CROSSMINER
Developer-Centric Knowledge Mining from Large Open-Source Software Repositories

Project Number 732223

D6.4 A tool for the unsupervised classification of OSS projects - Final Version

Version 1.0
24 December 2018
Final

Public Distribution

University of L′Aquila

Project Partners: **Athens University of Economics & Business**, **Bitergia**, **Castalia Solutions**, **Centrum Wiskunde & Informatica**, **Eclipse Foundation Europe**, **Edge Hill University**, **FrontEndART**, **OW2**, **SOFTEAM**, **The Open Group**, **University of L′Aquila**, **University of York**, **Unparallel Innovation**

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L'Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	Document outline	18 July 2018
0.2	First draft	13 September 2018
0.8	Second draft	7 November 2018
0.9	First release	5 December 2018
1.0	Final release	24 December 2018

Table of Contents

Executive Summary

Open source software (OSS) allows developers to study, change, and improve the code free of charge. There are several high quality software projects which deliver stable and well-documented products. Most OSS forges typically sustain vibrant expert and user communities which in turn provide decent levels of support both with respect to answering user questions as well as to repairing reported software bugs. Developing a new software system by making use of existing open source components reduces development effort and thus being beneficial to the whole software life cycle. However, to improve software quality, it is essential to develop using only well-defined, mature projects. In this sense, the ability to group projects into clusters so that projects in a cluster are more similar to each others than to any project in other clusters, is of highly importance. This functionality helps developers efficiently locate the most suitable projects with respect to certain criteria.

In this deliverable, we present the final version of the unsupervised clustering tool which equips developers with an important type of recommendations: Clustering OSS projects into independent groups. By exploiting CROSSSIM developed from the previous phases of Work Package 6, we are able to compute similarities among projects and provide as input for the clustering process. Afterwards, we exploit two well-known algorithms, i.e., K-Medoids and CLARA to produce final clusters. To further support developers, apart from the unsupervised tool, we also design and implement a supervised neural network named OSCAN to cluster OSS projects by learning from predefined labeled data. It is our firm belief that OSCAN would come in handy given that there is decent data available for the training process.

1 Introduction

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, are beneficial to the development process once they are 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 [63]. Among others, we work towards a tool that groups OSS projects into independent clusters. It is the process of assigning highly relevant projects, in terms of their functionalities, into a single cluster [15]. This is a practical gadget given that developers want to approach the most relevant projects, considering a project being developed, by effectively narrowing down the search scope. Furthermore, clustering software projects also helps stakeholders identify related requirements as well as forecast maintenance issues [40],[47].

In Section 1.1, we summarize the main types of recommendations which have been defined in the Description of Work (DoW) for Work Package 6. This section also recalls the results obtained in previous phases of our work package. Afterwards, in Section 1.2, we introduce the main tasks related to clustering OSS projects. Finally, Section 1.3 brings in the structure of the whole deliverable.

1.1 Recommendations and Similarity Computation

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.

In the context of the CROSSMINER project[1], we aim at supporting software developers by means of an advanced Eclipse-based IDE providing intelligent recommendations that go far beyond the current *code completion-oriented* practice. To this end, metadata is curated from different OSS forges and processed to properly feed the recommendation engines. By means of the augmented IDE, developers are able to select open source software and get real-time recommendations based on the CROSSMINER mining tools and on previous feedback. In the first place, the following main types of recommendations are envisaged to meet the requirements of several industrial partners [57]:

- Given a new software system being developed, *find a set of similar OSS projects* with respect to different criteria, such as external dependencies, application domain, or API usage. This type of recommendation is beneficial to the development since it helps developer learn how similar projects are implemented;
- *Recommend components* that similar projects have included, for instance, a list of external libraries [57],[77] code snippets [48],[56]. These artifacts can be directly embedded into the current project, thus sparing the development time;
- *Recommend code snippets* that show how an API is used in practice. These snippets supply to developer a deeper insight into the usage of the API;

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

- *Suggest additional sources of information*, e.g., technical documents, tutorials, communication channels, etc., that are relevant to the code being developed. For example, given an API, the task is to retrieve external posts to understand how other developers use the API [63];
- *Identify API changes and their consequences*. External libraries evolve over the course of time, any changes will have a certain effect on the depending projects. Thus, it is necessary to notify developers and recommend possible amendments to preserve program compatibility by means of notifications or communication channel items discussing about the corresponding API changes.

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 [55],[57]. This type of recommendation is beneficial to the development since it helps developer learn how similar projects are implemented. Moreover, to aim for software quality, developers normally build their project by learning from mature OSS projects having comparable functionalities [74]. Thus, it is necessary to equip software developers with suitable machinery which facilitates the similarity search process.

In Deliverable D6.1, an initial version of the CROSSMINER Knowledge Base has been introduced. In Deliverable D6.2, we developed CROSSSIM [55], a versatile tool for computing similarities among OSS projects. 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. In Deliverable D6.3, we built a recommender system named CROSSREC for providing software developers with third-party libraries [57] by exploiting CROSSSIM to compute similarities among projects. In this setting, CROSSSIM performs its computation using third-party libraries as the input features. The experimental results demonstrated that CROSSREC outperforms LibRec [77], a well-established baseline in terms of various quality metrics. To provide API function calls and usage patterns recommendation, we developed a context-aware recommender system, i.e., FOCUS [56]. In this system, CROSSSIM has been applied to compute similarities using API function calls as features. The performance of FOCUS is superior to that of PAM [17],[18]. The results obtained from the FOCUS approach were written in a paper which has then been accepted at the Technical Track of ICSE 2019[2] [60]. In this way, we demonstrated that CROSSSIM can effectively and efficiently compute similarities among projects by using different features as inputs.

1.2 Clustering OSS Projects

As planned in the CROSSMINER Description of Work (DoW), this deliverable provides the final version of the unsupervised clustering tool which is related to the following task:

> *Task 6.2: Automated classification of OSS projects:* This task will focus on multi-dimensional classification of OSS projects by exploiting information available in the knowledge base designed in Task 6.1. In particular, for satisfying developer requirements projects will be organized by means of clustering techniques that will automatically classify projects with respect to defined relationships e.g., dependency, conflict, substitutability, quality, license compatibility, and frequency usage. Similarity measures will be defined accordingly and they will be exploited during the clustering process to assess if two given projects are similar (dissimilar) and thus (do not) belong to the same cluster.

