CrossSim: exploiting mutual relationships to detect similar OSS projects

Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Davide Di Ruscio
Department of Information Engineering, Computer Science and Mathematics
Università degli Studi dell'Aquila
Via Vetoio 2, 67100 – L'Aquila, Italy
{phuong.nguyen, juri.dirocco, riccardo.rubei, davide.diruscio}@univaq.it

***Abstract*—Software development is a knowledge-intensive activity, which requires mastering several languages, frameworks, technology trends (among other aspects) under the pressure of ever-increasing arrays of external libraries and resources. Recommender systems are gaining high relevance in software engineering since they aim at providing developers with real-time recommendations, which can reduce the time spent on discovering and understanding reusable artifacts from software repositories, and thus inducing productivity and quality gains.**

In this paper, we focus on the problem of mining open source software repositories to identify similar projects, which can be evaluated and eventually reused by developers. To this end, CROSSSIM is proposed as a novel approach to model open source software projects and related artifacts and to compute similarities among them. An evaluation on a dataset containing 580 GitHub projects shows that CROSSSIM outperforms an existing technique, which has been proven to have a good performance in detecting similar GitHub repositories.

***Index Terms*—mining software repositories, software similarities, SimRank**

I. Introduction

Software development is a challenging and knowledge-intensive activity. It requires mastering several programming languages, frameworks, design patterns, technology trends (among other aspects) under the pressure of ever-increasing arrays of external resources [19]. Consequently, software developers are continuously spending time and effort to understand existing code, new third-party libraries, or how to properly implement a new feature. The time spent on discovering useful information can have a dramatic impact on productivity [6].

Over the last few years, a lot of effort has been spent on data mining and knowledge inference techniques to develop methods and tools able to provide automated assistance to developers in navigating large information spaces and giving recommendations that might be helpful to solve the particular development problem at hand. The main intuition is to bring to the domain of software development the notion of recommendation systems that are typically used for popular e-commerce systems to present users with interesting items previously unknown to them [18]. By setting the focus on contexts characterized by the availability of large repositories of reusable open source software (OSS) like GitHub[1], Bitbucket[2], and SourceForge[3] (just to mention a few), it is of paramount importance to conceive techniques and tools able to help software engineers identify reusable and similar open source projects, which can be re-used instead of implementing in-house proprietary solutions with similar functionalities.

Two applications 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 [14]. Understanding the similarities between open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations [21], [25]. Meanwhile measuring the similarities between developers and software projects is a critical phase for most types of recommender systems [17], [20]. Failing to compute precise similarities means concurrently adding a decline in the overall performance of these systems. Measuring similarities between software systems has been identified as a daunting task in previous work [3], [14]. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail. Currently available techniques for calculating OSS project similarities can be categorized in two different groups depending on the abstract layers they work on, i.e., low-level and high-level. The former considers source code, function calls, API references, etc., whereas the latter considers project metadata, e.g. textual description, readme files to calculate software system similarities.

In this paper we propose CROSSSIM, an approach that permits to represent in a homogeneous manner different project characteristics belonging to different abstraction layers. In particular, a graph-based model has been devised to enable both the representation of different open source software projects and the calculation of their similarity. Thus, the main contributions of this paper are the following: (*i*) proposing a novel approach to represent the open source software ecosystem exploiting its mutual relationships; (*ii*) developing an extensible and flexible framework for computing similarities

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.

[1]GitHub: https://github.com/
[2]Bitbucket: https://bitbucket.org/
[3]SourceForge: https://sourceforge.net/

978-1-5386-7383-6/18/$31.00 ©2018 IEEE
DOI 10.1109/SEAA.2018.00069

among open source software projects; and (*iii*) evaluating the performance of the proposed framework with regards to a well-established baseline.

The rest of the paper is organized as follows: Section II presents an overview of the most notable approaches for detecting similar software applications and open source projects. Section III brings in our proposed approach for computing similarities between OSS projects. An initial evaluation on a real GitHub dataset is described in Section IV. Section V presents the experimental results. Finally, Section VI concludes the paper and draws some perspective work.

