Automated approaches to assess the similarity of open source projects.

Rubei Riccardo

October 11, 2018

Contents

1 Introduction

1.1 Summary

Open source software (OSS) repositories contain a large amount of data that has been accumulated along the software development process. Not only source code but also metadata available from different related sources, e.g. communication channels, bug tracking systems, is beneficial to the development process once it is properly mined. Research has been performed to understand and predict software evolution, exploiting the rich metadata available at OSS repositories. This allows for the reduction of effort in knowledge acquisition and quality gain. Developers can leverage the underlying knowledge if they are equipped with suitable tools. For instance, it is possible to empower IDEs by means of tools that continuously monitor the developer's activities and contexts in order to activate dedicated recommendation engines [1].

To aim for software quality, developers normally build their project by learning from mature OSS projects having comparable functionalities. To this end, the ability to search for similar software projects with respect to different criteria such as functionalities and dependencies plays an important role in the development process. Two projects are deemed to be similar if they implement some features being described by the same abstraction, even though they may contain various functionalities for different domains [2]. Understanding the similarities between open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations [3], [4], thereby improving software quality. Meanwhile measuring the similarities between developers and software projects is a critical phase for most types of recommender systems [5], [6]. Similarities are used as a base by both content-based and collaborative-filtering recommender systems to choose the most suitable and meaningful items for a given item [3]. Failing to compute precise similarities means concurrently adding a decline in the overall performance of these systems. Nevertheless, measuring similarities between software systems has been considered as a daunting task [7], [2]. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail.

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

CROSSMINER[1] is a research project funded by the EU Horizon 2020 Research and Innovation Programme, aiming at supporting the development of complex software systems by *i)* enabling monitoring, in-depth analysis and evidence-based selection of open source components, and *ii)* facilitating knowledge extraction from large OSS reposito-

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

ries [8]. In the context of the project, we work towards an advanced Eclipse-based IDE providing intelligent recommendations that go far beyond the current *code completion-oriented* practice. Among others, an indispensable functionality is to find a set of similar OSS projects to a given project with respect to different criteria, such as external dependencies, application domain, or API usage [9], [10].

The purpose of this thesis is the implementation of two approaches, MUDABlue and Clan, with the aim of compare the results of new tool by CROSSMINER development team: (CrossSim). CROSSSIM (Cross Project Relationships for Computing Open Source Software Similarity), is an approach that makes use of graphs for rep-resenting different kinds of relationships in the OSS ecosystem. In particular, with the adoption of the graph representation, we are able to transform the relationships among non-human artifacts, e.g. API utilizations, source code, interactions, and humans, e.g. developers into a mathematically computable format, i.e. one that facilitates various types of computation techniques. Naturally this kind of approaches has to be evaluated, and confronted with others similar tools. My work helps addressing this challenge providing these two tools and evaluating all the results to show how nice is CrossSim.

2 Crossminer Project

2.1 CROSSMINER

2.1.1 Open Source Software Challenges

Open-source software (OSS) is computer software available in source code form, for which the source code and certain other rights are provided under a license that permits users to study, change, and improve the software for free. A report by Standish Group states that adoption of open-source software models has resulted in savings of about 58 billion per year to consumers. Unlike commercial software which is typically developed within the context of a particular organisation with a well-established business plan and commitment to the maintenance, documentation and support of the software, OSS is very often developed in a public, collaborative, and loosely-coordinated manner. This has several implications to the level of quality of different OSS software as well as to the level of support that different OSS communities provide to users of the software they produce. There are several high-quality and mature OSS projects that deliver stable and well-documented products. Such projects typically also foster a vibrant expert and user community, which provides remarkable levels of support both in answering user questions and in repairing reported defects in the provided software. However, there are also many OSS projects that are dysfunctional in one or more of the following ways:

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

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

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

2.1.2 Selecting Open Source Components

Deciding whether open source software (OSS) meets the required standards for adoption in terms of quality, maturity, activity of development and user support is not a straightforward process. It involves exploring various sources of information including:

- Its source code repositories to identify how actively the code is developed, how well the code is commented, whether there are unit tests etc.
- Communication channels such as newsgroups, forums and mailing lists to identify whether user questions are answered in a timely and satisfactory manner, to estimate the number of experts and users of the software
- Its bug tracking system to identify whether the software has many open bugs and at which rate bugs are fixed, and
- Other relevant metadata such as the number of downloads, the license(s) under which it is made available, its release history etc.

Dependence on an OSS project can thus either be a blessing or a curse. The ability to accurately assess the risks and benefits of adopting particular OSS projects is essential to the software community at large - especially open source software frameworks and platforms and highly specialised essential utility packages, which can make a depending product or service unexpectedly incur insurmountable technical difficulties when the OSS projects suddenly reach end-of-life.

2.1.3 Project Technologies

The overarching aim of CROSSMINER is to deliver an integrated open-source platform that will support the development of complex software systems by (1) enabling monitoring, in-depth analysis and evidence-based selection of open source components, and (2) facilitating knowledge extraction from large open-source software repositories. The six main scientific and technology objectives for the project are the following:

- Development of source code analysis tools to extract and store actionable knowledge from the source code of a collection of open-source projects
- Development of natural language analysis tools to extract quality metrics related to the communication channels, and bug tracking systems of OSS projects by using Natural Language Processing and text mining techniques
- Development of system configuration analysis tools to gather and analyse system configuration artefacts and data to provide an integrated DevOps-level view of a considered open source project
- Development of workflow-based knowledge extractors that simplify the development of bespoke analysis and knowledge extraction tools shielding engineers from technological issues to concentrate on core analysis tasks

- Development of cross-project relationship analysis tools to manage a wider range of open source project relationships, such as dependencies and conflicts, based on user-defined similarity measures underpinning the automated creation of project clusters.

- Development of advanced integrated development environments that will allow developers to adopt the CROSSMINER knowledge base and analysis tools directly from the development environment, while providing alerts, recommendations, and user feedback which will help developers to improve their productivity.

The outcomes of the different CROSSMINER analysis tools will contribute to the definition of a knowledge base supporting multidimensional classifications of projects and disclosing a number of applications such as automated identification of complementary and competing projects, project incompatibilities and prediction of the future of given projects based on the evolution of other projects having similar characteristics in the past.

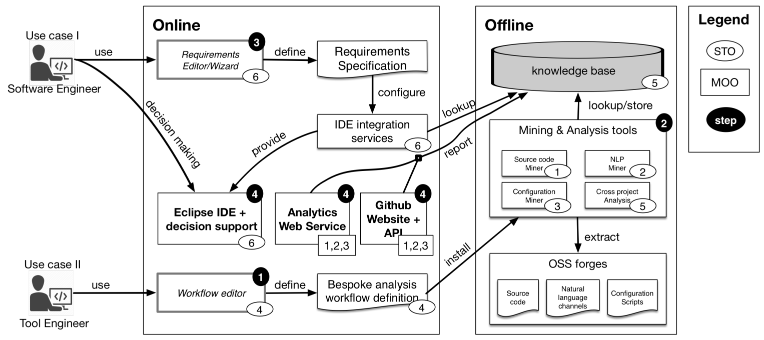


Figure 1: Crossminer Approach

According to such objectives and the overview about RSSE previously given, CROSSMINER can be seen as a recommendation system aimed at supporting developers while producing new software by integrating existing open source components. Figure 1 shows CROSSMINER at a glance and how the project aims at reaching its objectives.

Essentially, the *data preprocessing* challenge is addressed by the CROSSMINER components contained in the `Offline` box shown on the right-hand side of Fig.1. The developer context is captured by the IDE (see the `Online` box in Fig.1), which also produces recommendations that do not require particular and expensive data analysis. For more elaborated recommendations, preprocessed data available in the `Knowledge Base` is used. Both the IDE and Web based dashboards will be used to present the produced recommendations to the developer.

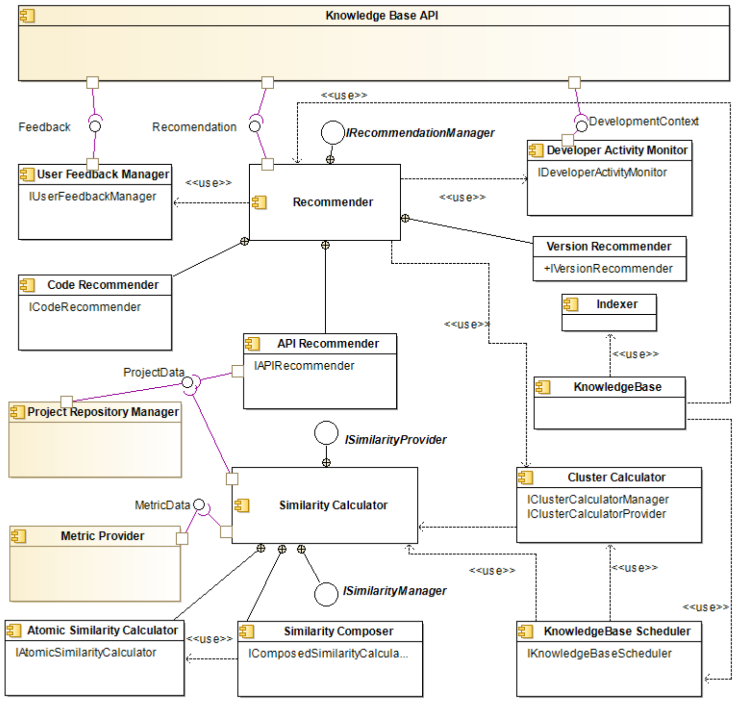


Figure 2: Component Diagram of the Crossminer Knowledge Base

`KnowledgeBase` The *KnowledgeBase* component is responsible for setting up the other components, as well as for their execution.

`Recommender` The *Recommender* component has the responsibility of creating recommendations in response to user requests. This component is extensible in order to add the management of specific kinds of artifacts. The model shows three specific recommender components: the `CodeRecommender` is able to provide developers with code examples that can be examined to solve the particular problem at hand (e.g., the correct usage of a used API). The `VersionRecommender` component is able to suggest developers the correct version of a used third party component. The `APIRecommender` implements the mechanisms for retrieving from the KB information about APIs that should be used in the project being implemented.

`ClusterCalculator` It has the responsibility of calculating clusters of analyzed artifacts as discussed in the previous sections. To this end similarity functions are used as implemented by the `SimilarityCalculator` component, which is in charge of managing the execution of both atomic and composed similarity calculations (see the `AtomicSimilarityComposer` and `SimilarityComposer` components, respectively).

`KnowledgeBaseScheduler` The calculation of clusters and thus the execution of the available similarity functions is performed periodically and it is triggered by the *KnowledgeBaseScheduler* component.

`UserFeedbackManager` It manages the feedback expressed by the user on received recommendation. Such a component is used by `Recommender` to take into account previous user feedback when creating future suggestions. Specific effort will be spent to understand how to identify developers e.g., to remember previous feedback at a user-level.

`DeveloperActivityMonitor` It stores and manages the data about the activity of the developer while using the CROSSMINER IDE.

`Indexer` The component implements indexing mechanisms to provide performance gains in the retrieval of data.

3 The Similarity Problem

3.1 Overview

Text similarity measures play an increasingly important role in text related research and applications in tasks such as information retrieval, text classification, document clustering, topic detection, topic tracking, questions generation, question answering, essay scoring, short answer scoring, machine translation, text summarization and others. Finding similarity between words is a fundamental part of text similarity which is then used as a primary stage for sentence, paragraph and document similarities. There two way in which words can be similar each other, lexically if they share sequences of characters similar and semantically if are used in the same context, used in the same way and so on.

3.2 String-Based

The world of lexical similarity con be divided in two categories: character-based and word-based. To better understand what character-based means, here one of the most well known technique: Levenshtein distance.

3.2.1 Levenshtein distance

Levenshtein distance defines distance between two strings by counting the minimum number of operations needed to transform one string into the other, where an operation is defined as an insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters.

This is an example:

- kitten to sitten (substitution of ”s” for ”k”).
- sitten to sittin (substitution of ”i” for ”e”).
- sittin to sitting (insertion of ”g” at the end).