[2] https://2019.icse-conferences.org/track/icse-2019-Technical-Papers#event-overview

Clustering is deemed to be among the fundamental techniques in Knowledge Mining and Information Retrieval [45],[75]. Clustering techniques have been widely used in other fields including Biology to classify plants and animals according to their properties, and Geology to classify observed earthquake epicenters and thus to identify dangerous zones. In Software Engineering, clustering has been used in reverse engineering and software maintenance for categorizing software artifacts [43],[50]. A clustering algorithm attempts to distribute objects into groups of similar objects so as the similarity between one pair of objects in a cluster is higher than the similarity between one of the objects to any objects in a different cluster [6],[27]. In recent years, several clustering methods have been developed to solve a wide range of issues [84]. Among others there are hierarchical and partitional clustering algorithms [27]. The former use a criterion function to identify partitions while the latter try to group similar partitions. By partitioning-based algorithms, there are K-Means, K-Medoids, CLARA, CLARANS [33],[53],[44]. By hierarchical-based algorithms, there are BIRCH [86], CURE [22], ROCK [1], Chameleon [30], to name a few.

Several existing clustering techniques share the property that they can be applied when it is possible to specify a *proximity (or distance) measure* that allows one to assess if elements to be clustered are mutually similar or dissimilar. The basic idea is that the *similarity level of two elements is inversely proportional to their distance*. The definition of the proximity measure is a key issue in almost all clustering techniques and it depends on many factors including the considered application domain, available data, and goals. Once the proximity measure has been defined, it is possible to produce a proximity matrix for the related objects. Given that there are n objects to be clustered, an $n \times n$ proximity matrix needs to be generated containing all the pairwise similarities or dissimilarities between the considered objects.

In this deliverable, we make use of the infrastructure developed from previous tasks to produce such proximity matrix and eventually design and implement two independent tools, namely unsupervised and supervised clustering. For the unsupervised tool, CROSSSIM is applied to generate a similarity matrix for a set of projects and perform clustering using K-Medoids and CLARA. The rationale behind the selection of these two algorithms is that they perform in a reasonable amount of time while still being able to produce highly relevant clusters. An evaluation on a dataset of 570 projects has been conducted to study our approach's performance. Experimental results show that the unsupervised clustering tool can generate relevant clusters with respect to ground-truth classes. Meanwhile, by the supervised tool we deploy a Fuzzy ARTMAP neural network [8],[87] to learn from labeled data and eventually to cluster testing OSS projects. Also for this tool, we performed an evaluation on a dataset of 745 SourceForge projects. We demonstrate that the supervised tool is able to effectively cluster testing data.

1.3 Document Structure

The deliverable is organized in the following sections:

- Section 2 presents a literature review on the related work;
- Section 3 introduces some metrics for evaluating the outcomes of a clustering process;
- The unsupervised clustering tool is presented in Section 4;
- Section 5 presents OSCAN, a supervised Fuzzy ARTMAP neural network for clustering OSS projects with respect to different sets of features;
- Section 6 provides a summary on the REST API for the Knowledge Base;
- Finally, Section 7 summarizes the existing work and concludes the deliverable.

2 Background

In this chapter, we provide a review on some of the most notable studies that are related to our study, i.e., software similarity computation, and techniques for clustering software projects. Since many clustering algorithms base their computation on a similarity matrix, in the first place it is necessary to compute similarities among OSS projects. Thus, Section 2.1 recalls studies that address the issue of detecting similar software projects. The section is actually a recapitulation of the work done in Deliverable D6.2 concerning OSS project similarities. From Section 2.2 to Section 2.9, we review different techniques for clustering software and OSS projects.

2.1 Software Similarities

The concept of similarity is a key issue in various contexts, such as detecting cloned code [16],[67],[66],[78] software plagiarism [38], or reducing test suite in model-based testing [12],[34]. Nevertheless, a globally exact definition of similarity is hard to come by since depending on the method used to compare items, various types of similarity may be identified. According to *Walenstein et al.* [81], a workable common understanding for software similarity is as follows: *"the degree to which two distinct programs are similar is related to how precisely they are alike."*

In the context of open source software, two projects are deemed to be similar if they implement some features being described by the same abstraction, even though they may contain various functionalities for different domains [46]. Understanding the similarities between open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations [72],[88], thereby improving software quality [74]. Meanwhile, measuring the similarities between developers and software projects is a critical phase for most types of recommender systems [61],[71]. 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 user [72]. In this sense, 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 [10],[46]. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail.

Having access to similar software projects helps developers speed up their development process. By looking at similar OSS projects, developers are able to learn how relevant classes are implemented, and in some certain extent, to reuse useful source code [72],[88]. Also, recommender systems rely heavily on similarity metrics to suggest suitable and meaningful items for a given item [14],[58],[59],[72],[77]. For example, by third-party library recommendation, it is important to find similar projects to a given project, and mine libraries from the most similar projects [57]. As a result, similarity computation among software and projects has attracted considerable interest from many research groups. In recent years, several approaches have been proposed to solve the problem of software similarity computation. Many of them deal with similarity for software systems, others are designed for computing similarities among open source software projects. According to [10], depending on the set of mined features, there are two main types of software similarity computation techniques:

