Thesis Draft

Rubei Riccardo

May 8, 2018

Contents

1 Introduction

1.1 Work Description

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.

1.2 CROSSMINER

1.2.1 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.

1.2.2 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.

1.2.3 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.

2 The Similarity Problem

2.1 Overview

In the field of Information Retrieval [NOTE], there are techniques being widely used in several applications. They can be considered as a basic and indispensable part of a knowledge mining system. [AGGIUNGERE QUALCOSA]

2.2 Mathematical Background

2.2.1 Term-Document Matrix

In Natural Language Processing [?], a term-document matrix (TDM) is used to represent the relationships between words and documents [?]. 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 [?]. 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 [?]:

$$w_k^d = tf \cdot idf(k, d, D) = f_k^d \cdot log \frac{n}{|\{d \in D : t_k \in d\}|} \tag{1}$$

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

Another common weighting scheme uses only the frequency of terms in documents for cells in TDM, i.e. the number of occurrence of a term in a document, instead of *tf-idf*. As an example, we consider a set of 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.*

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

Figure 1: An example of a term-document matrix

The set of terms t consists of 6 elements, i.e. $t = (she, is, today, a, nice, city)$ and the corresponding term-document matrix for D is depicted in Figure 1.

TDM has been exploited to characterize software systems and finally to compute similarities between them [?], [?], [?]. 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 1 where documents are replaced by software systems and terms are replaced by API calls.

2.2.2 Cosine Similarity

Cosine similarity is a metric used to compute similarity between two objects using their feature vectors [?]. 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.

$$CosineSim(\vec{\alpha}, \vec{\beta}) = \frac{\sum_{i=1}^{n} \alpha_i \cdot \beta_i}{\sqrt{\sum_{i=1}^{n} (\alpha_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (\beta_i)^2}} \tag{3}$$

Figure 2 illustrates the cosine similarity between two vectors $\vec{\alpha}$ and $\vec{\beta}$ in a three-dimension space. This can be thought as the similarity between two documents with three terms $t = (t_1, t_2, t_3)$.

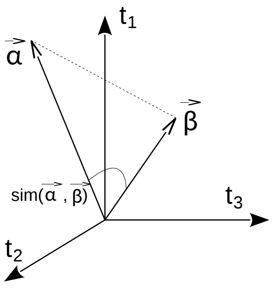


Figure 2: Cosine similarity between two feature vectors $\vec{\alpha}$ and $\vec{\beta}$

Cosine similarity has been popularly adopted in many applications that are related to similarity measurement in various domains [?], [?], [?], [?], [?]. 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* [?], *CLAN* [?], *CLANdroid* [?], *LibRec* [?], *SimApp* [?], *WuKong* [?], *TagSim* [?], and *RepoPal* [?]

2.2.3 Latent Semantic Analysis

The problem with the term-document matrix is that the intrinsic relationships among different terms of a document cannot fully be captured. Furthermore, same words can be used to explain different requirements or the other way around, the same requirements can be described using different words [?]. Latent Semantic Analysis (LSA), also known as Latent Semantic Indexing (LSI), has been proposed to overcome these problems [?]. 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 [?]. 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 [?]:

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

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 [?], [?], [?] 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. Another very relevant issue related to LSA is the low rank approximation applied by the SVD procedure. 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 we decided a k value = numer of reposotories/2 [TO BE COMPLETED]