II. Background

In this section, we introduce the problem of detecting similar software projects by referring to existing techniques and tools that have been developed over the last few years. According to [3], depending on the set of input features, there are two main types of software similarity computation, i.e. low-level and high-level as discussed below.

Low-level similarity. It is calculated by considering low-level data, e.g., source code, byte code, function calls, API reference, etc. The authors in [7] propose MUDABlue, an approach for computing similarity between software projects using source code. To compute similarities between software systems, MUDABlue first extracts identifiers from source code and removes unrelated content. It then creates an identifier-software matrix where each row corresponds to one identifier and each column corresponds to a software system. Afterwards, it removes too rare or too popular identifiers. Finally, latent semantic analysis (LSA) [10] is performed on the identifier-software matrix to compute similarity on the reduced matrix using cosine similarity. CLAN (Closely reLated ApplicatioNs) [14] is an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to packages class hierarchies. CLAN represents source code files as a term document matrix (TDM), in which a row contains a unique class or package and a column corresponds to an application. Singular value decomposition is then applied to reduce the matrix dimensionality. Similarity between applications is computed as the cosine similarity between vectors in the reduced matrix.

MUDABlue and CLAN are comparably similar in the way they represent software and identifiers/API in a term-document matrix and then apply LSA to compute similarities. CLAN includes API calls for computing similarity, whereas MUDABlue integrates every word in source code files into the term-document matrix. As a result, the similarity scores of CLAN reflect better the perception of humans of similarity than those of MUDABlue [14].

High-level similarity. It is calculated by considering project metadata, such as topic distribution, README files, textual descriptions, star events (if available e.g., in GitHub), etc. In [23] authors propose LibRec, a library recommendation technique to help developers leverage existing libraries. LibRec employs association rule mining and collaborative filtering techniques to search for top most similar projects and recommends libraries used by these projects to a given project. A project is characterized by a feature vector where each entry corresponds to the occurrence of a library and the similarity between two projects is computed as the similarity between their feature vectors.

In [13] tags are leveraged to characterize applications and then to compute similarity between them. The proposed approach can be used to detect similar applications written in different languages. Based on the hypothesis that tags capture better the intrinsic features of applications compared to textual descriptions, the approach extracts tags attached to an application and computes their weights [13]. This information forms the features of a given software system and is used to distinguish it from others. An application is characterized by a feature vector with each entry corresponding to the weight of a tag. Eventually, the similarity between two applications is computed using cosine similarity.

In [25], RepoPal is proposed to detect similar GitHub repositories. In this approach, two 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: the `README.md` file and the stars of each repository, and the time gap between two subsequent star events from the same user. RepoPal has been evaluated against CLAN and the experimental results [25] show that RepoPal has a better performance compared to that of CLAN with regards to two quality metrics.

In summary, by reviewing other additional similarity metrics [4], [11], [12], [24] which cannot be presented here due to space limitation, we have seen that they normally deal with either low-level or high-level similarity. We are convinced that combining various input information in computing similarities is highly beneficial to the context of OSS repositories. We aim to design a representation model that integrates semantic relationships among various artifacts and the model is expected to improve the overall performance of the similarity computation.

III. A novel approach for computing similarities among OSS projects

In Linked Data [1], an RDF[4] graph is made up of an enormous number of nodes and oriented links with semantic relationships. Thanks to this feature, the representation paves the way for various computations [5]. By considering the analogy of typical applications of RDF graphs and the problem of detecting the similarity of open source projects, in this section we propose CrossSim (**C**ross Project **R**elationships for Computing **O**pen **S**ource **S**oftware **Sim**ilarity), an approach that makes use of graphs for representing different kinds of relationships in the OSS ecosystem. Specifically, the graph model has been chosen since it allows for flexible data integration and facilitates numerous similarity metrics