Moving to the word-based, the word or string similarity measures operate on string sequences and character composition. A string metric is a metric that measures similarity or dissimilarity (distance) between two text strings for approximate string matching or comparison.

3.2.2 Cosine Similarity

Cosine similarity is a metric used to compute similarity between two objects using their feature vectors [11]. An object is characterized as a vector, and for a pair of vectors $\vec{\alpha} = (\alpha_1, \alpha_2, .., \alpha_n)$ and $\vec{\beta} = (\beta_1, \beta_2, .., \beta_n)$ there is an angle between them. Intuitively, the cosine similarity metric measures the similarity as the cosine of the corresponding angle between the two vectors and it is computed using the inner product as follows.

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

Figure 3 illustrates the cosine similarity between two vectors $\vec{\alpha}$ and $\vec{\beta}$ in a three-dimension space. This can be thought as the similarity between two documents with three terms $t = (t_1, t_2, t_3)$.

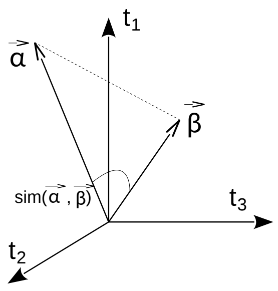


Figure 3: Cosine similarity between two feature vectors $\vec{\alpha}$ and $\vec{\beta}$

Cosine similarity has been popularly adopted in many applications that are related to similarity measurement in various domains [12], [13], [14], [15], [16]. Among the similarity metrics being recalled in this deliverable, the prevalence of Cosine Similarity is obvious as it is utilized in almost all of them as follows: *MUDABlue* [17], *CLAN* [2], *CLANdroid* [18], *LibRec* [19], *SimApp* [7], *WuKong* [20], *TagSim* [21], and *RepoPal* [4].

This is an example of how the cosine similarity can be done between two sentences. These are the two string that we want to compare to see how much they are related each other.

- Julie loves me more than Linda loves me.
- Jane likes me more than Julie loves me.

$$
\begin{array}{c}
 \\ me \\ Jane \\ Julia \\ Linda \\ likes \\ loves \\ more \\ than
\end{array}
\begin{array}{c}
\begin{array}{cc} string1 & string2 \end{array} \\
\begin{pmatrix}
2 & 2 \\
0 & 1 \\
1 & 1 \\
1 & 0 \\
0 & 1 \\
2 & 1 \\
1 & 1 \\
1 & 1
\end{pmatrix}
\end{array}
$$

Table 1: The occurrencies.

From the strings is possible to count the occurrencies of each term, putting everything in a matrix.

Since in this kind of evaluation is not important the meaning or where the words are, is possibile to create the related vectore in order to compute the similarity.

$$String1 = [2, 0, 1, 1, 0, 2, 1, 1]$$

$$String2 = [2, 1, 1, 0, 1, 1, 1, 1]$$

Applying the cosine similarity formula this is the outcome:

$$CosineSim(\vec{\alpha}, \vec{\beta}) = \frac{9}{\sqrt{12} \cdot \sqrt{10}} = 0.822 \tag{2}$$

This means that these strings are close each other 0.822, in a range bewteen 0.0 and 1.0.

3.3 Corpus-Based

3.3.1 Term-Document Matrix

In Natural Language Processing [22], a term-document matrix (TDM) is used to represent the relationships between words and documents [23]. In a TDM, each row corresponds to a document and each column corresponds to a term. A cell in the TDM represents the weight of a term in a document. The most common weighting scheme used in document retrieval is the *term frequency-inverse document frequency (tf-idf)* function [24]. If we consider a set of n documents $D = (d_1, d_2, .., d_n)$ and a set of terms $t = (t_1, t_2, .., t_r)$ then the representation of a document $d \in D$ is vector $\vec{\delta} = (w_1^d, w_2^d, .., w_r^d)$, where the weight w_k^d of term k in document d is computed using the *tf-idf* function [25]:

$$w_k^d = tf \cdot idf(k, d, D) = f_k^d \cdot log \frac{n}{|\{d \in D : t_k \in d\}|} \tag{3}$$

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

Another common weighting scheme uses only the frequency of terms in documents for cells in TDM, i.e. the number of occurrence of a term in a document, instead of *tf-idf*. As an example, we consider a set of three simple documents $D = (d_1, d_2, d_3)$ as follows:

+ d_1: *She is nice.*

+ d_2: *Today is nice.*

+ d_3: *Nice is a nice city.*

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \end{array} \begin{array}{c} \begin{array}{cccccc} \textit{she} & \textit{is} & \textit{today} & \textit{a} & \textit{nice} & \textit{city} \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 2 & 1 \end{pmatrix} \end{array}$$

Table 2: An example of a term-document matrix

The set of terms t consists of 6 elements, i.e. $t = (\textit{she}, \textit{is}, \textit{today}, \textit{a}, \textit{nice}, \textit{city})$ and the corresponding term-document matrix for D is depicted in Figure 7.

TDM has been exploited to characterize software systems and finally to compute similarities between them [17], [18], [2]. In a TDM for software systems, each row represents a package, an API call or a function and each column represents a software system. A cell in the matrix is the number of occurrence of a package/an API/function in each corresponding software system. A TDM for software systems has a similar form to the matrix shown in Figure 7 where documents are replaced by software systems and terms are replaced by API calls.

3.3.2 Latent Semantic Analysis

The problem with the term-document matrix is that the intrinsic relationships among different terms of a document cannot fully be captured. Furthermore, same words can be used to explain different requirements or the other way around, the same requirements can be described using different words [17]. Latent Semantic Analysis (LSA), also known as Latent Semantic Indexing (LSI), has been proposed to overcome these problems [26]. The technique exploits a mathematical model that can infer latent semantic relationships to compute similarity. LSA represents the contextual usage meaning of words by statistical computations applied to a large corpus of text. It then generates a representation that captures the similarity of words and text passages. To perform LSA on a text, a term-document matrix is created to characterize the text. Afterwards, Singular Value Decomposition (SVD) - a matrix decomposition technique - is used in combination with LSA to reduce matrix dimensionality [27]. SVD takes a highly variable set of data entries as input and transforms to a lower dimensional space but reveals the substructure of the original data. Essentially, it decomposes a rectangular matrix into the product of three other matrices as given below [27]:

$$A_{mn} = U_{mm} S_{mn} V_{mn}^T \tag{4}$$

in which

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

U_{mm} describes the original row entities as vectors of derived orthogonal factor values. S_{mn} represents the original column entities in the same way, and V_{mn} is a diagonal matrix containing scaling values. With the application of LSA it is possible to find the most relevant features and remove the least important ones by means of the reduced matrix U_{mm}. As a result, an equivalence of A_{mm} can be constructed using the most relevant features. LSA helps reveal the latent relationship among words as well as among passages which cannot be guaranteed by a simple term-document matrix. The similarity measurement by LSA reflects adequately human perception of similarity and association among texts. Using LSA, similarities among documents are measured as the cosine of the angle between their row vectors (see Sec. **??**). LSA has been applied in [17], [18], [2] to compute similarities of software systems. The main disadvantage of LSA is that it is computational expensive when a large amount of information is analyzed.

An example. Image that these are a set of document to be analyzed and we want to apply the procedure stated before.

- doc1: Human machine interface for ABC computer applications
- doc2: A survey of user opinion of computer system response time
- doc3: The EPS user interface management system
- doc4: System and human system engineering testing of EPS
- doc5: Relation of user perceived response time to error measurement
- doc6: The generation of random, binary, ordered trees
- doc7: The intersection graph of paths in trees
- doc8: Graph minors IV: Widths of trees and well-quasi-ordering
- doc9: Graph minors: A survey

$$
\begin{array}{l} \\ human \\ interface \\ computer \\ user \\ system \\ response \\ time \\ EPS \\ survey \\ trees \\ graph \\ minors \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} doc1 & doc2 & doc3 & doc4 & doc5 & doc6 & doc7 & doc8 & doc9 \end{array} \\
\left(\begin{array}{ccccccccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 2 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{array}\right)
\end{array}
$$

Table 3: Term-Document matrix related to the example.

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

Table 4: U_{mm}x

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

Table 5: S_{mn}

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

Table 6: V_{mn}^T

The following image depict the result of the decomposition with a rank of 2.

$$\begin{array}{l|ccccccccc|}
 & doc1 & doc2 & doc3 & doc4 & doc5 & doc6 & doc7 & doc8 & doc9 \\
human & 0.16 & 0.40 & 0.38 & 0.47 & 0.18 & -0.05 & -0.12 & -0.16 & -0.09 \\
interface & 0.14 & 0.37 & 0.33 & 0.40 & 0.16 & -0.03 & -0.07 & -0.10 & -0.04 \\
computer & 0.15 & 0.51 & 0.36 & 0.41 & 0.24 & 0.02 & 0.06 & 0.09 & 0.12 \\
user & 0.26 & 0.84 & 0.61 & 0.70 & 0.39 & 0.03 & 0.08 & 0.12 & 0.19 \\
system & 0.45 & 1.23 & 1.05 & 1.27 & 0.56 & -0.07 & -0.15 & -0.21 & -0.05 \\
response & 0.16 & 0.58 & 0.38 & 0.42 & 0.28 & 0.06 & 0.13 & 0.19 & 0.22 \\
time & 0.16 & 0.58 & 0.38 & 0.42 & 0.28 & 0.06 & 0.13 & 0.19 & 0.22 \\
EPS & 0.22 & 0.55 & 0.51 & 0.63 & 0.24 & -0.07 & -0.14 & -0.20 & -0.11 \\
survey & 0.10 & 0.53 & 0.23 & 0.21 & 0.27 & 0.14 & 0.31 & 0.44 & 0.42 \\
trees & -0.06 & 0.23 & -0.14 & -0.27 & 0.14 & 0.24 & 0.55 & 0.77 & 0.66 \\
graph & -0.06 & 0.34 & -0.15 & -0.30 & 0.20 & 0.31 & 0.69 & 0.98 & 0.85 \\
minors & -0.04 & 0.25 & -0.10 & -0.21 & 0.15 & 0.22 & 0.50 & 0.71 & 0.62
\end{array}$$

Table 7: Matrix decomposed.

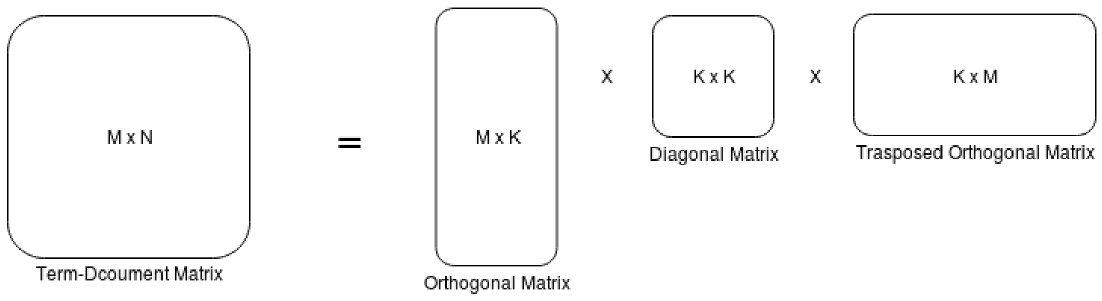


Figure 4: Reduction phase

Figure 4 depicts the low rank reduction phase and is interesting to graphically see what does it mean. The new matrix is the product of the other three, but reducted, this is a very relevant issue. If the singular values in S_{mn} are ordered by size, the first k largest may be kept and the remaining smaller ones set to zero. The product of the resulting matrices is a matrix X which is only approximately equal to A_{mm} , and is of rank k . It can be shown that the new matrix X is the matrix of rank k which is closest in the least squares sense to A_{mm} . The amount of dimension reduction, i.e., the choice of k , is critical to our work. Ideally, we want a value of k that is large enough to fit all the real structure in the data, but small enough so that we do not also fit the sampling error or unimportant details. The proper way to make such choices is an open issue in the factor analytic literature. In practice, we currently use an operational criterion - a value of k which yields good retrieval performance.
In our work we decided a $k\ value = \frac{repository}{2}$

3.4 Knowledge-Based

Knowledge-Based Similarity aims to identify the degree of similarity between words using informations derived from semantic networks. Knowledge-based similarity measures can be divided roughly into two groups: measures of semantic similarity and measures of semantic relatedness.By semantic similarity we mean concepts that are related each other on the basis of their likeness. Semantic relatedness, on the other hand, is a more general notion of relatedness, not specifically tied to the shape or form of the concept. Semantic similarity is a metric defined over a set of documents or terms, where the idea of distance between them is based on the likeness of their meaning or semantic content as opposed to similarity which can be estimated regarding their syntactical representation (e.g. their string format). An example of relatedness is the Lask algorithm which identify senses of words in context using definition overlap. To clarify let's take a look an example in [28]. Using the Oxford Advanced Learner's Dictionary, it finds that word *pine* has two senses:

- Sense 1: kind of **evergreen tree** with needle-shaped leaves
- Sense 2: waste away through sorrow ir illness.