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

	MUDABlue [20]	**CLAN** [46]	**CLANdroid** [39]	**GPLAG**[41]	**LibRec** [77]	**SimApp** [10]	**AnDarwin** [13]	**WuKong** [82]	**TagSim** [42]	**RepoPal** [88]	
Considered features											**Descriptions**
Lines of code				×							The number of lines of code of all source files in a project
Dependencies					×			×			Set of third-party libraries a project includes
API Calls	×	×	×				×	×			API function calls that appear in the source code of the analysed projects. They are used to build term-document matrices and then to calculate similarities among applications
Functions	×			×			×				Functions defined in a project's source code
Stars										×	The GitHub star events occurred for each analysed projects
Timestamps										×	The point of time when a user stars a repository
Statements	×			×							Source code statements
Identifiers	×		×					×			All artifacts related to source code, such as variable names, function names, package names, etc.
App name						×					Name of a mobile app may reveal its functionalities
Descriptions						×					The description text of an app
Developers						×					All developers who contribute to the development of a software/an app
Readme							×			×	Descriptions or `README.MD` files, used to describe the functionalities of an open source project
Tags									×		The tags that are used by OSS platforms, e.g., SourceForge to classify and characterize an OSS project
Updates						×					The newest changes made to the considered applications
Permissons			×			×					This feature is available by mobile apps. It specifies the permission of an app to handle data in a smartphone
Screenshots						×					This feature is available by mobile apps. It is a picture representing an app
Contents						×					Each app has content rating to describe its content and age appropriateness
Size						×					Some similarity metrics assume that two apps whose size is considerably different cannot be similar
Reviews						×					All user reviews for an app are combined in a document
Intents			×								For a mobile app, an intent is description for an operation to be performed
Sensors			×								In mobile devices, sensors can provide raw data to monitor 3-D device movement or positioning, or changes in the environment. A set of features can be built from sensors and used to characterize an app
Category											
High (H) / Low (L) level similarity	L	L	L	L	L	H	H	L	H	H	*Low-level similarity* metrics consider only low-level data, e.g., function calls, API reference, meanwhile *high-level similarity* metrics exploit metadata e.g., readme files, texts, star events, etc. to compute similarities

Table 1: Summary of software and OSS project similarity algorithms.

Table 1 gives a summary of various techniques for computing software similarities. The approaches are either low-level or high-level similarity. It is evident that each of these similarity tools is able to manage a certain set of features. Thus, they can only be applied in *prescribed contexts*, and cannot exploit additional information when this is available for similarity computation.

To overcome these limitations, in Deliverable D6.2 we proposed CROSSSIM – a novel approach that attempts to effectively exploit the rich metadata infrastructure provided by the OSS ecosystem to compute software similarities [55]. We assume that combining multi types of information as input in computing similarities can be highly beneficial to the context of OSS repositories. In other words, the ability to compute software similarity in a *flexible* manner is of highly importance. Flexibility in similarity computation is understood as the way a tool computes similarities using different types of input features, without being limited to just source code as by CLAN [46], MUDABlue [20], or to just metadata as by REPOPAL [88], TagSim [42], or SimApp [10]. For instance, in the context of the CROSSMINER project, the required project similarity technique should be flexible enough to enable the development of different types of recommendations, e.g., third-party library [57], API function calls [51],[60], StackOverflow posts [63]. To this end, CROSSSIM is capable of incorporating new features into the similarity computation without the need to modify its internal design. By deploying CROSSSIM to compute project similarity in various settings [55],[56],[57] we demonstrated that it can effectively and efficiently compute similarities among projects in a flexible manner. In this sense, CROSSSIM becomes a *hybrid similarity tool* as it allows for the incorporation of various inputs, both low-level and high-level features. In this deliverable, we exploit CROSSSIM to compute similarities among projects to provide input for the unsupervised clustering tool.

In recent years, the problem of software categorization or clustering has attracted considerable interest from the research community. In the next subsections, we review techniques and tools that have been developed to cluster software and OSS projects.

2.2 Automatic Categorization with Deep Neural Network

Nguyen et al. [54] propose an approach to find similar applications in the same category and to detect high-level features based on a deep neural network (DNN). The proposed DNN consists of four layers: the input layer represents the projects in form of low-level code token. Two hidden layers are used to derive the high-level concepts for each project. Finally, the output layer represents the categories. The code tokens extracted for the input layer belong to the following three categories: identifiers, APIs methods and classes and strings in comments. From the low-level code tokens, the two hidden layers extract features and concepts of the project and then provide the output results in form of an N-dimension vector. Given an input project, the systems produces an N-dimension vector, where each entry corresponds to the likelihood of the project being classified to a given category. For the evaluation, the authors compare the results of the proposed approach with some standard machine learning approaches, i.e., naive Bayes and classical neural networks. Ten-fold cross-validation has been applied to perform the comparison using a dataset of $1,000$ popular Java projects collected from the SourceForge platform [3]. A ground-truth dataset was generated by manually labeling projects to validate the approach's outcome. The experimental results show that the DNN approach obtains a higher accuracy compared with the other approaches as it retrieves almost the same categories with respect to the labels given by humans.

[3] `https://sourceforge.net/`

2.3 SCC: Automatic Classification of Code Snippets

To recognize different programming languages from StackOverflow code snippets, Alreshedy et al. [5] develop Source Code Classification (SCC). SCC is built using the Scikit-learn library in Python based on the supervised machine learning technique Multinomial Naive Bayes (MNB) [19],[35]. MNB is a supervised machine learning algorithm and works based on Bayes theorem. It is used in text classification and Natural Languages Processing (NLP). First, a preprocessing phase is performed on the original code snippets to convert them into a numerical feature vector using the bag-of-word model. MNB recognizes the programming language from a snippet of code by considering the vector feature and the likelihood with the programming language itself. In particular, it calculates the probability that a specific code snippet belongs to a programming language. The Grid-SearchCV library was used to hypertune the machine learning models. An evaluation SCC has been conducted to compare SCC against PLI (Programming languages identification) which performs recognition on program languages. The proposed approach obtains better outcomes compared to PLI with respect to *Precision*, *Recall*, and *F1-Measure*. There are two main limitations of SCC. First, it has been validated using only a StackOverflow dump and may not generalize to other contexts. Second, in the evaluation, only one baseline, i.e., PLI is used.

2.4 BUCS: Unsupervised Software Categorization using Bytecode

BUCS (Bytecode-based Unsupervised Categorization Software) is an approach that extracts semantic information recovered from bytecode and executes an unsupervised algorithm to assign categories to software systems [15]. Among other findings, the authors show that bytecode is sufficient for replacing source code in clustering projects.

The approach consists of 3 main processing steps, namely *Data Extraction*, *Model Construction*, and *Categorization*. The steps are explained as follows.