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

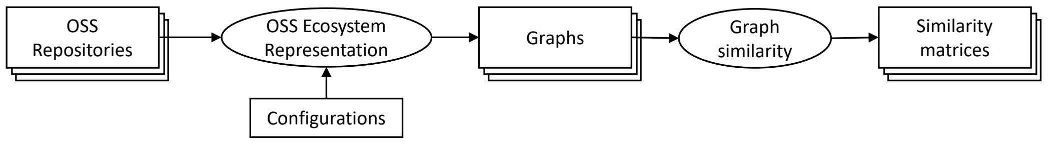


Fig. 1. Overview of the CROSSSIM approach

[2]. We consider the community of developers together with OSS projects, libraries and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependency and implication on the others. There, several connections and interactions prevail, such as developers commit to repositories, users star repositories, or projects contain source code files, just to name a few.

The architecture of CROSSSIM is depicted in Fig. 1: the rectangles represent artifacts, whereas the ovals represent activities that are automatically performed by the developed CROSSSIM tooling. In particular, the approach imports project data from existing OSS repositories and represents them into a graph-based representation by means of the `OSS Ecosystem Representation` module. Depending on the considered repository (and thus to the information that is available for each project) the graph structure to be generated has to be properly configured. For instance in case of GitHub, specific configurations have to be specified in order to enable the representation in the target graphs of the stars assigned to each project. Such a configuration is "forge" specific and specified once, e.g., SourceForge does not provide the star based system available in GitHub. The `Graph similarity` module implements the similarity algorithm that is applied on the source graph-based representation of the input ecosystems generates matrices representing the similarity value for each pair of input projects.

A detailed description of the proposed graph-based representation of open source projects is given in Sec. III-A. Details about the implemented similarity algorithm are given in Sec. III-B.

A. Representation of the OSS Ecosystem

With the adoption of the graph-based representation, we are able to 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 by means of the algorithm presented in the next section.

- *isUsedBy* $\subseteq$ *Dependency*$\times$*Project*: this relationship depicts the reliance of a project on a dependency (e.g., a third-party library). The project needs to include the dependency in order to function. According to [14], [23] the similarity between two considered projects relies on the dependencies they have in common because they aim at implementing similar functionalities;
- *develops* $\subseteq$ *Developer* $\times$ *Project*: we suppose that there is a certain level of similarity between two projects if they are built by same developers [3];
- *stars* $\subseteq$ *User* $\times$ *Project*: this relationship is inspired by the star event in RepoPal [25] to represent GitHub projects that a given user has starred. However, we consider the star event in a broader scope in the sense that not only direct but also indirect connections between two developers is taken into account;
- *develops* $\subseteq$ *User* $\times$ *Project*: this relationship is used to represent the projects that a given user contributes in terms of source code development;
- *implements* $\subseteq$ *File* $\times$ *File*: it represents a specific relation that can occur between the source code given in two different files, e.g. a class specified in one file implementing an interface given in another file;
- *hasSourceCode* $\subseteq$ *Project* $\times$ *File*: it represents the source files contained in a given project.

Fig. 2 shows a graph representing an explanatory example consisting of two projects `project#1` and `project#2`. The former contains `HttpSocket.java` and the latter contains `FtpSocket.java` with the corresponding semantic predicate `hasSourceCode`. Both source code files implement `interface#1` marked by `implements`. In practice, an OSS graph is much larger with numerous nodes and edges, and the relationship between two projects can be thought as a sub-graph.