The word *cone* has three senses:

- Sense 1: solid body which narrows to a point
- Sense 2: something of this shape whether solid or hollow
- Sense 3: fruit of certain **evergreen tree**

Each of the two senses of the word *pine* is compared with each of the three senses of the *cone* and it is found that the words *evergreen tree* occurs in one sense each of the two words. These two senses are then declared to be the most appropriate senses when words *pine* and *cone* are used togheter.

Concerning the similarity an example could be Resnik(1995) which uses the information content of concepts, computed from their frequency of occurrence in a large corpus, to determine the semantic relatedness of word senses [28]. Another example is Jian & Conrath [29]s It combines a lexical taxonomy structure with corpus statistical information so that the semantic distance between nodes in the semantic space constructed by the taxonomy can be better quantified with the computational evidence derived from a distributional analysis of corpus data. Specifically, the proposed measure is a combined approach that inherits the edge-based approach of the edge counting scheme, which is then enhanced by the node-based approach of the information content calculation.

4 The Approaches

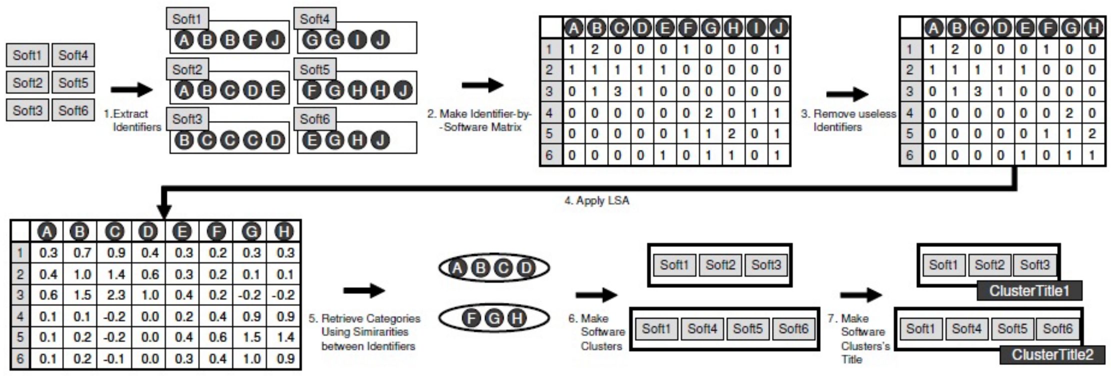
4.1 Overview

In this chapter we are going to illustrate the approaches that we have implemented, in particular we will discuss about MudaBlue,Clan, RepoPal and CrossSim. As explained before, the purpose is to provide a baseline to evaluate CrossSim, so it's mandatory to have a precise idea of what these approaches are and how does they work. Concerning MudaBlue and Clan, we will discuss about their rationale, showing also their original results, then we will show how we reimplemented such approaches from an high-level point of view.

4.2 MUDABlue

The first procedure analysed was MUDABlue, unfortunately none implentation was available on the web, so i reimplemented it from scratch. The MUDABlue method is an automatc categorizaton method of a large collecton of software systems. MUDABlue method does not only categorize sooware systems but also determines categories from the software systems collection automatigcally. MUDABlue has three major aspects: 1) it relies on no other information than the source code, 2) it determines category sets automatically, and 3) it allows a software system to be a member of multiple categories. Since we were interested only in the evaluation of the similarity we discarded the phases related to clusterization and categorization.

The MUDABlue approach can be briefly summarized in 7 steps, as the following image depicts:



	A	B	C	D	E	F	G	H	I	J
1	1	2	0	0	0	1	0	0	0	1
2	1	1	1	1	1	0	0	0	0	0
3	0	1	3	1	0	0	0	0	0	0
4	0	0	0	0	0	0	2	0	1	1
5	0	0	0	0	0	1	1	2	0	1
6	0	0	0	0	1	0	1	1	0	1

	A	B	C	D	E	F	G	H
1	1	2	0	0	0	1	0	0
2	1	1	1	1	1	0	0	0
3	0	1	3	1	0	0	0	0
4	0	0	0	0	0	0	2	0
5	0	0	0	0	0	1	1	2
6	0	0	0	0	1	0	1	1

	A	B	C	D	E	F	G	H
1	0.3	0.7	0.9	0.4	0.3	0.2	0.3	0.3
2	0.4	1.0	1.4	0.6	0.3	0.2	0.1	0.1
3	0.6	1.5	2.3	1.0	0.4	0.2	-0.2	-0.2
4	0.1	0.1	-0.2	0.0	0.2	0.4	0.9	0.9
5	0.1	0.2	-0.2	0.0	0.4	0.6	1.5	1.4
6	0.1	0.2	-0.1	0.0	0.3	0.4	1.0	0.9

Figure 5: MUDABlue phases.

4.2.1 Exctract Identifiers

With identifier we are talking about relevant strings that can allow to characterize a document. In this phase each repository is scanned in order to find the target files, and for each of them the identifiers are exctracted, avoiding adding useless items such as comments. The dataset was a 41C projects gathered from SourceForge.

4.2.2 Create identifier-by-software matrix

As stated before, the main item to work with is the term-document matrix, in this case we count how many times each term appears in each file for all the projects. The result is matrix **m x n** with m terms and n projects.

4.2.3 Remove useless identifiers

From the matrix we remove all the useless terms, that is all the terms that apperas in just one repository, considered a specific terms, and all the terms that appears in more than 50% of the repositories, considered as general terms.

4.2.4 Apply the LSA

Once the matrix is ready con be worked, the *SVD* procedure is applied and then the LSI. As explained before [NOTE] the *SVD* procedure decompose the original matrix in 3 other matrices. When we multiply back these matrices we use a rank reducted version of the S matrix in order to generete the final one. The authors didn't provide us any details about their final rank value, so we tested many values and eventually selected one.

4.2.5 Apply the Cosine Similarity

By using the cosine similarity method, we compare each repository vector with all the others and eventually getting an **n x n** matrix, in which is expressed the similarity of all the repository couple, with a value *[0.0-1.0]*. Thereafter, the cluster analysis is applied using calculated similarities.

4.2.6 Categorization

Make software clusters from identifier clusters. From each identifier clusters, the software systems that contain one or more identifiers in the cluster are retrieved. The last step is to make software clusters' titles. This can be done by summing all identifier-vectors comprised in the identifier cluster and then consider the ten identifiers that got the highest value in the summation vector.

4.2.7 Results

The experimentation was conducted on a corpus of 41 *C* projects, taken form SourceForge belonging to 5 categories. Developers used *precision* and *recall* as criteria, defined as follows:

$$Precision = \frac{\sum_{s \in S} \text{precision}_{soft}(\text{S})}{\mid S \mid} \tag{5}$$

$$Recall = \frac{\sum_{s \in S} \text{recall}_{soft}(\text{S})}{\mid S \mid} \tag{6}$$

$$Precision_{soft}(\text{s}) = \frac{\mid C_{\text{MudaBlue}}(\text{s}) \cap C_{\text{Ideal}}(\text{s}) \mid}{\mid C_{\text{MudaBlue}}(\text{S}) \mid} \tag{7}$$

$$Recall_{soft}(\text{s}) = \frac{\mid C_{\text{MudaBlue}}(\text{s}) \cap C_{\text{Ideal}}(\text{s}) \mid}{\mid C_{\text{Ideal}}(\text{s}) \mid} \tag{8}$$

where $C_{\text{MudaBlue}}(\text{s})$ is a set of categories containing software s, generated by MUDABlue, $C_{\text{Ideal}}(\text{s})$ is a set of categories containing software s, determined manually by the experimenters. In both of the criteria, the larger value, the better result.

4.3 CLAN: Closely reLated ApplicatioNs

CLAN [2] is an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to packages class hierarchies. *CLAN* works based on the document framework for computing similarity, semantic anchors, e.g. those that define the documents' semantic features. Semantic anchors and dependencies help obtain a more precise value for similarity computation between documents. The assumption is that if two applications have API calls implementing requirements described by the same abstraction, then the two applications are more similar than those that do not have common API calls. The approach uses API calls as semantic anchors to compute application similarity since API calls contain precisely defined semantics. The similarity between applications is computed by matching the semantics already expressed in the API calls.

The process consist of 12 steps here graphically reported.

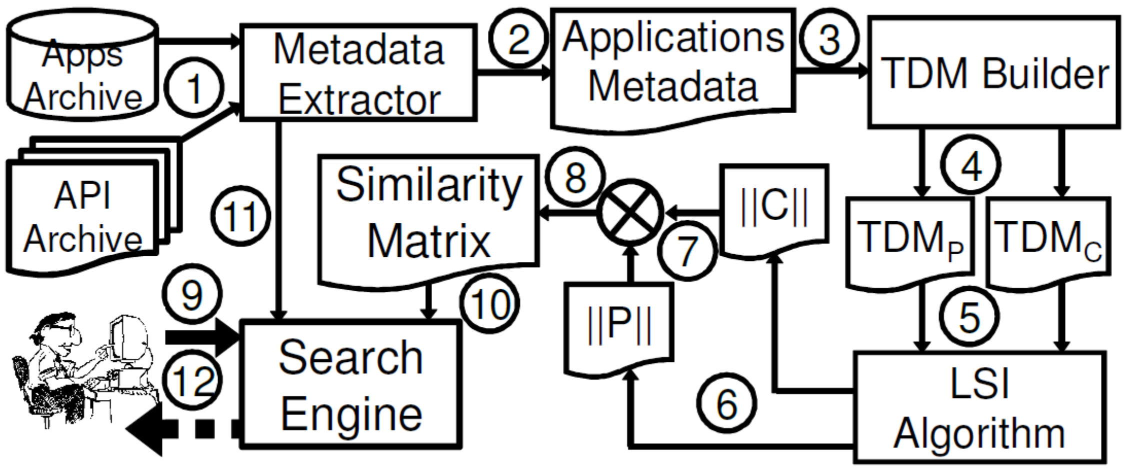


Figure 6: CLAN phases.

4.3.1 Terms Extraction

Steps from 1 to 3 can be merged together since are related to extraction of terms from the repositories. As stated before, an important concept is that terms extracted are only API calls, this means that all other things present in a piece of code are discarded, for example all the variables or the function declaration and invocation. Furthermore these API calls belong only to the JDK, in such a way also the calls to any other external library are discarded. This idea is also applied in the extraction of the import declaration, focus only on the JDK packages import. The result of this process will be an ordered set of data, representing the occurrencies of any Package;Class for all the projects.

4.3.2 TDMs Creation

Once the dataset as been created, is reorganized in TDMs. Here two different matrices are created, one for the Classes and one for the Packages. Class-level and package-level similarities are different since applications are often more similar on the package level than on the class level because there are fewer packages than classes in the JDK. Therefore, there is the higher probability that two applications may have API calls that are located in the same package but not in the same class.

4.3.3 LSI Procedure

The paper refers to LSI procedure, Latent Semantic Indexing[dumais2], but the term are synonym, so from here on, we will refer as Latent Semantic Analysis LSA.

4.3.4 Apply the Cosine Similarity

As for Mudablue, we will apply the cosine similarity to the matrix got from the LSA procedure.