- Data Extraction: The system extracts the following types of data: *(i)* code identifiers which come from bytecode; *(ii)* Software profiles extracted from textual descriptions of software libraries, if available.
- Model Costruction: BUCS clusters similar bytecode documents and and assigns to a specific category. Bytecode documents are represented using the Vector Space Model. However, terms with the lowest TF-IDF values are removed from the vectors. Afterwards, BUCS clusters the bytecode documents using the Dirichlet Process Clustering technique (DPC) [76]. At the end of the clustering process, DPC produces a vector of terms for each cluster, which represents all the bytecode documents.
- Categorization: Given a new library to be categorized, BUCS extracts identifiers from its bytecode and builds a corresponding document to represent the library. Similar projects are assigned into groups by means of an unsupervised clustering algorithm. BUCS represents the document in a vector using TF-IDF and computes the similarity between this vector and each vector of the existing clusters to determine which cluster the new library should belong to. Once the cluster has been identified, the library will be labeled the same categories with those previously assigned to the cluster stored in repositories.

A new library can be assigned to one of the clusters by computing the similarity. BUCS has been evaluated using a set of libraries crawled from the Apache Software Foundation (ASF) repository. The experimental results showed that BUCS obtains an average match rate of 86%.

2.5 Clustering Mobile Apps Based on Mined Textual Features

A solution to discover latent clusters of mobile applications in app stores is introduced in [4]. In particular, features are extracted using an information retrieval technique proposed by Harman et al. [23], and then apps are clustered by using an agglomerative hierarchical clustering technique. There are three main steps as follows:

- Features Extraction: Features are extracted from apps' description. Afterwards, non-English and stop words are removed and the remaining words are converted to their lemma form, in order to refine the features. NLTK's N-gram CollocationFinder is used to extract so-called *featurelets* which are lists of bi- or tri-grams of commonly collocating words. A greedy hierarchical clustering algorithm is used to aggregate similar features. The result of this process is a collection of featurelets where each represents a certain feature.
- Feature Clustering: Features are cluster to reduce the granularity in app's description.
- App Clustering: Eventually, the prototype-based spherical k-means technique (skmeans) is applied on the resulting FTM to cluster the apps. Skmeans is built upon the vector space model and it measures the similarities among vectors using cosine similarity.

A two-step clustering algorithm was proposed to reduce the granularity of the extracted features. Afterwards, the feature clusters are used to describe the relationships between apps which are modeled using an App-Feature Matrix (AFM). In this matrix, each row corresponds to an app and each column is a Feature Clusters (FC). A Feature-Term Matrix (FTM) is used to capture the relationship between each feature and the linguistic terms it contains. The similarity between word in the featurelet and term is computed by means of WordNet [49], as it provides an adequate way of quantifying the similarity among general English terms. Finally an agglomerative hierarchical clustering technique is adopted to cluster the apps, due to its efficiency when studying the effects of selecting different granularity levels. An evaluation on a dataset containing $17,877$ apps mined from the BlackBerry and Google app stores shows that the approach improves the overall performance.

2.6 Cataloging GitHub Repositories with LDA-GA Algorithm

A cataloging system for GitHub projects is proposed in [73]. Input data for the system is text segments collected from `README.MD` files of GitHub repositories. The segments are pre-processed and fed as input to LDA-GA [62] to identify categories. A heuristic algorithm is applied to automatically extract descriptive text segments related to project functionality from `README.MD` files. In particular, a combination of text preprocessing steps, i.e., tokenization, stop word removal, stemming, and filtering are applied on extracted text segments. Next, the text segments are fed into a LDA-GA topic modeling algorithm. The output of this stage is a set of topics, i.e., categories and projects that belong to them.

The approach has been empirically evaluated using $10,000$ fairly popular GitHub projects. The experiments aim to evaluate the following aspects:

- How the description text extraction method is accurate;
- How the proposed approach provides new categories that complement existing GitHub ones; and
- How the proposed approach identify additional projects to existing categories.

The experimental results show that the approach is able to retrieve functionality descriptive texts with a good accuracy: the obtained *F-Measure* score is 0.7. The approach can identify new categories that are not captured by GitHub showcases, and as well as discover new projects for existing GitHub categories.

2.7 Automated Tagging of Software Projects

In [80], the authors introduce Sally – a technique for automatic tagging of Maven-based software projects. The tool extracts identifiers and variables directly from bytecode. A filter is applied on the identifiers to produce primary and secondary tags for the analyzed projects. Primary tags are related to the project itself without considering dependencies, while secondary ones consider also dependencies. Moreover, Sally is able to build a dependency graph of the whole project by using the DepFind technique [4].

As the final output, Sally produces tag cloud to represent all the tags related to the project. This represents a visual hint for the user, as in the cloud the tags are classified by dimensions: the size is proportional related to the relevance for the project. The approach has been evaluated with the two most online tools for browsing dependencies in Java projects: SourceForge and MVNRepository. The experimental results demonstrate that Sally outperforms two existing approaches as it does not require additional information as categories or `pom.xml` files. The second experiment exploits MUDABLUE [20] as baseline. A user study was conducted to evaluate the performance of both approaches. Also in this evaluation, Sally obtains a better performance compared to that of MUDABLUE with respect to the expressiveness and completeness of the categorization task.

2.8 Software Categorization by Adapting Neural Text Classification

An issue with source code representation is that low-level details from a project's code normally do not match with its high-level features, due to the different programming styles of developers. To tackle this, Le Clair et al. [37] develop a neural text classification technique. The approach uses a word embedding technique to assign a single category to a project. Each method declaration is represented as a vector of integers. Each project is encoded in a string that summarizes the project's name, function name, and content. Then, the string is transformed into a vector of integers. A word is represented as a vector of integers and it is used to feed the neural network and to produce categories. There are four main steps: *(i)* Model each function by means of a vector containing integer numbers and assign a label; *(ii)* Train a word embedding for code; *(iii)* Train a neural network using the function representations and labels from the first step as input; and *(iv)* Given a project to be categorized, every function is classified and a voting mechanism is applied to predict a label for the project.

After the preprocessing phase, a neural network is used to classify the input data. There are three layers. The first one, called *Embedding layer*, takes as input a word previously described and produces a matrix, composed by the sequence length times embedding dimensions. Then the *Convolution layer* takes as input from the previous layer and assigns a category to a function by analyzing the sequence of tokens. A long short term memory (LSTM) is used to capture the semantics between the sequence of tokens. Then the model uses a *Hidden layer* for learning the LSTM units and understand at which category they belong. Finally, the output dense layer produces a real value vector with the category.