Based on the graph structure, one can exploit nodes, links, and the mutual relationships to compute similarity using existing graph similarity algorithms. To the best of our knowledge, there exist several metrics for computing similarity in graph [2], [16]. In Fig. 2, we can compute the similarity between `project#1` and `project#2` using related semantic

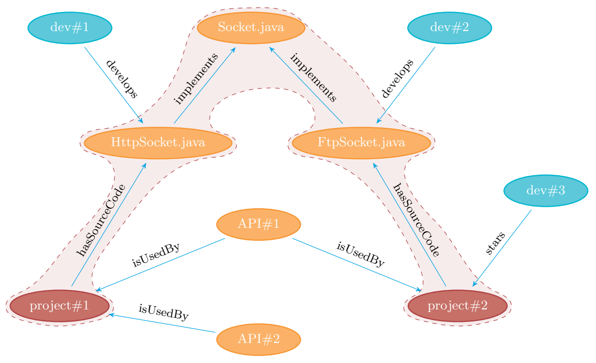


Fig. 2. Similarity between OSS projects w.r.t their implementation

paths, e.g. the one-hop path `isUsedBy`, or the two-hop path `hasSourceCode` and `implements` as already highlighted in the figure. The hypothesis is based on the fact that the projects are aiming at creating common functionalities by using common libraries [14], [23]. Using the graph, it is also possible to compute the similarity between developers `dev#1` and `dev#2` since they are indirectly connected by the `develops` and `implements` relationships.

The currently available implementation of CROSSSIM [15] is able to manage the *isUsedBy*, *develops*, and *stars* relationships as discussed in Sec. IV.

B. Similarity Computation

To evaluate the similarity of two nodes in a graph, their intrinsic characteristics like neighbour nodes, links, and their mutual interactions are incorporated into the similarity calculation [5], [16]. In [8], SimRank has been developed to calculate similarities based on mutual relationships between graph nodes. Considering two nodes, the more similar nodes point to them, the more similar the two nodes are. In this sense, the similarity between two nodes α, β is computed by using a fixed-point function. Given $k \geq 0$ we have $R^{(k)}(\alpha,\beta) = 1$ with $\alpha = \beta$ and $R^{(k)}(\alpha,\beta) = 0$ with $k = 0$ and $\alpha \neq \beta$, SimRank is computed as follows:

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

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

For the first implementation of CROSSSIM we adopt SimRank as the mechanism for computing similarities among OSS graph nodes. For future work, other similarity algorithms can also be flexibly integrated into CROSSSIM, as long as they are designed for graph.

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 [13], [14], [25]. 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.

IV. EVALUATION

In this section we discuss the process that has been conceived and applied to evaluate the performance of CROSSSIM in comparison with RepoPal. The rationale behind the selection of RepoPal is that according to *Zhang et al.* [25], RepoPal outperforms CLAN in terms of *Confidence* and *Precision*, and CLAN is deemed to be a well-established baseline [14]. Intuitively, we consider RepoPal as a good starting point for a performance comparison. Research questions that we wanted to answer by means of the performed evaluation are the following:

- **RQ1:** Which similarity metric yields a better performance: RepoPal or CROSSSIM?
- **RQ2:** How does the graph structure affect the performance of CROSSSIM?

To this end, the evaluation process that has been applied is shown in Fig. 3 and consists of activities and artifacts that are detailed below.

Data Collection We collected a dataset consisting of GitHub Java projects that serve as inputs for the similarity computation and satisfy the following requirements: (*i*) being GitHub Java projects; (*ii*) providing the specification of their dependencies by means of `pom.xml` or `.gradle` files[5]; (*iii*) having at least 9 dependencies; (*iv*) having the `README.md` file available; (*v*) possessing at least 20 stars [25]. We realized that the final outcomes of a similarity algorithm are to be validated by human beings, and in case the projects are irrelevant by their very nature, the perception given by human evaluators would also be *dissimilar* in the end. This is valueless for the evaluation of similarity. Thus, to facilitate the analysis, instead of crawling projects in a random manner, we first observed projects in some specific categories (e.g., PDF processors, JSON parsers, Object Relational Mapping projects, and Spring MVC related tools). Once a certain number of projects for each category had been obtained, we also started collecting randomly to get projects from various categories.