4.3.5 Sum of the matrices

The 2 matrices are summed, but before are multplied by a certain value. Since the values for the entries in the 2 matrices are between 0.0 and 1.0 a simple sum could result in a value over 1.0, by this multiplication these values are reducted in order to be summed togheter but still maintaining the logical meaning. The authors chosen 0.5, also we, since is a good value to equal distribute the weight of the packages and method calls.The sum of this value is 1.0, and can span from 0.1 to 0.9 for each matrix, is clear that more is high on a matrix, more is important the values that we are considering from such matrix.

4.3.6 Final similarity matrix

Once the matrix is ready, the system will use it to answer the query of users, from such matrix the system will retrieve the common projects ordered by rank.

4.3.7 Results

The developers used a corpus of 8310 projects from SourceForge, for a total of 114146 API calls. The evaluation method was similar to our, a user study with a group of 33 student of University of Illinois at Chicago with at least 6 months of java experience. Their main task was to examine the retrieved applications and to determine if they are relevant to the tasks and the source application. Each participant accomplished this step individually, assigning a confidence level, C, to the examined applications using a four-level Likert scale.

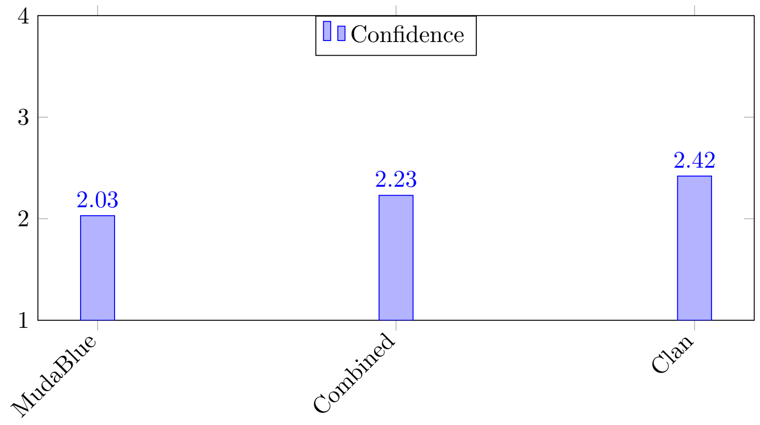


Figure 7: Confidence Comparison Original Clan

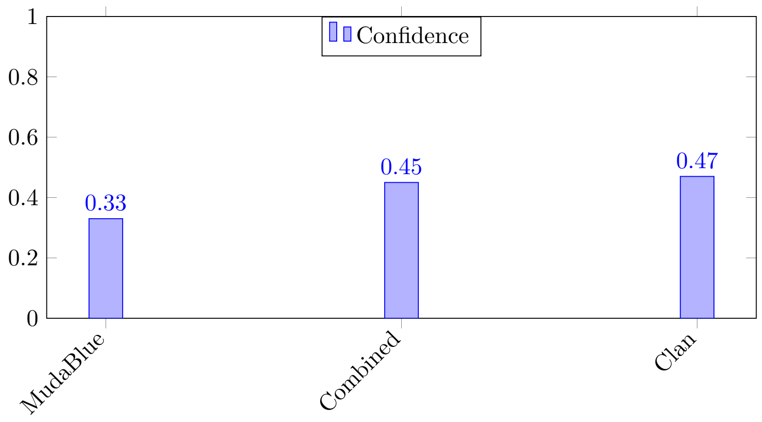


Figure 8: Precision Comparison Original Clan

Figure 7 and figure 8 reports the mean of the results they got from the students. As you can see the Clan approach performs better than MudaBlue. Since Similarity Matrices of MUDABlue and CLAN have the same dimensions, it is possible to construct a combined matrix whose values are the average of the values of the MUDABlue and CLAN matrix elements at the corresponding position. [nota e chiedere]

4.4 RepoPal: Exploiting Metadata to Detect Similar GitHub Repositories

In contrast to many previous studies that are generally based on source code [17], [30], [2], *RepoPal* [4] is a high-level similarity metric and takes only repositories metadata as its input. With this approach, two GitHub repositories are considered to be similar if:

i) They contain similar readme files;

ii) They are starred by users of similar interests;

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

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

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

There are three similarity indices as follows:

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

$$sim_f(r_i, r_j) = CosineSim(\vec{f_i}, \vec{f_j}) \tag{9}$$

4.5 CrossSim

Since the purpose of this thesis is to provide implementation and data to validate CrossSim approach, is useful to explain in detail it. So this section we are going to present CROSSSIM (Cross Project Relationships for Computing Open Source Software Similarity), an approach that makes use of graphs for representing different kinds of relationships in the OSS ecosystem. In particular, with the adoption of the graph representation, we are able to transform the relationships among non-human artifacts, e.g. API utilizations, source code, interactions, and humans, e.g. developers into a mathematically computable format, i.e. one that facilitates various types of computation techniques.

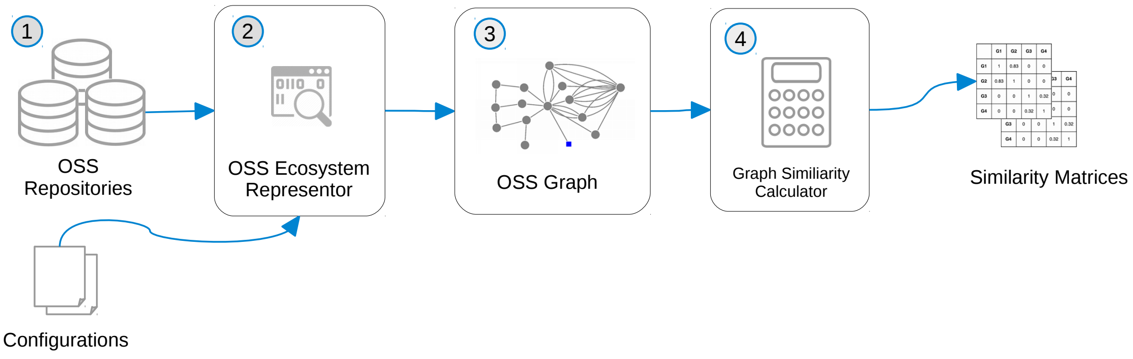


Figure 9: CrossSim Structure

The architecture of CrossSim is depicted in Fig. 9: 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.

4.5.1 Graph-based Representation of OSS Ecosystems

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

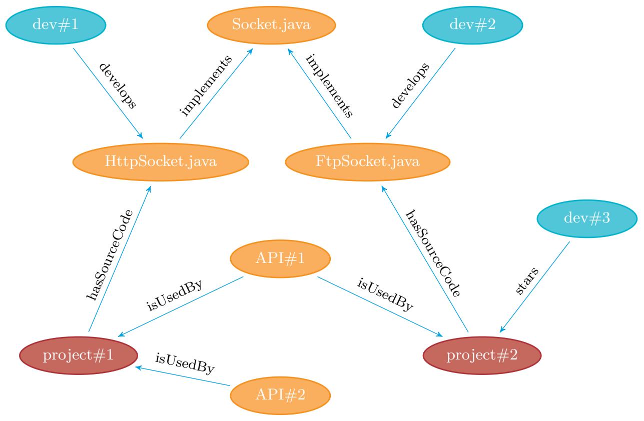


Figure 10: Sample graph-based representation of OSS ecosystems

makes use of graphs for representing relationships in OSS ecosystems. Specifically, the graph model has been chosen since it allows for flexible data integration and facilitates numerous similarity metrics and clustering techniques [31], [32], [33]. All the playing actors and their communications are transformed into a directed graph. Humans and non-human artifacts are represented as nodes and there is a directed edge between a pair of nodes if they interact with each others. The representation model considers different artifacts in a united fashion, taking into account their mutual, both direct and indirect relationships as well as their co-occurrence as a whole. The representation is twofold: First, it incorporates semantic relationships into the graph. Second, it helps combine both low-level and high-level information into a homogeneous representation.

To demonstrate the utilization of graphs in an OSS ecosystem, we consider an excerpt of the dependencies for two OSS projects, namely `project#1` and `project#2` in Figure 10. Using dependency information extracted from source code and the corresponding metadata (e.g. coming from the tools developed in by Work Package 2), this graph can be properly built to represent the two projects as a whole. In this figure, `project#1` contains code file `HttpSocket.java` and `project#2` contains `FtpSocket.java` with the corresponding edges being marked with the semantic predicate `hasSourceCode`. Both source code files implement `interface#1` being marked by the semantic predicate `implements`. `Project#1` and `project#2` are also connected via other semantic paths, such as API `isUsedBy` highlighted in Figure 11. 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 [31], [34], [35]. The graph structure also allows for graph kernel methods, which are an effective way to compute similarity [36]. Considering Figure 10, we can compute the similarity between `project#1` and `project#2` with regards to the semantic paths between them, e.g.

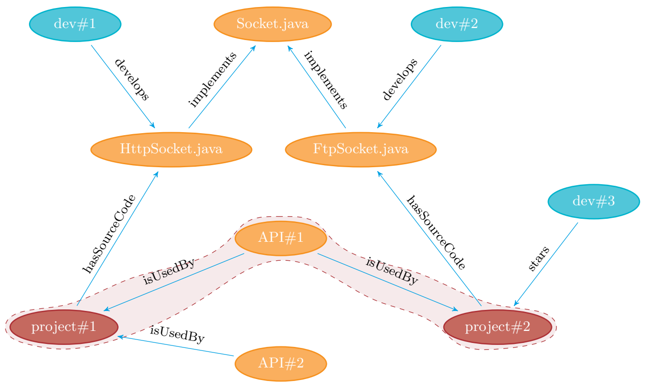


Figure 11: Similarity between OSS projects with respect to API usage

the two-hop path using `hasSourceCode` and `implements` (Figure 12), or the one-hop path using API `isUsedBy`. For example, concerning `isUsedBy`, the two projects are considered to be similar since with the predicate both projects originate from `API#1`. The hypothesis is based on the fact that the projects are aiming at creating common functionalities by using common libraries [2], [19].

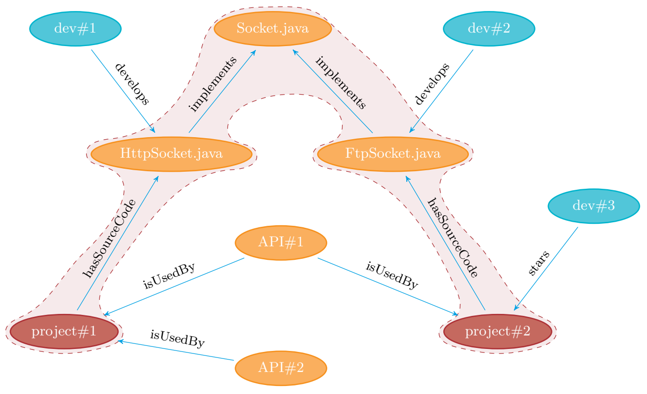


Figure 12: Similarity between OSS projects with respect to source implementation

The representation allows us to compute similarity between other graph components, e.g. developers. Back to Figure 10, though there is no direct connection between `developer#1` and `developer#2`, their similarity can still be inferred from indirect semantic paths, such as `develops` and `implements` which are highlighted in Figure 13. If we consider other semantic paths, we see that the two developers have more in common as they both take part in `project#1` and `projects#2` represented by `commits`. To a certain extent, the two developers are considered to be similar, although they are not directly connected. In reality, the connection between `developer#1` and `developer#2` is enforced by further semantic paths and as a result their similarity can be more precisely

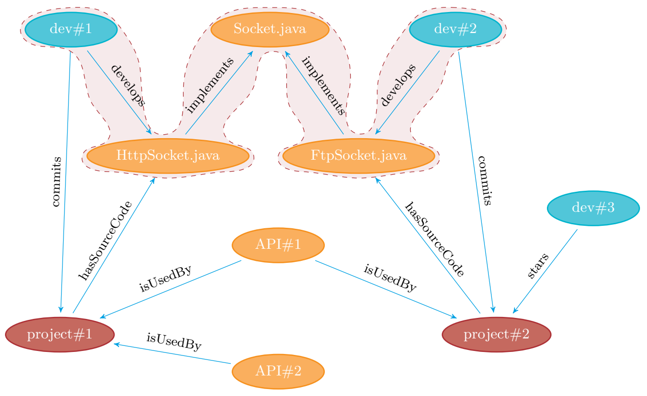


Figure 13: Similarity between developers

computed. The similarities between developers can serve as input for a collaborative filtering recommendation system, with which a developer is recommended a list of projects or libraries that similar developers already worked with [37], [3]. This is an invaluable tool in the context of the CROSSMINER project since it is essential to equip developers with recommendation functionalities to help them increase reusability and productivity.