2.9 A Machine Learning Approach for Classifying Software Applications

Based on the observation that developers normally build their applications by exploiting third-party libraries and packages, API calls are considered to be good features for characterizing a software, Linares-Vásquez et al. [40],[47] propose a system for automatic categorizing software applications. of software applications using various features. A category is defined as a group of relevant applications with respect to their functionality, e.g., games or email. The system extracts API calls from third-party libraries and packages. Afterwards,

[4] `http://depfind.sourceforge.net/`

different Machine Learning algorithms were used to cluster data, namely Support Vector Machines [24], Naïve Bayes [85], Decision Tree [65], RIPPER [11], and IBK [3]. Two datasets collected from SourceForge and Sharejar were used in the evaluation. The former contains $3,286$ projects and the latter contains 745 projects. Experimental results show that among others, SVM is the most effective algorithm. Furthermore, it has been shown that using packages as feature helps bring a better clustering performance, compared to other set of features, such as method names, class name, and terms.

Table 2 gives a summary on the clustering algorithms presented in this section. Most of the approaches exploit source code artifacts, such as API function calls, token, package names, etc. as input features. Furthermore, various clustering algorithms have been exploited to classify input data. Among others, machine learning algorithms which have been used in other domains also start becoming popular in this domain. Furthermore, neural networks appear to be suitable for clustering projects as demonstrated in [37] and [54]. This is an inspiration to us and we are going to implement a neural network to cluster OSS projects in Section 5.

In the next section, we introduce our propose approach by exploiting CROSSSIM as the similarity measure. This allows for the incorporation of various features, both low-level and high-level information, into the similarity computation. A unsupervised learning clustering approach is proposed by adopting K-Medoids and CLARA as the clustering engine and CROSSSIM as the similarity engine. The two algorithms have been chosen as they possess characteristics that are suitable for the problem of clustering OSS projects, such as effectiveness and efficiency. Moreover, we also present a supervised approach that exploits a Fuzzy ARTMAP neural network [8],[9] for clustering input vectors.

Name/Authors	Input Features	Clustering Algorithms	Dataset	Output
Nguyen et al. [54]	Identifiers' names (variables and methods), API usages, string literal tokens, code tokens	Deep Learning neural network	SourceForge Java projects	Clusters
SCC [5]	Code snippets	Multinomial Naive Bayes [19],[35]	StackOverflow posts	Clusters with labels as programming languages
BUCS [15]	Bytecode	Dirichlet Process Clustering technique (DPC) [76]	ASF jar files	Clusters with labels
Al-Subaihin et al. [4]	Apps' textual descriptions	Agglomerative hierarchical clustering [52]	BlackBerry and Google apps	Clusters of mobile apps
LDA-GA [73]	Text segments from `README.MD`	Latent Dirichlet Analysis [36]	Java projects	Clusters with labels
Sally [80]	Bytecode: class names, class fields, method names, and method arguments		Maven projects	A set of tags for each project
LeClair et al. [37]	Project names, function names, function contents	Three-layer neural network	C/C++ projects from the Debian packages repository	Clusters with label
Linares-Vásquez et al. [40]	Packages names, method names, classes, terms	Support Vector Machines [24], Naïve Bayes [85], Decision Tree [65], RIPPER [11], and IBK [3]	Sharejar and Sourceforge Java projects	Clusters

Table 2: Summary of software clustering techniques.

3 Evaluation Metrics

Depending on the availability of so-called ground-truth data, various quality metrics can be utilized to study the performance of our clustering algorithms. Given that no ground-truth dataset is available, *internal evaluation* is used to evaluate the connectedness and connectivity of the resulting clusters. Otherwise, when there is a dataset where every element has already been assigned specific classes, *external evaluation* can be conducted to measure the extent to which the produced clusters match the classes.

In the following we call the categories in a ground-truth dataset C = (C_1, C_2, .., C_k) as classes, with C_i being the set of repositories whose category is i, and the resulting groups of the clustering process $\hat{C}$ = ($\hat{C}_1$, $\hat{C}_2$, .., $\hat{C}_k$) as clusters. We performed external evaluation to measure the extent to which the produced clusters match the classes [69]. Given a clustering solution $\hat{C}$, the task is to compare the relatedness of the clusters in $\hat{C}$ to the classes in C. By the external validation, we exploit three evaluation metrics *Entropy*, *Purity* and *F-Measure* to analyze the clustering solutions since they have been widely utilized in evaluating clustering and appear to be effective [26, 69, 75, 89]. To further study the performance of our approach, we use three other internal validation metrics, i.e., *Silhouette Coefficient*, *Compactness*, and *Rand Index*.

The metrics are recalled in the following subsections.

3.1 External Validation

This section presents the metrics used for evaluating clustering given that ground-truth classes are available. In particular, we recall Entropy, Purity, and F-Measure as they have been widely used in evaluation of clustering solutions.

3.1.1 Entropy

This metric evaluates the relatedness of a cluster to the classes by measuring the presence of the classes in a cluster. We call p_{ij} the probability that an object of class i is found in cluster j, then Entropy for cluster j is computed according to the probability of the existence of all classes in j:

$$E_j = -\sum_{i=1}^{\kappa} p_{ij} \cdot log(p_{ij}) = -\sum_{i=1}^{\kappa} \frac{\left|C_i \cap \hat{C}_j\right|}{\left|\hat{C}_j\right|} \cdot log\left(\frac{\left|C_i \cap \hat{C}_j\right|}{\left|\hat{C}_j\right|}\right) \tag{1}$$

The Entropy value for a clustering solution is weighted across all clusters using the cardinality of each cluster:

$$E = \frac{1}{n}\sum_{j}^{\kappa} \left|\hat{C}_j\right| \cdot E_j \tag{2}$$

where n is the number of objects being clustered. Following this definition, a better clustering solution has a lower entropy.

3.1.2 Purity

It is used to evaluate how well a cluster matches a single class in C using the following formulas. Similar to Entropy, the overall Purity of a clustering solution is computed using the following formula:

$$P_j = \frac{1}{\left|\hat{C}_j\right|} \cdot \underset{i}{max} \left\{ \left| C_i \cap \hat{C}_j \right| \right\} \tag{3}$$

and

$$P = \frac{1}{n} \sum_{j}^{\kappa} \left|\hat{C}_j\right| \cdot P_j \tag{4}$$

In contrast to Entropy, a good clustering solution is about to have a high Purity. When $C_i \equiv \kappa_i \mid_{i=1,..,n}$ Purity is equal to 1 and that means every cluster is exact the same as one pre-defined class.