Using the GitHub API[6], we crawled projects to provide input for the evaluation. Though the number of projects that fulfill the requirements of a single approach, i.e. either RepoPal or CROSSSIM, is high, the number of projects that meet the requirements of both approaches is considerably lower. For example, a project contains both `pom.xml` and `README.md`, albeit having only 5 dependencies, thus it does not meet the constraints and must be discarded. The crawling is time consuming as for each project, at least 6 queries must be sent to get the relevant data. GitHub already sets a rate limit for an ordinary account[7], with a total number of 5.000 API calls per hour being allowed. And for the search operation, the rate is limited to 30 queries per minute. Due to these reasons, we ended up getting a dataset of 580 projects that are eligible for the evaluation. The dataset we collected and the CROSSSIM tool are already published online for public usage [15].

***Application of RepoPal and* CROSSSIM** Both RepoPal and CROSSSIM have been applied on the collected dataset. For explanatory purposes about the graph based representation, Fig. 4 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 I. The turquoise nodes

[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.

[6]GitHub API: https://developer.github.com/v3/

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

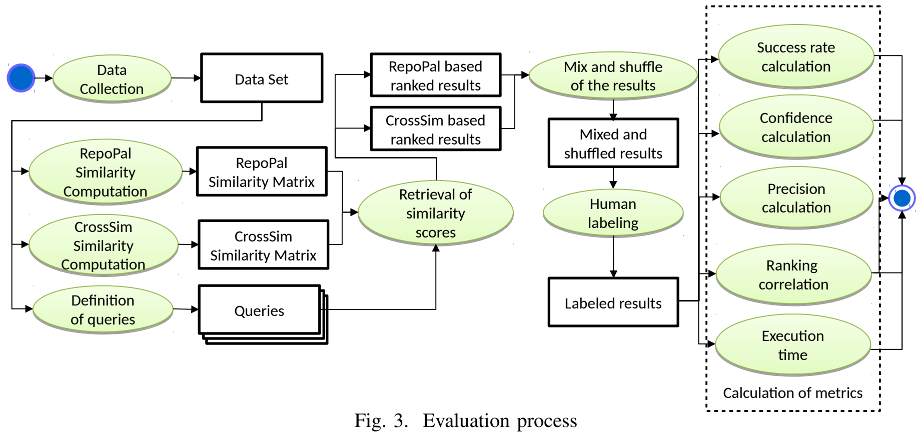

Data Collection
Data Set
RepoPal based ranked results
CrossSim based ranked results
Mix and shuffle of the results
Mixed and shuffled results
Human labeling
Labeled results
RepoPal Similarity Computation
RepoPal Similarity Matrix
CrossSim Similarity Computation
CrossSim Similarity Matrix
Retrieval of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

Fig. 3. Evaluation process

are developers who already starred the repositories. Every node is encoded using a unique number across the whole graph.

In order to address RQ2, we investigated the implication of graph structure on the performance of CROSSSIM by considering various types of graphs. By the first configuration, only *star events* and *dependencies* were used to build the graph and hereafter this is named as CROSSSIM$_1$. In the second configuration we extended CROSSSIM$_1$ by representing also committers and such a configuration is named as CROSSSIM$_2$. Next, we studied the influence of the most frequent dependencies (shown in Table II) on the computation. To this end, from the graph in the configuration CROSSSIM$_1$, all the nodes and edges derived from these dependencies are removed, and this configuration is denoted as CROSSSIM$_3$. Finally, the most frequent dependencies are also removed from CROSSSIM$_2$, resulting in CROSSSIM$_4$.

Query definition Among 580 projects in the dataset, 50 have been selected as queries. Due to space limitation, the list of the 50 queries is omitted from the paper, interested readers are referred to the dataset we published online [15] for more detail. To aim for variety, the queries have been chosen to equally cover all the categories of the projects in the dataset.

Retrieval of similarity scores Our evaluation has been conducted in line with some other existing studies [13], [14], [25]. 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 Sec. III-B. From the retrieved projects, only top 5 are selected for the subsequent evaluation steps. For each query, similarity is also computed using RepoPal to get the top-5 most similar retrieved projects.