4.6 System Description

In this section we want to explain our work and what we have done. In order to do this we will use some UML diagram plus a generich high level architecture diagram.

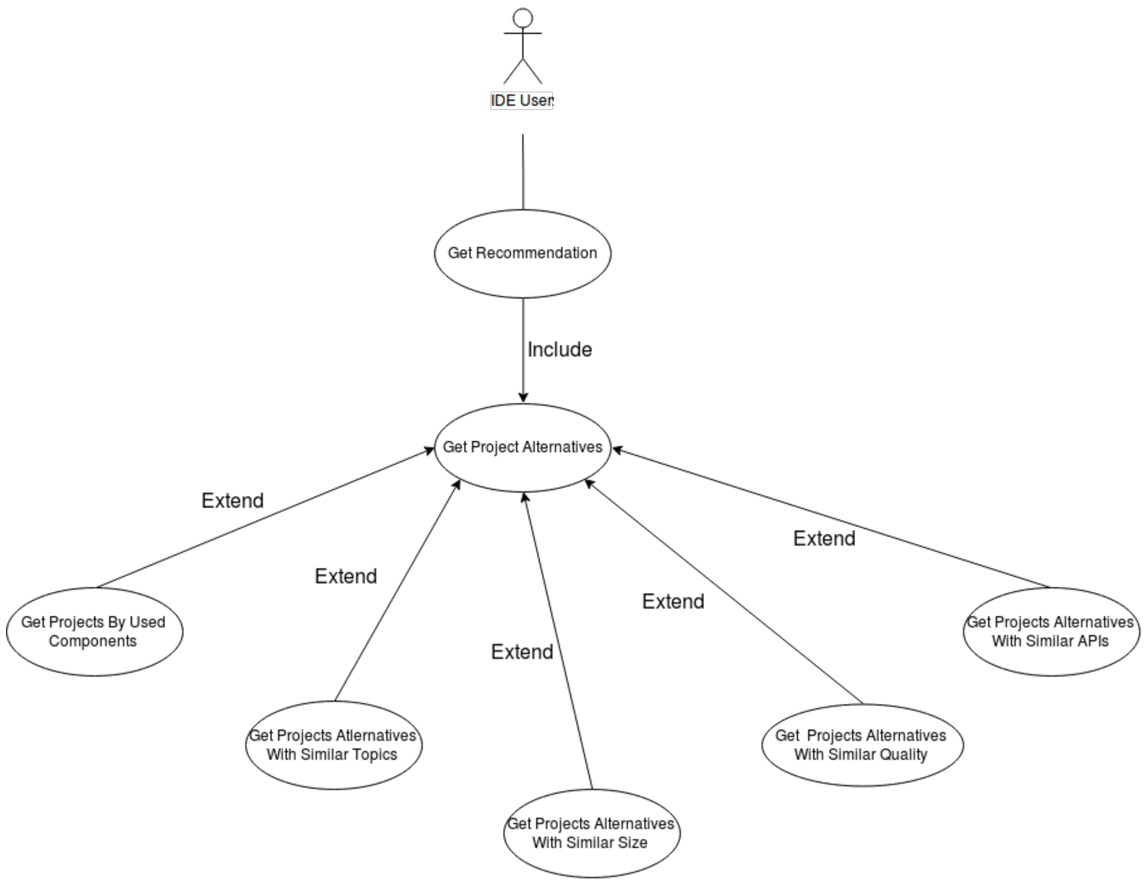

IDE User
Get Recommendation
Include
Get Project Alternatives
Extend
Extend
Extend
Extend
Extend
Get Projects By Used Components
Get Projects Atlernatives With Similar Topics
Get Projects Alternatives With Similar Size
Get Projects Alternatives With Similar Quality
Get Projects Alternatives With Similar APIs

Figure 14: Use Case Diagram

This image, exctracted from CrossSim documentation depicts how a similarity calculator fits inside a real project. It is clear that in order to provide a meaningful recommendation, it is mandatory to have a similarity calculator better as possible since all the functionality extending the *GetProjectAlternatives* use case, rely on some similarity computation.

4.6.1 Component Point of View

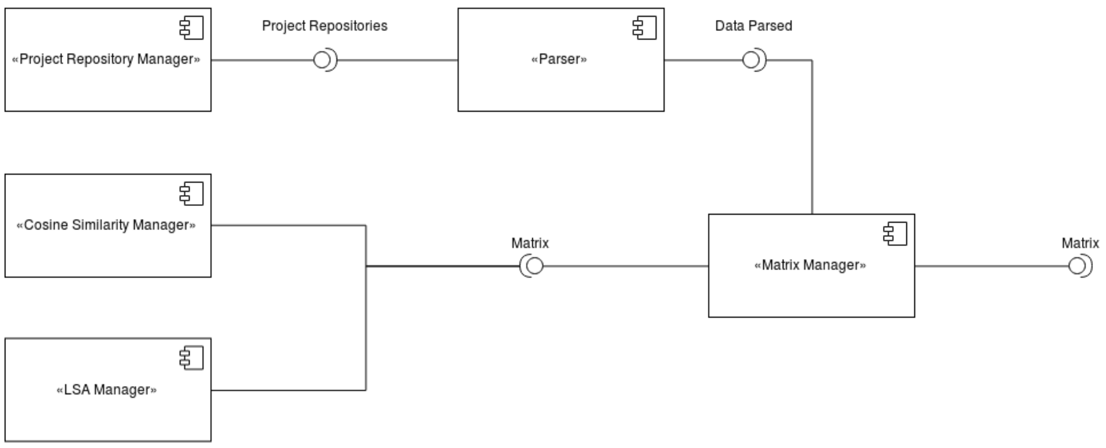


Figure 15: Component Diagram

As you can see there are 5 main components:

- Project Repository manager: this is the component who provide the repositories and who manage the file system.
- Parser: this component analyzes all the *.java* files in order to retrieve the keywords to create the term-document matrix. As stated before we search for the *JDK* related imports and methods for Clan and any imports, method, variables and field variables for MudaBlue.
- Matrix Manager: the matrix manager is the central component, in the sense that manages the creation of the term-document matrix, but not only, it coordinates all the matrices "roaming" during the process. For example the term-document matrix can't be analzed as it is by the SVD component, it requires a rework before.
- LSA Manager: here all the operations concerning the Latent Semantic Analsys occurs, from the low-rank matrix reduction to the Singular Value Decomposition.
- Cosine Similarity Manager: once the LSA completes his work we can apply the cosine similarity to get the final version of the matrix.

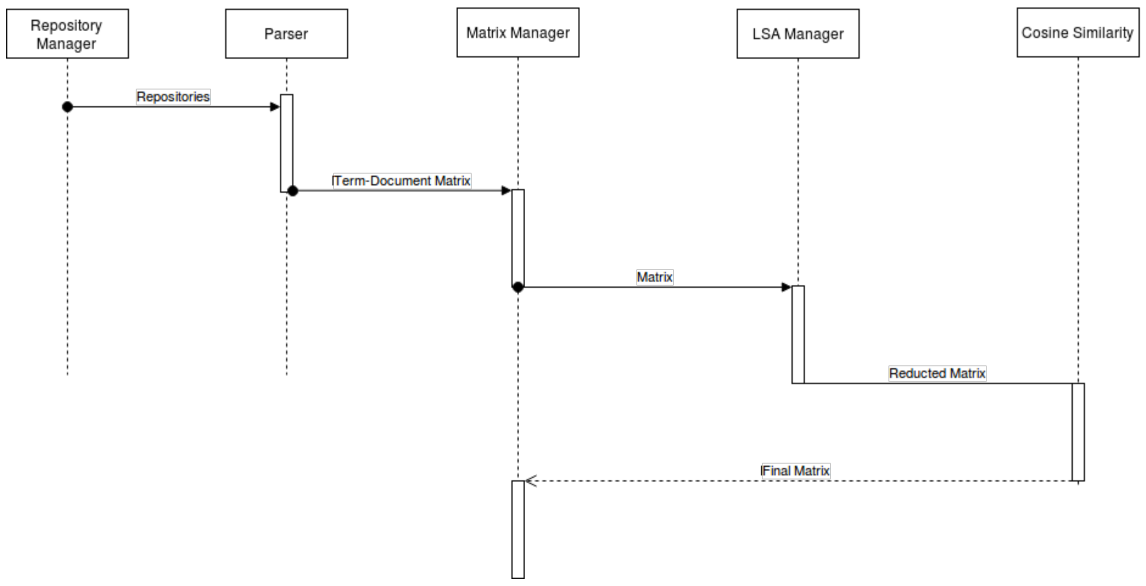


Figure 16: Sequence Diagram

In the figure 16 we can see the sequence diagram. When the process starts, the repository manager analyzes the file system in order to provide all the repositories to be analyzed. It also check if the parsing has been occurred before to such repositories, this is due the extremely high consumption of memory, so we splitted the phases in two moments.
When we know what are the repositories to be analyzed we can start, as explained before for MudaBlue and Clan the terms are different, but we still use the same library in both cases *Java Parser.*
The outcome will be a term-document matrix worked by the Latent Semantic Analysis manager. First of all we invoke the *commons math* for decomposing the matrix, then we can multiply them back in order the get the LSA matrix.
At this stage we just need to take the matrix and then apply the cosine similarity, for each vector of the matrix we calculate the cosine with all the others vectors. In this way we will get a final matrix of *580 x 580.* The final matrix can be taken for further analysis or anything that we need.

4.6.2 Description

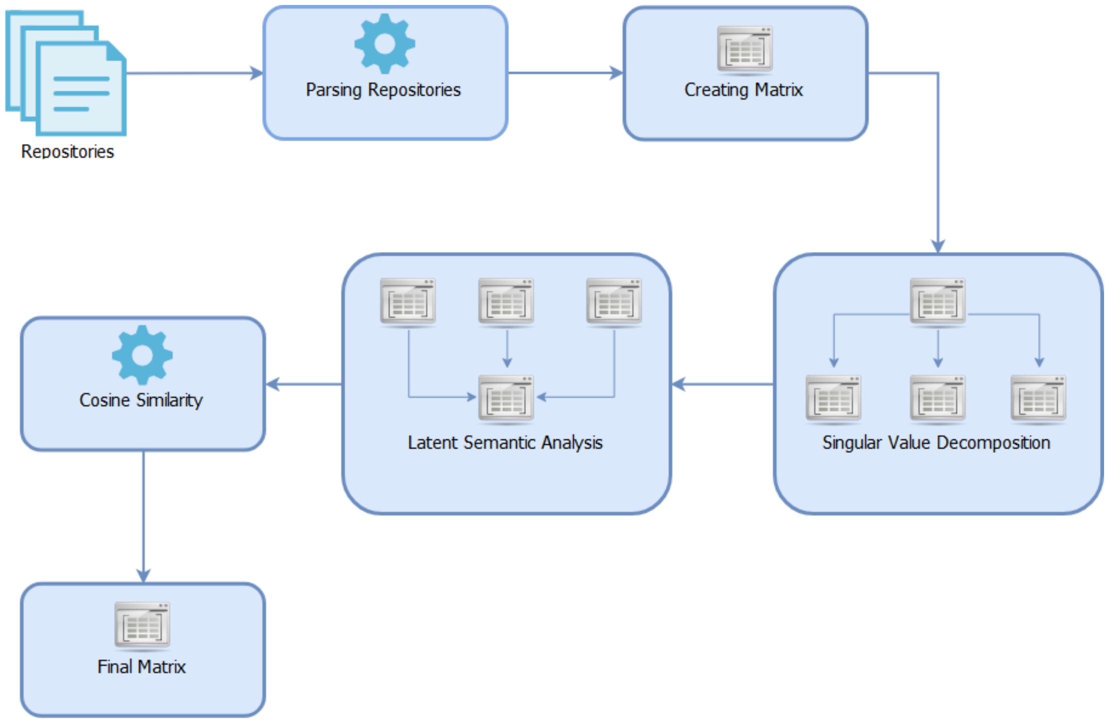


Figure 17: System Structure

In this image is depicted the general architecture of the implemented systems, as you can see the systems share the same architecture with some differences that will be discussed later. As you can see, the process consist of 7 steps.