3.1.3 F-Measure

The final F-Measure score for a clustering solution C is the sum of all weighted constituent scores according to the fraction of the cardinality of a class and the total number of documents as follows:

By this metric *Precision* and *Recall* are utilized as follows:

$$F_{ij} = \frac{2 \cdot precision_{ij} \cdot recall_{ij}}{precision_{ij} + recall_{ij}}$$

and

$$F = \frac{1}{n} \sum_{i}^{\kappa} |C_i| \cdot \underset{j}{max} \{Fij\} \tag{5}$$

Precision is the fraction of objects in class i that is found in cluster j, whereas Recall is the fraction of documents of cluster j in class i:

$$precision_{ij} = \frac{\left| C_i \cap \hat{C}_j \right|}{\left|\hat{C}_j\right|} \tag{6}$$

$$recall_{ij} = \frac{\left| C_i \cap \hat{C}_j \right|}{|C_i|} \tag{7}$$

A perfect clustering solution is found, i.e., $\hat{C}_i \equiv C_i \mid_{i=1,..,\kappa}$ when Entropy is equal to 0, whilst both Purity and F-Measure are 1.0. If the clusters are completely different to the classes then Purity and F-Measure are equal to 0. This implies that a good clustering solution has low Entropy but high Purity and F-Measure. It is expected that the modification helps increase Purity, F-Measure but decrease Entropy at the same time.

3.2 Internal Validation

This sections recalls three metrics, i.e., Silhouette Coefficient, Compactness, and Rand Index for evaluating clustering solutions given that no ground-data is available. Together with the external validation, these metrics help further study performance of a clustering algorithm.

3.2.1 Silhoutte Coefficient

Given an object within a cluster, the silhouette index is used to measure the similarity between the object and the cluster [70]. The metric gives an indication on how similar an object is to its containing cluster compared to other clusters. A silhouette value $S \in [-1, +1]$, where 1 implies that the object matches well with its containing cluster but not other clusters. A clustering solution is deemed to be good if all objects have a high silhouette value.

Let $X = \{X_1, X_2, ..., X_N\}$ be a set of N elements, and $C = \{C_1, C_2, ..., C_K\}$ be a possible clustering result in K partitions. For each element $i, 1 \leq i \leq N$, let $a(X_i)$ be the average distance between i and all other elements within the same cluster. Intuitively, $a(X_i)$ is understood as a measure of how well Xi is assigned to its cluster. Let $b(X_i)$ be the smallest average distance of X_i to all points in any other cluster C_j such that $Xi \notin C_j$. The cluster which is closest to X_i is the *neighbouring cluster* of X_i as it is the next best fit cluster for point i. The silhouette coefficient is defined as follows:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} = \begin{cases} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{cases} \tag{8}$$

3.2.2 Compactness

Similar to Silhouette, Compactness is an internal evaluation metric which evaluates clusters by comparing the inter-subset and intra-subset distances. In particular, the class compactness of a subset is defined as the ratio of the two distances. Formally, let $X = \{X_1, X_2, ..., X_N\}$ be a set of N elements, let $C = \{C_1, C_2, ..., C_K\}$ a possible clustering result in K partitions, and $a(X_i)$ be the sum of distances between i and all other elements within the same cluster, let $b(X_i)$ the distance between the element X_i and the most representative elements for the cluster that does not contain X_i, Compactness is defined using the following formula:

$$Compactness(k) = \frac{1}{N} \sum_{j=1}^{N} \frac{a_i}{b_i} \tag{9}$$

3.2.3 Rand Index

The Rand index [68] in clustering measures the similarity between two clusters. First, we consider the following notations:

- $S = \{o_1, \ldots, o_n\}$ is a set of n elements;
- X and Y are two partitions of S to compare such that: $X = \{X_1, ..., X_r\}$ is a partition of S into r subsets and $Y = \{Y_1, ..., Y_s\}$, a partition of S into s subsets

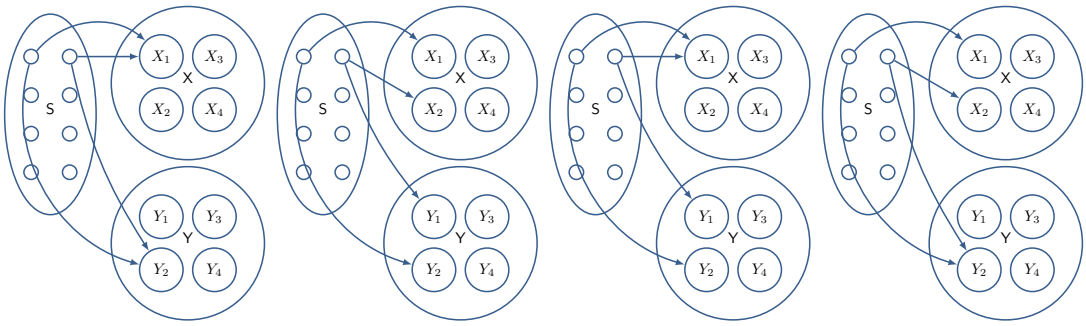


Figure 1: Pairs classification

- a is the number of pairs of elements in S that are in the same subset in X and in the same subset in Y (see Figure 1(a));
- b is the number of pairs of elements in S that are in different subsets in X and in different subsets in Y (see Figure 1(b));
- c is the number of pairs of elements in S that are in the same subset in X and in different subsets in Y (see Figure 1(c)), and
- d is the number of pairs of elements in S that are in different subsets in X and in the same subset in Y (see Figure 1(d));
- $a + b$ is the number of agreements between X and Y, $c + d$ the number of disagreements between X and Y.

The Rand index R is defined as follows:

$$R = \frac{a+b}{a+b+c+d} = \frac{2(a+b)}{n(n-1)} \tag{10}$$

The metrics introduced in this section are going to be used to evaluate our clustering tools presented in Section 4 and Section 5.