Mix and shuffle of the results *and* ***Human Labeling*** In order to have a fair evaluation, for each query we mix and shuffle the top-5 results generated from the computation by all similarity metrics in a single file and present them to human evaluators. This helps eliminate any bias or prejudice against a specific similarity metric. In particular, given a query, a user study is performed to evaluate the similarity between the query and the corresponding retrieved projects. Three postgraduate students participated in the user study with two of them being skilful Java programmers. The participants are asked to label the similarity for each pair of projects (i.e., *<query,*

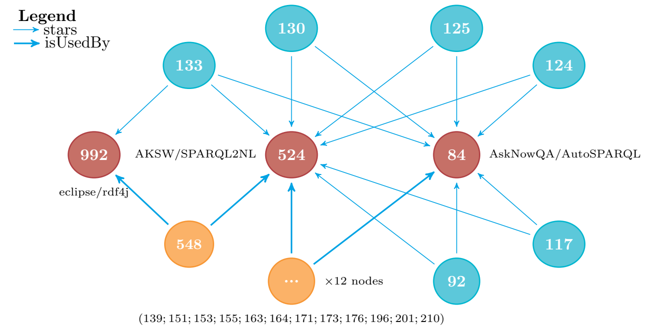


Fig. 4. Sub-graph showing a fragment of the AskNowQA/AutoSPARQL, AKSW/SPARQL2NL, and eclipse/rdf4j project representation

TABLE I
SHARED DEPENDENCIES IN THE CONSIDERED DATASET

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

TABLE II
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 III
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

retrieved project>) with regards to their application domains and functionalities using the scales listed in Table III [14].

Calculation of metrics To evaluate the outcomes of the algorithms with respect to the user study, the following metrics have been considered as typically done in some related work [13], [14], [25]:

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

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

- *Ranking*: The obtained human evaluation has been 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 [22] is used to measure how well a similarity metric ranks the retrieved projects given a query. Considering two ranked variables $r_1 = (\rho_1, \rho_2, .., \rho_n)$ and $r_2 = (\sigma_1, \sigma_2, .., \sigma_n)$, r_s is defined as: $r_s = 1 - \frac{6\sum_{i=1}^{n}(\rho_i - \sigma_i)^2}{n(n^2-1)}$. Because of the large number of ties, we also used *Kendall's tau* [9] 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 RepoPal and CROSSSIM on the dataset to obtain the corresponding similarity matrices.

V. EXPERIMENTAL RESULTS

In this section the data that has been obtained as discussed in the previous section is analyzed to answer the research questions RQ1 and RQ2 (see Sec. V-A). Threats to validity of our evaluation are also discussed in Sec. V-B.

A. Data analysis

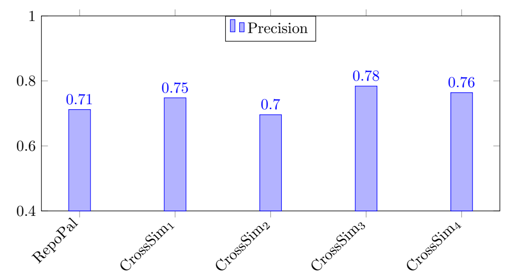
RQ1: Which similarity metric yields a better performance: RepoPal or CROSSSIM? The experimental results suggest that RepoPal is a good choice for computing similarity among OSS projects. This indeed confirms the claim made by the authors of RepoPal in [25]. In comparison with RepoPal, three CROSSSIM configurations gain a superior performance, with CROSSSIM$_3$ overtaking all.

As can be seen in and Fig. 5(a), CROSSSIM$_3$ outperforms RepoPal with respect to *Precision*. Both gain a *success rate* of 100%, however CROSSSIM$_3$ has a better precision. CROSSSIM$_3$ obtains a precision of 0.78 and RepoPal gets 0.71. The *Confidence* for both metrics is shown in Fig. 5(b). Also by this index, CROSSSIM$_3$ yields a better outcome as it has more scores that are either 3 or 4 and less scores that are 1 or 2.