- Retrieving the dataset, in this case a folder with all 580 repositories.
- All these repositories are analyzed, and any *.java* file is parsed.
- For each repository a vector that contains all the frequencies for each term found is created, and then added in a matrix.
- The SVD procedure, decomposing the matrix in other 3.
- The matrices are multiplied back to realize the LSA procedure.
- For each vector, we count the cosine similarity with all the others.
- Now we have the final matrix, where any repositories is compared to all the others.

4.6.3 System Details

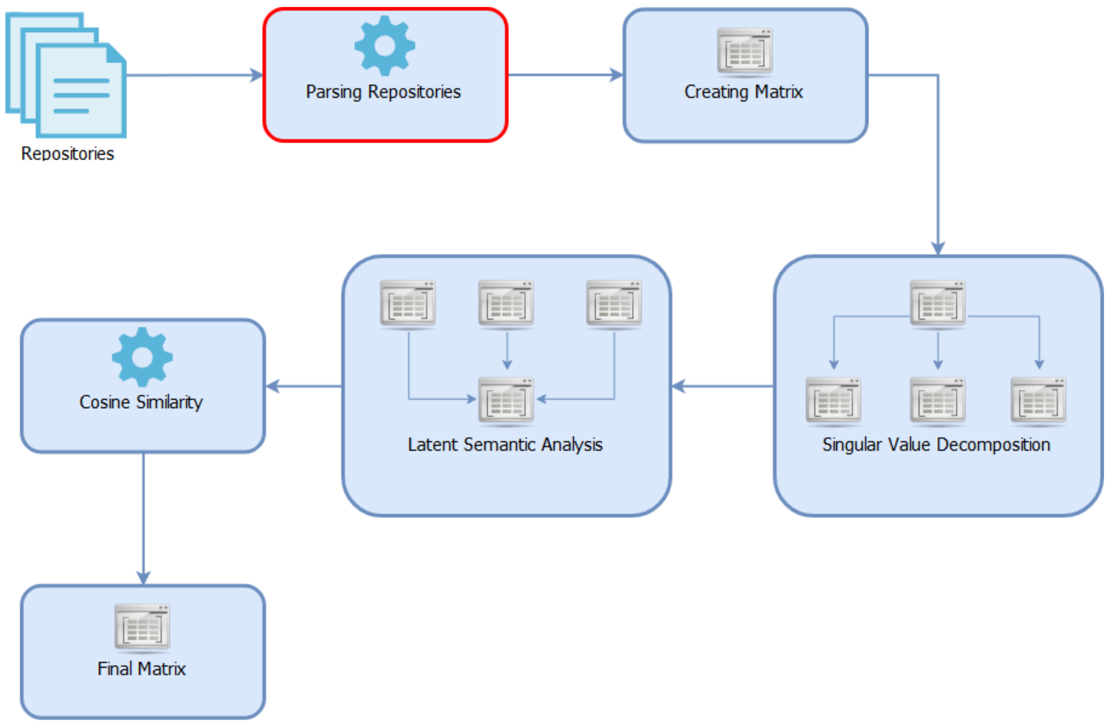


Figure 18: Parsing

Parsing: The first step is clearly parsing the java files of the 580 repositories. We used the javaparser library to directly access the main components of the files (import and method invocation for CLAN, import, method declaration, variables and field variables for MudaBlue). For each repository we created a relative .txt file containing the frequencies, for the CLAN approach such terms are filtered by searching only the terms belonging to the Java JDK. All these terms are merged in another file, called mainlist.txt which is used to avoid reps. The idea is parsing the files and compare with the mainlist.txt to add new terms, and then count, for each terms how many times appears inside the files. So the result will be a vector of numbers.

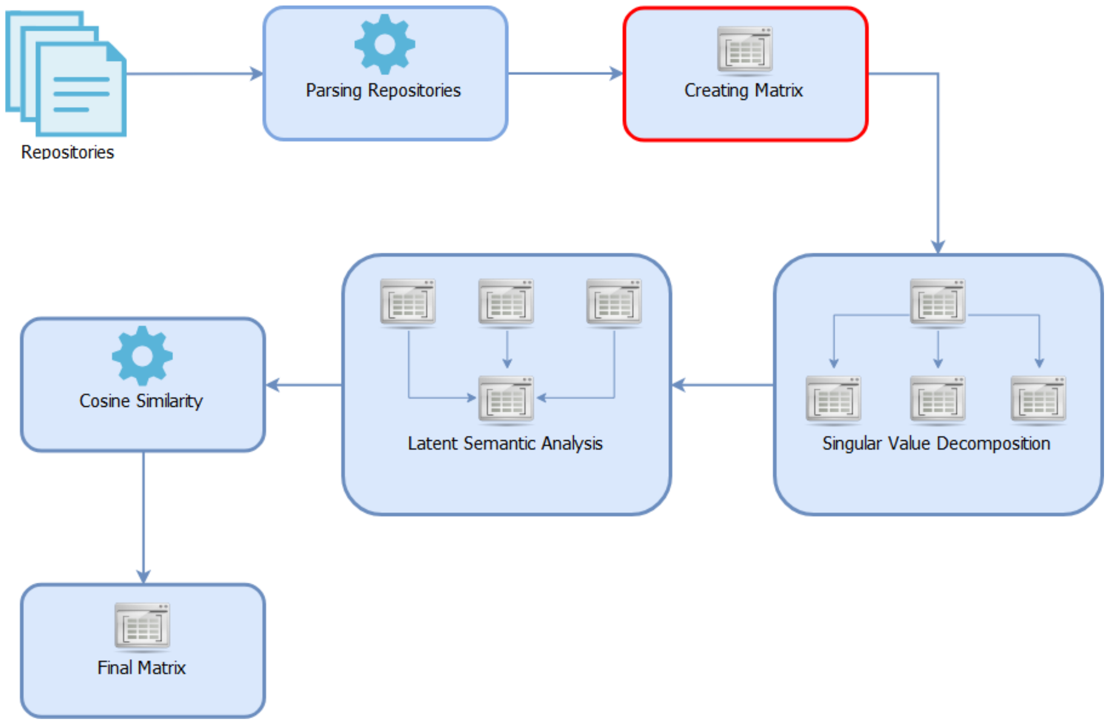


Figure 19: Matrix Creation

Matrix Creation: Once all the repositories are analyzed we can procede in creating the term-document matrix. The matrix is created using the library *apache commons math3* and in particular these components:

- ArrayRealVector.
- RealMatrix.
- RealVector.

Each files contains only his own terms naturally, so the idea is, once the parsing process is done, to count how many terms we have and then, adding many zeros as many terms are missing. To clarify, imagine that we have 3 documents A,B,C for 10 different terms. Now if we examine the document A, we might discover 4 terms, this means that the other 6 terms are missing here, so can be marked as 0. For the document B, we might find out 2 new terms, and so on.

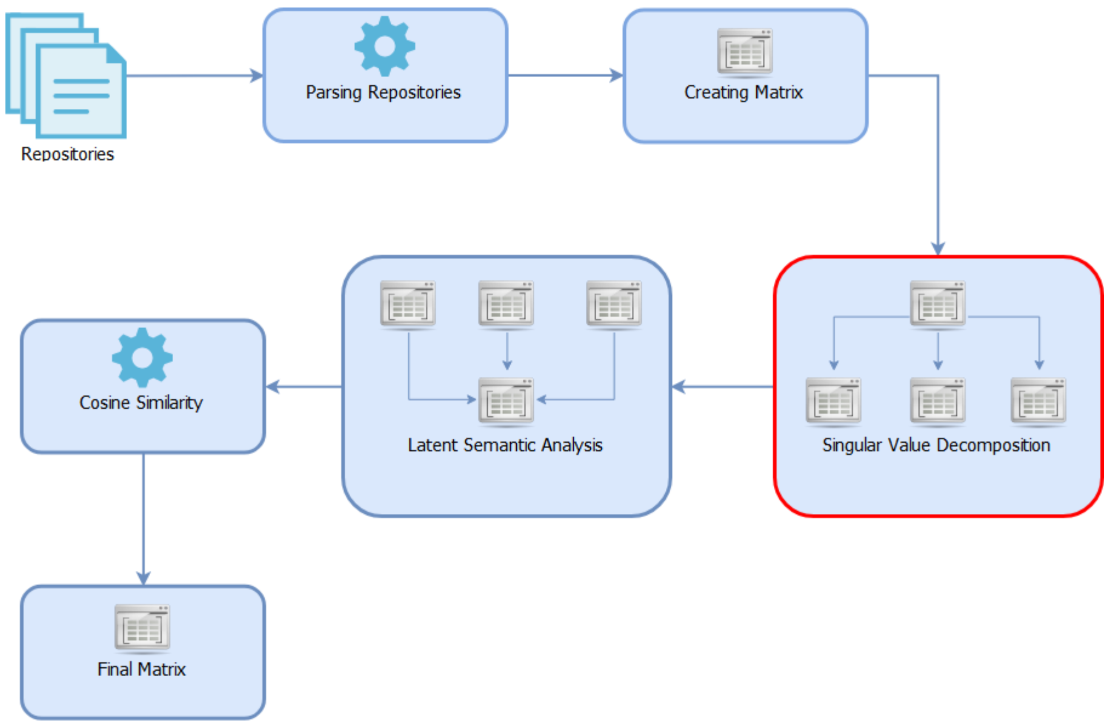


Figure 20: Singular Value Decomposition

SVD: As stated before the svd operation consist in decomposing the main matrix in other 3.

$$A_{mn} = U_{mm} S_{mn} V_{mn}^T \tag{10}$$

in which

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

Such operation are provided by *math3 linear SingularValueDecomposition*. So we invoke the methods passing as parameter the term-document matrix. As you can see this operation was already available in the library, so we just retrieved the results of the operation.

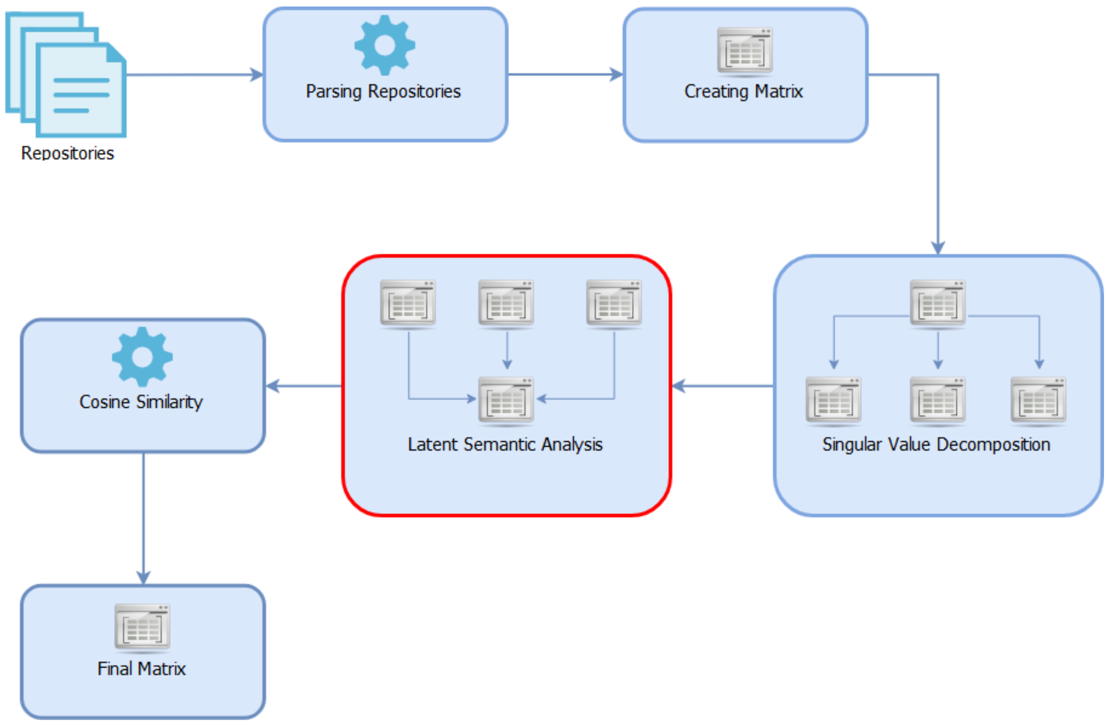


Figure 21: Latent Semantic Analysis

LSA: Unfortunately an implementation of Latent Semantic Analysis wasn't available, so we re-implemented from scratch. Basically we multiplied the 3 matrix provided by the SVD procedure, the important point is the value k for the reduced rank, we selected a value of total $\frac{repository}{2}$. As explained in the Dumais paper, this value should be selected empirically. Here we got a very big issue since the $Memory\,in\,gigabytes = \frac{(columns*rows*8)}{(1024*1024*1024)}$ is required for a matrix, for MudaBlue we got an amount of 700000 distinct terms for a total of 3GB of dedicated memory just for matrix, without cosidering any kind of operation. This is due to the fact that MudaBlue considers many different terms from a file. Clan instead, focusing only on the the import and method that belongs to the JDK, reduced greatly the number of distinct terms. The main solution was to increase the available memory for eclipse up to 8GB. Even though this memory space, we got many crashes, so we spent some time in refactoring the code to save memory, e.g. deleting unused data structure, using more light structures and so on.