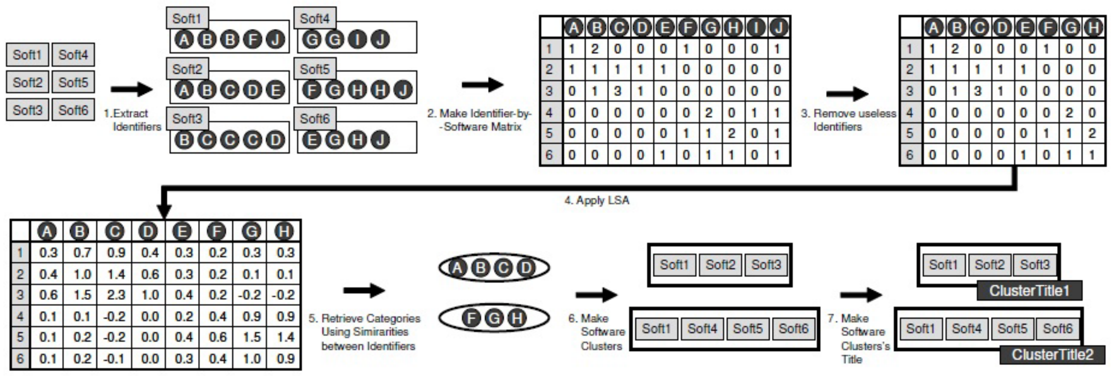
3 The Approaches

3.1 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 or a large collecton of software systems. MUDABlue method does not only categorize sooware systemsd but also determines categories rom the sooware systems collecton automatcally. 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.

3.2 The Approach

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



Soft1 Soft4
Soft2 Soft5
Soft3 Soft6
1.Extract Identifiers
Soft1 A B B F J
Soft2 A B C D E
Soft3 B C C C D
Soft4 G G I J
Soft5 F G H H J
Soft6 E G H J
2. Make Identifier-by--Software Matrix
3. Remove useless Identifiers
4. Apply LSA
5. Retrieve Categories Using Simirarities between Identifiers
A B C D
F G H
6. Make Software Clusters
Soft1 Soft2 Soft3
Soft1 Soft4 Soft5 Soft6
7. Make Software Clusters's Title
Soft1 Soft2 Soft3 ClusterTitle1
Soft1 Soft4 Soft5 Soft6 ClusterTitle2

	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 3: MUDABlue phases.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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].

3.2.6 Categorization

The point 6 and 7 are not covered because not related to our work.

3.3 CLAN: Closely reLated ApplicatioNs

CLAN [**?**] 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.

3.4 The Approach

The process consist of 12 steps here graphically reported.

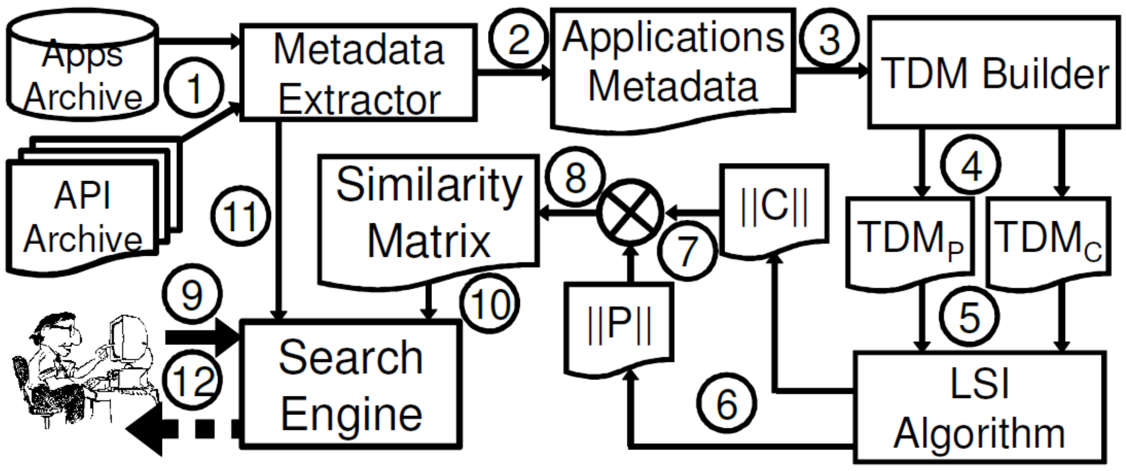


Figure 4: CLAN phases.

3.4.1 1 - 3: 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.

3.4.2 4: 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.

3.4.3 5: LSI Procedure

3.4.4 6: Apply the Cosine Similarity

3.4.5 7: 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.

3.4.6 8: Final similarity matrix

3.4.7 RepoPal: Exploiting Metadata to Detect Similar GitHub Repositories

In contrast to many previous studies that are generally based on source code [?], [?], [?], *RepoPal* [?] 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 $\vec{f_j}$:

$$sim_f(r_i, r_j) = CosineSim(\vec{f_i}, \vec{f_j}) \tag{5}$$