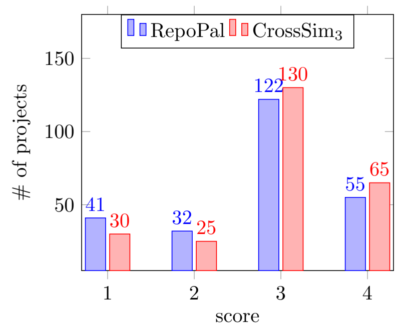
In addition to the conventional quality indexes, we investigated the ranking produced by the two metrics using the Spearman's (r_s) and Kendall's tau (τ) correlation indexes. The aim is to see how good is the correlation between the rank generated by each metric and the scores given by the users, which are already sorted in descending order. In this way, a lower r_s (τ) means a better ranking. r_s and τ are computed for all 50 queries and related first five results. The value of r_s is 0.250 for CROSSSIM$_3$ and -0.193 for $RepoPal$. The value of τ is -0.214 for CROSSSIM$_3$ and -0.163 for $RepoPal$. By this quality index, CROSSSIM$_3$ performs slightly better than $RepoPal$.

The execution time related to the application of RepoPal and CROSSSIM$_3$ is shown in Fig. 5(c). For the experiments on the dataset using a laptop with Intel Core i5-7200U CPU @ 2.50GHz × 4, 8GB RAM, Ubuntu 16.04, RepoPal takes ≈4 hours to generate the similarity matrix, whereas the execution of CROSSSIM$_3$, including both the time for generating the input graph and that for generating the similarity matrix, takes ≈16 minutes. Such an important time difference is due to the time needed to calculate the similarity between `README.md` files, on which *RepoPal* relies.

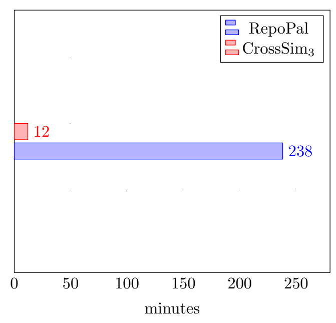
The results obtained by CROSSSIM confirm our hypothesis that the incorporation of various features, e.g. dependen-



(a) Precision



(b) Confidence



(c) Execution time

Fig. 5. Outcomes of the considered metrics

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

RQ2: How does the graph structure affect the performance of* CROSSSIM*? When we consider CROSSSIM$_1$ in combination with CROSSSIM$_2$, the effect of the adoption of committers can be observed. CROSSSIM$_1$ gains a success rate of 100%, with a precision of 0.748. Whereas, the number of false positives by CROSSSIM$_2$ goes up, thereby worsening the overall performance considerably with 0.696 being as the precision. The precision of CROSSSIM$_2$ is lower than those of *RepoPal* and all of its CROSSSIM counterparts. The performance degradation is further witnessed by considering CROSSSIM$_3$ and CROSSSIM$_4$ together. With respect to CROSSSIM$_3$, the number of false positives by CROSSSIM$_4$ increases by 5 projects. We come to the conclusion that the inclusion of all developers who have committed updates at least once to a project in the graph is counterproductive as it adds a decline in precision. In this sense, we make an assumption that the deployment of a weighting scheme for developers may help counteract the degradation in performance. We consider the issue as our future work.

We consider CROSSSIM$_1$ and CROSSSIM$_3$ together to analyze the effect of the removal of the most frequent dependencies. CROSSSIM$_3$ outperforms CROSSSIM$_1$ as it gains a precision of 0.78, the highest value among all, compared to 0.75 by CROSSSIM$_1$. The removal of the most frequent dependencies helps also improve the performance of CROSSSIM$_4$ in comparison to CROSSSIM$_2$. 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 III-B. 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 II this characteristic may not hold anymore. For example, two projects are pointed by junit:junit because they use JUnit[8] 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 graph.