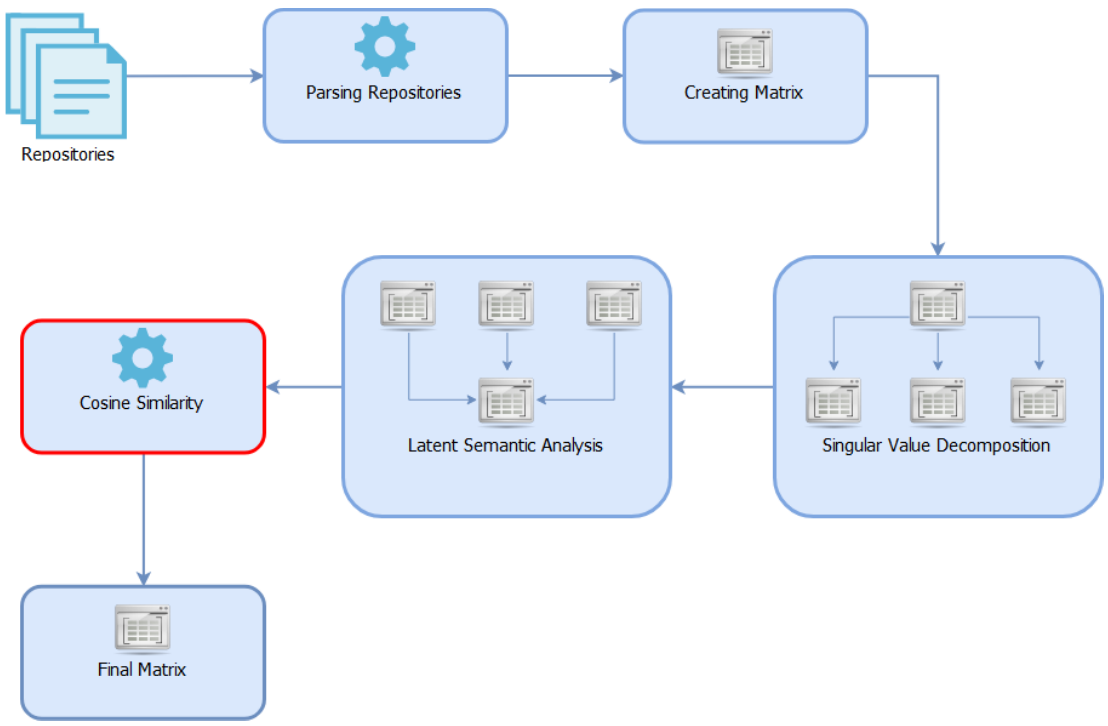


Figure 22: Cosine Smilarity

As stated before, by cosine similarity we mean a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them, that is, how much they are close or far each other. As for the *LSA* we re-implemented the operation from scratch, so the method take as input two vectors and computes the operation, such vectors are taken from the *LSA* matrix, in such a way that every couple is taken into account. Since in the final matrix we will have the simlarity between *repo1 - repo2* and *repo2 - repo1*, we computed the cosine only in the upper triangular matrix to cut half of the calculation.

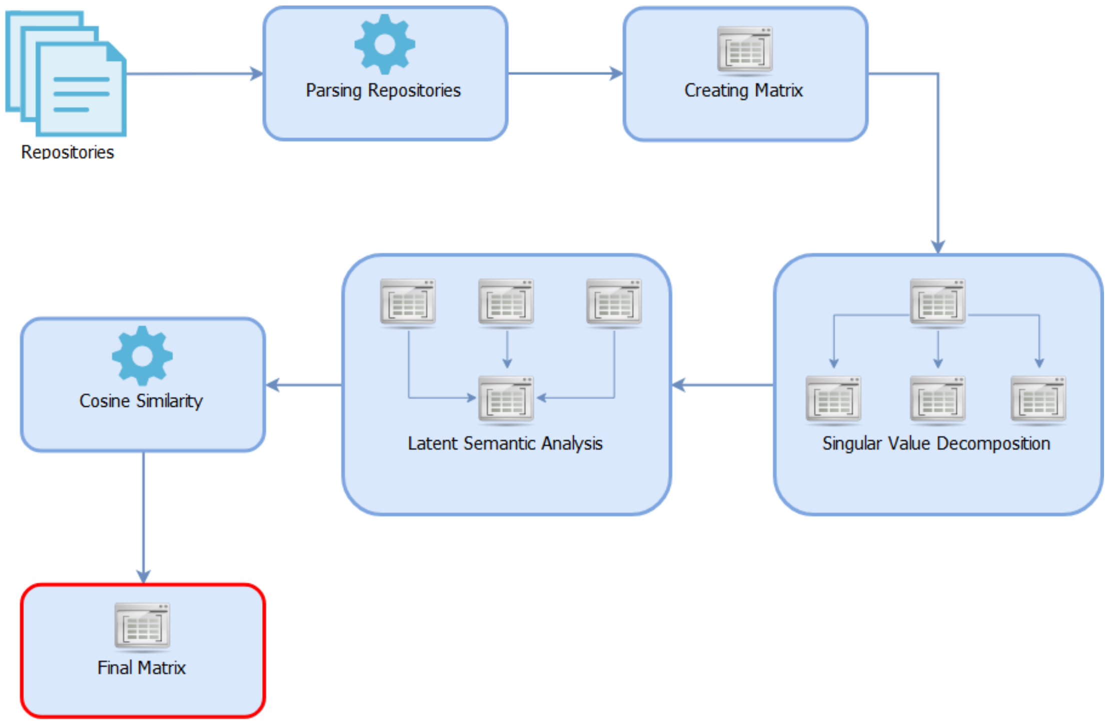


Figure 23: Final Matrix

At this stage the matrix is complete with *580 * 580* in dimension, and with values between *0.0* and *1.0*. This matrix is actually a collection of vectors, representing the similarity of a project with all the other projects. To be more formal the final matrix $\|M\|$ is square matrix whose rows and colunum represents projects. In particular, for any two project P_i and P_j, each element of the matrix $M_{i,j}$ represents the similarity score defined as follows:

$$M_{i,j} = \begin{cases} 0 \leq M \leq 1 & \text{if i} \neq \text{j} \\ \\ 1 & \text{if i = j.} \end{cases} \tag{11}$$

There is one more step for CLAN, since the approach consider the matrices separately, that is, at this stage we have two different matrices, one for the imports and one for the methods. So we have to sum up both in order to get the final one.

4.7 Tools and Libreries

During the experience, the work has been done using Eclipse IDE Oxygen .2, and using the following libreries.

- org.eclipse.jdt.core 3.10.0. This is the core part of Eclipse's Java development tools. It contains the non-UI support for compiling and working with Java code, including the following:
  - an incremental or batch Java compiler that can run standalone or as part of the Eclipse IDE
  - Java source and class file indexer and search infrastructure
  - a Java source code formatter
  - APIs for code assist, access to the AST and structured manipulation of Java source.
- eclipse-astparser 8.1. This is used to analyze the AST at runtime on Eclipse.
- commons-math3 3.6.1 Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang. In particular used to compute the SVD, singular value decomposition.
- commons-text 1.2. Apache Commons Text is a library focused on algorithms working on strings.
- javaparser-core 3.5.14. This is a library for parsing the java files.
- ejml 0.33. Efficient Java Matrix Library (EJML) is a linear algebra library for manipulating real/complex/dense/sparse matrices. Its design goals are; 1) to be as computationally and memory efficient as possible for both small and large matrices, and 2) to be accessible to both novices and experts.These goals are accomplished by dynamically selecting the best algorithms to use at runtime, clean API, and multiple interfaces.

5 Evaluation

5.1 Overview

In this section we discuss the process that has been conceived and applied to evaluate the performance of *CROSSSIM* in comparison some baselines. As stated before we opted for *MUDABLUE* and *CLAN*. The rationale behind the selection of these approaches is that they are well-established algorithms.

5.2 Dataset

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

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

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

Using the GitHub API[2], we crawled projects to provide input for the evaluation. Though the number of projects that fulfill the requirements of a single approach, i.e. either RepoPal or CrossSim, is high, the number of projects that meet the requirements of both approaches is considerably lower. For example, a project contains both ”pom.xml” and ”README.md”, albeit having only 5 dependencies, does not meet the constraints and must be discarded. The crawling is time consuming as for each project, at least 6 queries must be sent to get the relevant data. GitHub already sets a rate limit for an ordinary account[3], with a total number of $5,000$ API calls per hour being allowed.

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

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

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 [38].

Further than collecting projects for each category, we also started collecting random projects. These projects serve as a means to test the stability of the algorithms. If the algorithms work well, they will not perceive newly added random projects as similar to projects of some other specific categories. To this end, the categories and their corresponding cardinality to be studied in our evaluation are listed in Table 8. This is an approximate classification since a project might belong to more than one category.

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

Table 8: List of software categories

As can be seen in Table 8, among 580 considered projects, 309 of them belong to some specific categories, such as *SPARQL, RDF, Jena Apache*, *Selenium Test*, *Elastic Search*, *Spring MVC*, etc. The other 271 projects being selected randomly belong to *Miscellaneous Categories*. These categories disperse in several domains and sometimes it happens that there is only one project in a category.

5.3 User Study

We involved a group of 15 people software developers who have at least 5 years of experience in the user study. To get information about the participants related to their programming experience, we sent them a questionnaire similar to the one presented in [2].According to the survey, we found out that most developers use StackOverflow to search for code.

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

Table 9: Similarity scales

By the user study, in order to have a fair evaluation, for each query we mixed and shuffled the top-5 results generated from the computation by all similarity metrics in a single file and present them to the evaluators. This mimics a *taste test* where users are asked to evaluate a product, (e.g. food or drink, without having a priori knowledge about what is being addressed [**?**], [39]). This aims at eliminating 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. The participants are asked to label the similarity for each pair of projects *(, <query, retrieved project>)*with regards to their application domains and functionalities using the scales listed in Table 9 [2]. For example, an OSS project p_1 that performs the sending of files across a TCP/IP network is somehow similar to an OSS project p_2 that exchanges text messages between two users, *i.e.* $Score(p_1, p_2) = 3$. However an OSS project p_3 with the functionalities of a pure text editor is dissimilar to both p_1 and p_2, *i.e.* $Score(p_1, p_2) = Score(p_1, p_3) = 1$. Given a query, a retrieved project is considered as a *false positive* if its similarity to the query is labeled as *Dissimilar* (1) or *Neutral* (2). In contrast, *true positives* are those retrieved projects that have a score of 3 or 4, *i.e.* Similar *of* Highly similar. A good similarity metric should produce as much true positives as possible.