4 Unsupervised Clustering using K-Medoids and CLARA

This chapter presents an unsupervised tool for clustering OSS projects. As a base for our work, in Section 4.1 we recall the graph representation developed and evaluated in Deliverable D6.2 and D6.3. Section 4.2 introduces two versions of CROSSSIM to compute similarities among software projects. Afterwards, two clustering algorithms, i.e., K-Medoids and CLARA are briefly presented in Sections 4.3. Finally, in Section 4.4 we introduce an evaluation of the clustering process exploiting the above mentioned similarity and clustering algorithms on a dataset collected from GITHUB.

4.1 A Knowledge Graph to Represent the OSS Ecosystem

In Deliverable D6.2, we introduced a model to capture the intrinsic relationships among various artifacts in the OSS *ecosystem* [55],[56]. Both human i.e., developers and non-human factors such as source code, and software libraries are incorporated into a homogeneous representation by means of a semantic graph. A node represents either a person, such as developer, or an artifact, such as a source file, a library, and an edge represents the relationship between them. Such a representation considers all artifacts as a whole and allows for computing similarities by means of several graph similarity techniques.

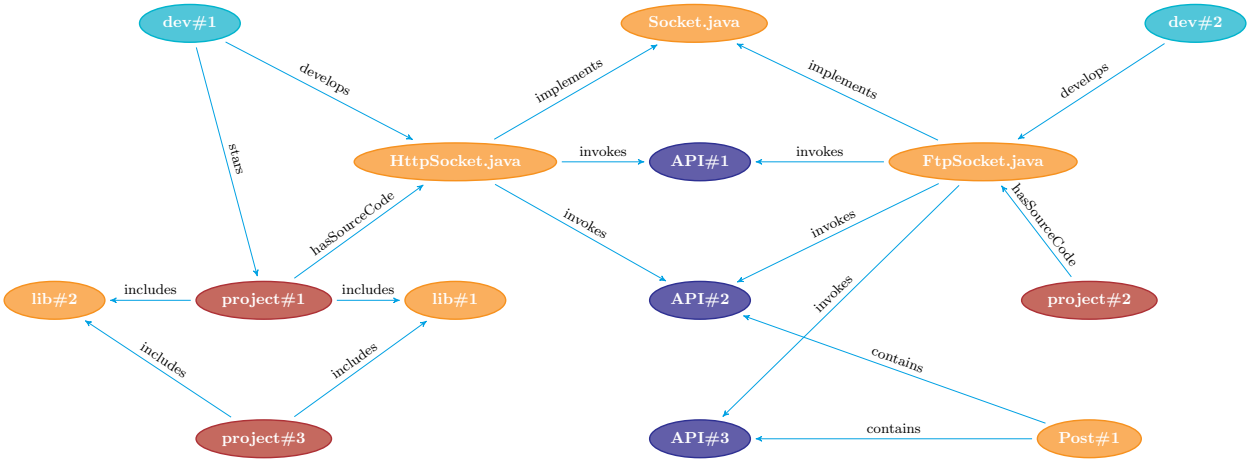


Figure 2: Representation of the OSS ecosystem.

Figure 2 depicts an example of the graph representation for various OSS artifacts. There are several semantic edges to describe the mutual relationship between graph nodes. For example, the edge `includes` describes the relationship between a project and a third-party library, whereas the edge `hasSourceCode` dictates that a project contains a source code file. The graph structure facilitates similarity computation [7]. For instance, several existing algorithms are able to compute the similarity between `project#1` and `project#2` as they are indirectly connected by the pair of edges, i.e., `hasSourceCode` and `implements` [55, 28]. This semantic path reflects the actual relevance of the projects, they contain classes that implement a common interface. The similarity is further enforced by another path via `hasSourceCode` and `invokes`, leading to two API functions, i.e., `API#1` and `API#2`. The two projects are a bit more similar since they invoke same APIs. Analogously, the similarity between `project#1` and `project#3` can also be inferred since they both include two third-party libraries `lib#1` and `lib#2`.

Using this graph, it is also possible to compute similarities among developers. We see that `dev#1` and `dev#2` share a common activity, they develop two classes that are somehow similar. The graph structure allows for

the computation of similarity between the two developers [55]. In practice, such a similarity can be computed by considering additional relationships.

To understand how to incorporate existing APIs into current code, a developer normally looks for API documentations that describe the constituent functions. For instance, StackOverflow provides the developer with a broader insight of API usage, and in some cases, with sound code examples [63]. In Figure 2, there is a StackOverflow post, i.e., `Post#1` contains code snippets with two function calls `API#2` and `API#3`. In practice, this a typical scenario when users discuss the usage of libraries containing `API#2` and `API#3`. In this respect, it might be helpful if we recommend `Post#1` to the developer of class `FtpSocket.java`, i.e., `dev#2`. This is completely feasible since the graph structure allows one to compute the similarity between `Post#1` and `FtpSocket.java`. In the end, a recommendation engine can provide the developer with a list of StackOverflow posts relevant to the code being developed.

In summary, with the adoption of the proposed graph representation, we are able to transform the relationships among humans, e.g., developers and non-human artifacts, e.g., API utilizations, source code, and interactions, into a mathematically computable format, which can facilitate several types of manipulations. Among others, in the next sections, we introduce two CROSSSIM configurations that are used to compute input for the clustering process. Different from the clustering approaches presented in Section 2, we are able to incorporate various features into the similarity computation.

4.2 Similarity Computation

In this section, we briefly recall two versions of the CROSSSIM implementation to compute similarities among OSS projects. The first tool exploits third-party libraries, star events as features. Meanwhile, the second one uses only third-party libraries as features. The final outcomes of both tools are a similarity matrix which is fed as input for the clustering module.

