Information and Software Technology*, *55*(7), 1165–1199.

Schafer, J.B., Frankowski, D., Herlocker, J., Sen, S. (2007). *The adaptive web. Chapter collaborative filtering recommender systems*, (pp. 291–324). Berlin: Springer.

Spearman, C. (1904). The proof and measurement of association between two things. *The American Journal of Psychology*, *15*(1), 72–101.

Spinellis, D., & Szyperski, C. (2004). How is open source affecting software development? *IEEE Software*, *21*(1), 28–33.

Stadler, C., Lehmann, J., Höffner, K., Auer, S. (2012). LinkedGeoData: a core for a web of spatial open data. *Semantic Web*, *3*, 333–354.

Thung, F., Lo, D., Lawall, J. (2013). Automated library recommendation. In *2013 20th Working conference on reverse engineering (WCRE)* (pp. 182–191).

Tiarks, R., Koschke, R., Falke, R. (2011). An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal*, *19*(2), 295–331.

Turney, P.D., & Pantel, P. (2010). From frequency to meaning: vector space models of semantics. *Journal of Artificial Intelligence Research*, *37*(1), 141–188.

Tversky, A. (1977). Features of similarity. *Psychological Review*, *84*(4), 327–352.

Ugurel, S., Krovetz, R., Giles, C.L. (2002). 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* (pp. 632–638). New York: ACM.

Walenstein, A., El-Ramly, M., Cordy, J.R., Evans, W.S., Mahdavi, K., Pizka, M., Ramalingam, G., von Gudenberg, J.W. (2006). Similarity in programs. In *Duplication, redundancy, and similarity in software, 23.07. - 26.07.2006.*

Wang, H., Guo, Y., Ma, Z., Chen, X. (2015a). 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* (pp. 71–82). New York: ACM.

Wang, M., Wang, C., Yu, J.X., Zhang, J. (2015b). Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *Proceedings of the VLDB Endowment*, *8*(10), 998–1009.

Xia, X., Lo, D., Wang, X., Zhou, B. (2013). Tag recommendation in software information sites. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13* (pp. 287–296). Piscataway: IEEE Press.

Zhang, Y., Lo, D., Kochhar, P.S., Xia, X., Li, Q., Sun, J. (2017). Detecting similar repositories on GitHub. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, *00*, 13–23.

Phuong T. Nguyen is a postdoctoral researcher at the University of L'Aquila, Italy. He obtained a PhD in Computer Science from the University of Jena, Germany. Since the graduation, he has worked as a university teaching and research assistant in Vietnam and Italy. His research interests include Computer Networks, Semantic Web, and Recommender Systems. Recently, he has been working to develop recommender systems in Software Engineering for mining open source code repositories.

Juri Di Rocco is a postdoctoral researcher at the University of L'Aquila, Italy. He obtained a PhD in Computer Science from the University of L'Aquila. He is interested in several aspects of software language engineering and Model Driven Engineering (MDE) including domain specific modeling languages, model transformation, model differencing, modeling repositories, and mining techniques. More information is available at http://www.di.univaq.it/juri.dirocco

Riccardo Rubei is PhD student at the Unversity of L'Aquila, Italy. He is working on mining techniques to analyze open source software with the aim of providing developers with useful real-time recommendations.

Davide Di Ruscio is an Associate Professor at the University of L'Aquila. His main research interests include software engineering, and several aspects of Model Driven Engineering including domain-specific languages, model transformations, and model evolution. He has published more than 130 papers in various journals, conferences, and workshops on such topics. Over the last decade, he has worked on several European projects by contributing the application of MDE in different application domains like service-based software systems, autonomous systems, and open source software (OSS). More information is available at http://www.di.univaq.it/diruscio.