5.3.1 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 related work [2, 4, 21]:

- *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.

5.4 Evaluation Results

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

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

By this question, we study the performance of different approaches.

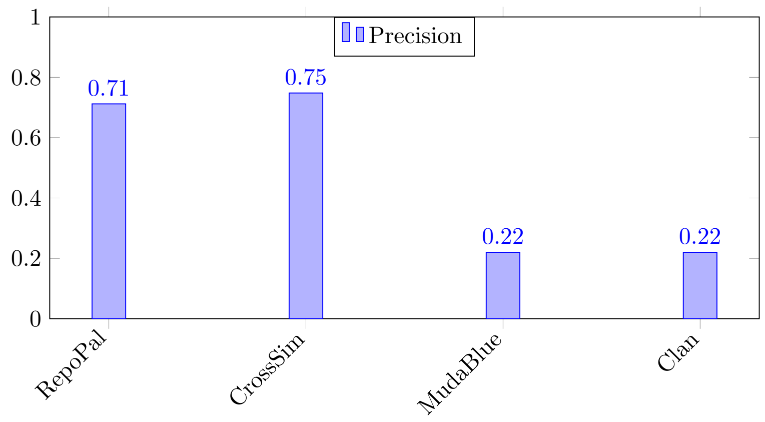


Figure 24: Precision Comparison

Experimental results suggests that CrossSim approach overperforms all the other approaches, in particular Clan and MudaBlue. Repopal got a good score, this means that is still a valid choice for similarity in the OSS environment. The precision,as the figure 24 depicts, shows that CrossSim and Repopal got a score *greater than 70%*. Clan and MudaBlue instead, reported a very low score, *about 20%*, on 10 queries evaluted, just 2 got a score ≥ 3.

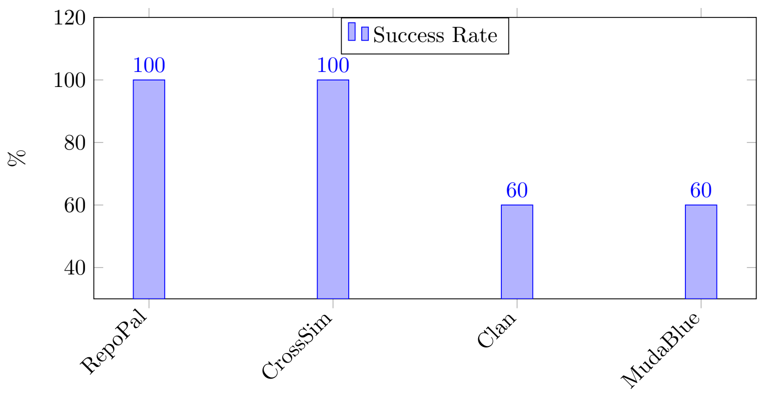


Figure 25: Success Rate Comparison

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

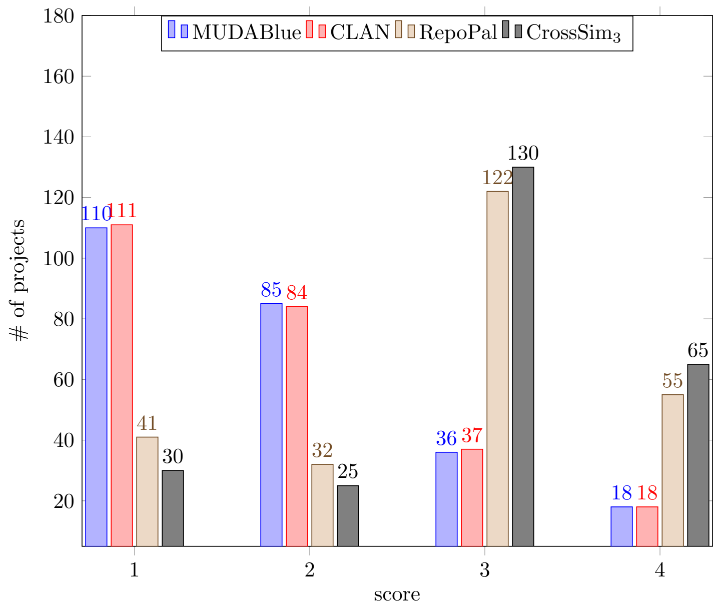


Figure 26: Confidence Comparison

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

RQ_2: Which similarity metric is more efficient?

An important factor for a similarity metric is the ability to compute within an acceptable amount of time.

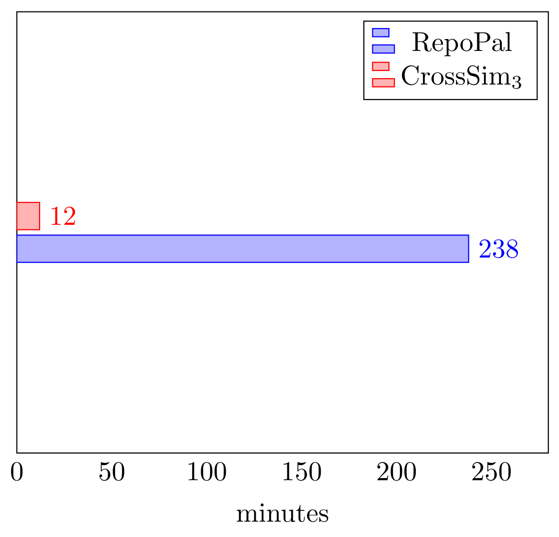


Figure 27: Execution Time Comparison

5.5 Threats to Validity

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

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

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

Reliable validity is related to the reproducibility of our experiments. To allow anyone to seamlessly replicate the evaluation, we made available the source code implementation of MUDABlue, CLAN, RepoPal, and CrossSim as also the dataset exploited in the paper in our GitHub repository [38].

6 Conclusion

The purpose of this work was providing a baseline result for the evaluation of a novel similarity calculator approach, CrossSim. CrossSim is an approach developed by us inside the context of the CrossMiner project.

CROSSMINER[4] is a research project funded by the EU Horizon 2020 Research and Innovation Programme, aiming at supporting the development of complex software systems by *i)* enabling monitoring, in-depth analysis and evidence-based selection of open source components, and *ii)* facilitating knowledge extraction from large OSS repositories [8].

In order to provide such a baseline was mandatory to find some similar approach, we decided to use MudaBlue and Clan which are two close approaches, since there weren't any implementation available we re-implemented them from scratch. The contribute can be summarized as follows:

- Study and Analysis of the problem. In this phase we have analyzed the similarity problem discovering that is well known problem studied in order to find a solution to some very interesting problems such as: (plagiarism detection, information retrieval,text classification, document clustering, topic detection and so on).In order to validate our novel approach we eventually decided to study in detail and implement two similarity calculator approaches: MudaBlue and Clan.
- Implementation. The implementation phase covered a lot of aspects. First of all was necessary analyzing the projects by parsing each *.java* file and then summing up everything in a *Term-Document matrix.* We applied then, the core of the apporaches, the *Latent Semantic Analysis*, applying then the cosine similarity on the matrix we got the final matrix ready to be evaluated. The Ide was Eclipse and the language Java, with a lot of supporting library.
- Results Validation. At this stage we started the evaluation phase which consisted in a user study. We asked to a group of 10 people with experencie in Java delepoment, to rate a pull of queries provided by us. The results confirmed that CrossSim is a more precise method to calculate similarity with rispect to Clan and MudaBlue.

One of the most hard issue faced was related to the physical memory required to compute the Latent Semantic Analysis, for MudaBlue in particular we got something like *700000* terms. This means that the required memory, only to manage the matrix was about 3Gb, this excluding all the memory used for other data structures and for the parsing. That's why we put a bound for the Eclipse virtual memory up to 8Gb and worked in two phase. During the first phase we collected all the terms by parsing everything and then, after an IDE restart, computing the LSA.

Since the evualation was succesfully, in the sense that, results confirmed that CrossSim is a valuable similarity approach, the idea is to continue the development of the other features that still are missing (e.g. Code snippet suggestion, Api reccomandation).

[4] `https://www.crossminer.org`

References

[1] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of MSR 2014*, pages 102–111. ACM, 2014.

[2] Collin McMillan, Mark Grechanik, and Denys 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.

[3] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. The adaptive web. chapter Collaborative Filtering Recommender Systems, pages 291–324. Springer-Verlag, Berlin, Heidelberg, 2007.

[4] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 00:13–23, 2017.

[5] Tommaso Di Noia and Vito Claudio Ostuni. Recommender systems and linked open data. In *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*, pages 88–113, 2015.

[6] Badrul Sarwar, George Karypis, Joseph Konstan, and John 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.

[7] Ning Chen, Steven C.H. Hoi, Shaohua Li, and Xiaokui 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.

[8] Alessandra Bagnato et. al. Developer-centric knowledge mining from large open-source software repositories (crossminer). In *Software Technologies: Applications and Foundations*, pages 375–384. Springer International Publishing, 2018.

[9] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. CrossSim: exploiting mutual relationships to detect similar OSS projects. In *Procs. of 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) - to appear*, 2018.

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

[11] Amos Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.

[12] Lan Huang, David Milne, Eibe Frank, and Ian H. Witten. Learning a concept-based document similarity measure. *J. Am. Soc. Inf. Sci. Technol.*, 63(8):1593–1608, August 2012.

[13] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Discov. Data*, 2(2):10:1–10:25, July 2008.

[14] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[15] Ainura Madylova and Sule Gündüz Ögüducü. A taxonomy based semantic similarity of documents using the cosine measure. In *ISCIS*, pages 129–134. IEEE, 2009.

[16] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 775–780. AAAI Press, 2006.

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

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

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

[20] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 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, pages 71–82, New York, NY, USA, 2015. ACM.

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

[22] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.

[23] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Int. Res.*, 37(1):141–188, January 2010.

[24] Joel W. Reed, Yu Jiao, Thomas E. Potok, Brian A. Klump, Mark T. Elmore, and Ali R. Hurson. Tf-icf: A new term weighting scheme for clustering dynamic data streams. In *Proceedings of the 5th International Conference on Machine Learning and Applications*, ICMLA '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Juan Ramos. Using tf-idf to determine word relevance in document queries, 1999.

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

[27] Kirk Baker. Singular value decomposition tutorial. 2005.

[28] Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 448–453, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[29] Jay J. Jiang and David W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. In *International Conference Research on Computational Linguistics (ROCLING X), 1997, Taiwan*, 1997.

[30] Chao Liu, Chen Chen, Jiawei Han, and Philip 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.

[31] Vincent D. Blondel, Anahí Gajardo, Maureen Heymans, Pierre Senellart, and Paul Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Rev.*, 46(4):647–666, April 2004.

[32] Wangzhong Lu, J. Janssen, E. Milios, N. Japkowicz, and Yongzheng Zhang. Node similarity in the citation graph. *Knowledge and Information Systems*, 11(1):105–129, Jan 2007.

[33] Satu Elisa Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, August 2007.

[34] Phuong T. Nguyen, Paolo Tomeo, Tommaso Di Noia, and Eugenio Di Sciascio. Content-based recommendations via dbpedia and freebase: A case study in the music domain. In *Proceedings of the 14th International Conference on The Semantic Web - ISWC 2015 - Volume 9366*, pages 605–621, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[35] Phuong T. Nguyen, Paolo Tomeo, Tommaso Di Noia, and Eugenio 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.

[36] Vito Claudio Ostuni, Tommaso Di Noia, Roberto Mirizzi, and Eugenio Di Sciascio. A linked data recommender system using a neighborhood-based graph kernel. In *The 15th International Conference on Electronic Commerce and Web Technologies*, Lecture Notes in Business Information Processing. Springer, 2014.

[37] Michael J. Pazzani and Daniel Billsus. *Content-Based Recommendation Systems*, pages 325–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[38] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. CrossSim tool and evaluation data, 2018. `https://doi.org/10.5281/zenodo.1252866`.

[39] Simone Pettigrew and Stephen Charters. Tasting as a projective technique. *Qualitative Market Research: An International Journal*, 11(3):331–343, 2008.