In summary, it can be seen that the graph structure considerably affects the outcome of the similarity computation. In this sense, finding a graph structure that nourishes similarity computation is of particular importance. This is considered as an open research problem.

B. 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 internal and external threats to validity as discussed below.

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

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

External validity refers to the generalizability of obtained results and findings. Concerning the generalizability of our approach, we were able to consider only a dataset of 580 projects, due to the fact that the number of projects that meet the requirements of both RepoPal and CROSSSIM is low and thus required a prolonged crawling. During the data collection, we crawled both projects in some specific categories as well as random projects. The random projects served as a means to test the generalizability of our algorithm. If the algorithm works well, it will not perceive newly added random projects as similar to projects of the specific categories. For future work, we are going to validate our proposed approach by incorporating other similarity metrics and more GitHub projects.

VI. CONCLUSIONS

In this paper, we presented an approach to detect similar open source software projects. We proposed a graph-based representation of various features and semantic relationships of open source projects. By means of the proposed graph representation, we were able to transform the relationships among various artifacts, e.g. developers, API utilizations, source code, interactions, into a mathematically computable format.

An evaluation was conducted to study the performance of our approach on a dataset of 580 GitHub Java projects. The obtained results are promising: by considering RepoPal as baseline, we demonstrated that CROSSSIM can be considered as a good candidate for computing similarities among open source software projects. For future work, we are going to investigate which graph structure can help obtain a better similarity outcome as well as to define a threshold so that a project dependency is considered to be *frequent*.

REFERENCES

[1] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):122, 2009.

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

[3] N. Chen, S. C. Hoi, S. Li, and X. Xiao. 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, pages 305–314, New York, NY, USA, 2015. ACM.

[4] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar android applications. In J. Crampton, S. Jajodia, and K. Mayes, editors, *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 182–199, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] T. Di Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker. Linked open data to support content-based recommender systems. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 1–8, New York, NY, USA, 2012. ACM.

[6] E. Duala-Ekoko and M. P. Robillard. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 266–276, Piscataway, NJ, USA, 2012. IEEE Press.

[7] P. K. Garg, S. Kawaguchi, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 184–193, 2004.

[8] G. Jeh and J. Widom. Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 538–543, New York, NY, USA, 2002. ACM.

[9] M. G. Kendall. Rank correlation methods. 1948.

[10] T. K. Landauer. *Latent semantic analysis*. Wiley Online Library, 2006.

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

[12] C. Liu, C. Chen, J. Han, and P. S. Yu. 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, pages 872–881, New York, NY, USA, 2006. ACM.

[13] D. Lo, L. Jiang, and F. Thung. Detecting similar applications with collaborative tagging. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 600–603, Washington, DC, USA, 2012. IEEE Computer Society.

[14] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, Piscataway, NJ, USA, 2012. IEEE Press.

[15] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio. CrossSim tool and evaluation data, 2018. https://doi.org/10.5281/zenodo.1252866.

[16] P. T. Nguyen, P. Tomeo, T. Di Noia, and E. Di Sciascio. 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, pages 1477–1482, New York, NY, USA, 2015. ACM.

[17] T. D. Noia and V. C. Ostuni. Recommender systems and linked open data. In *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*, pages 88–113, 2015.

[18] F. Ricci, L. Rokach, and B. Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[19] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. DOI: 10.1007/978-3-642-45135-5.

[20] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM.

[21] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen. The adaptive web. chapter Collaborative Filtering Recommender Systems, pages 291–324. Springer-Verlag, Berlin, Heidelberg, 2007.

[22] C. Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[23] F. Thung, D. Lo, and J. Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191, Oct 2013.

[24] X. Xia, D. Lo, X. Wang, and B. Zhou. Tag recommendation in software information sites. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 287–296, Piscataway, NJ, USA, 2013. IEEE Press.

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