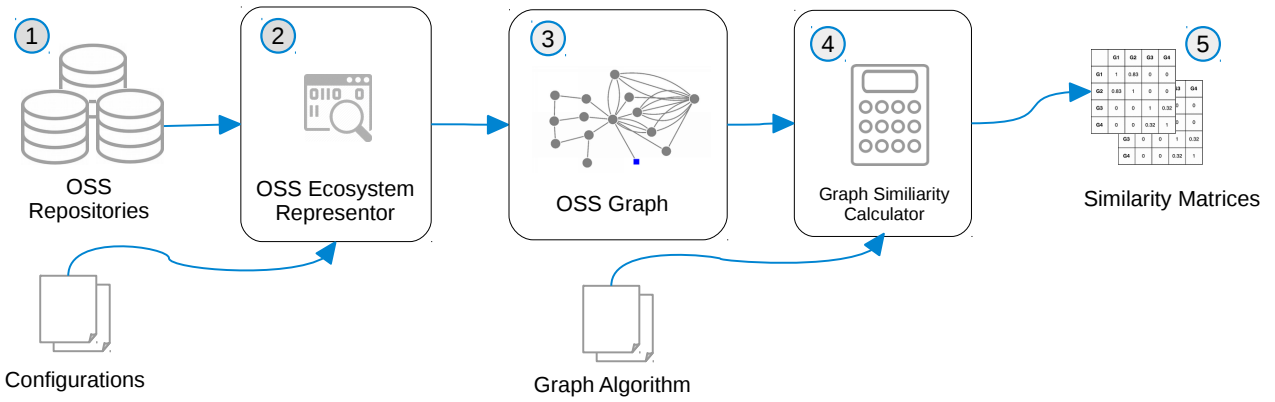


Figure 3: The CROSSSIM Architecture.

We recall the architecture of CROSSSIM which is depicted in Figure 3. In particular, the system imports project data from existing OSS repositories and represents them into a graph-based representation by means of the `OSS Ecosystem Representor`. Depending on the considered repository, the graph structure needs to be properly configured. For instance, in case of GITHUB, specific configurations have to be specified in order to enable the representation in the target graphs of the stars assigned to each project. Such a configuration is "forge" specific and specified once, e.g., SourceForge does not provide the star based system available in GITHUB. The `Graph Similarity Calculator` module, depending on the similarity function to be

applied, computes similarity on the source graph-based representation of the input ecosystems to generate matrices representing the similarity value for each pair of input projects.

Given a project, we get star and developer information using the GITHUB API[5], furthermore we parse the list of embedded third-party libraries from its `pom.xml`. Such as file defines all project dependencies with external Maven libraries [6]. An excerpt of a `pom.xml` file is depicted in Figure 4 as an explanatory example. All the dependencies of the project at hand are specified in the document between the `<dependencies>` ... `</dependencies>` tags.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.eluder.logback</groupId>
        <artifactId>logback-ext-core</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.eluder.logback</groupId>
        <artifactId>logback-ext-jackson</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.eluder.logback</groupId>
        <artifactId>logback-ext-aws-core</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.eluder.logback</groupId>
        <artifactId>logback-ext-sqs-appender</artifactId>
        <version>${project.version}</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>

</project>
```

Figure 4: An excerpt of a *pom.xml* file.

[5] GITHUB API: `https://developer.github.com/v3/`
[6] `https://maven.apache.org/guides/introduction/introduction-to-the-pom.html`

4.2.1 CROSSSIM$_C$: Computing similarities with various features

This implementation of CROSSSIM is called CROSSSIM$_C$ as it uses a combination of different features, i.e., *isUsedBy*, *develops*, and *stars*. We utilize SimRank [28] to calculate similarities based on mutual relationships between graph nodes. Considering two nodes, the more similar nodes point to them, the more similar the two nodes are. In this sense, the similarity between two project nodes p, q is computed by using a fixed-point function. Given $k \geq 0$ we have $sim^{(k)}(p,q) = 1$ with $p = q$ and $sim^{(k)}(p,q) = 0$ with $k = 0$ and $p \neq q$, SimRank is computed as follows:

$$sim^{(k+1)}(p,q) = \frac{\Delta}{|I(p)| \cdot |I(q)|} \sum_{i=1}^{|I(p)|} \sum_{j=1}^{|I(q)|} sim^{(k)}(I_i(p), I_j(q)) \tag{11}$$

where Δ is a damping factor ($0 \leq \Delta < 1$); $I(p)$ and $I(q)$ are the set of incoming neighbors of p and q, respectively. $|I(p)| \cdot |I(q)|$ is the factor used to normalize the sum, thus forcing $sim^{(k)}(p,q) \in [0,1]$.

In [55], we utilized CROSSSIM$_C$ to compute similarities for a set of 580 GITHUB Java projects. The outcomes of this work were then compared against those of REPOPAL [88] which computes similarities by exploiting `README.MD` files, star events and the time gap between two consecutive starring activities. We found out that CROSSSIM$_C$ obtains a superior performance with respect to various quality metrics, i.e., Success Rate, Confidence, Precision, and Ranking. The reason for the improvement in performance is that CROSSSIM$_C$ incorporates the mutual relationships among projects to compute similarities, whereas REPOPAL just takes into account the relationship between two projects being considered.

4.2.2 CROSSSIM$_L$: Computing similarities with third-party libraries

We derive from CROSSSIM [55] a tool that exploits third-party libraries as features for the similarity computation. To compute the similarities among OSS graph nodes, the approach proposed by Di Noia et al. [14] has been adopted. In this setting, two nodes in a graph are deemed to be similar if they point to the same node with the same semantic edge. Using this metric, the similarity between two project nodes p and q in an OSS graph is computed by considering their feature sets [14]. Given that p has a set of neighbour nodes $(lib_1, lib_2, .., lib_n)$, the features of p are represented by a vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_n)$, with ϕ_i being the weight of node lib_i. It is computed as the *term-frequency inverse document frequency* value $\phi_i = f_{lib_i} * log(\frac{|P|}{a_{lib_i}})$; where f_{lib_i} is the number of occurrence of lib_i with respect to p, it can be either 0 or 1 since there is a maximum of one lib_i connected to p by the edge *includes*; $|P|$ is the number of projects in the collection; a_{lib_i} is the number of projects connecting to lib_i via the edge *includes*.

Eventually, the similarity between p and q with their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_i\}_{i=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$ is computed as given below:

$$sim(p,q) = \frac{\sum_{i=1}^{n} \phi_i \times \omega_i}{\sqrt{\sum_{i=1}^{n} (\phi_i)^2} \times \sqrt{\sum_{i=1}^{n} (\omega_i)^2}} \tag{12}$$

where n is the cardinality of the set of libraries that p and q share in common [14]. Intuitively, p and q are represented as vectors in an n-dimensional space and Equation 12 measures the cosine of the angle between them.

CROSSSIM$_L$ has been successfully applied in CROSSREC to compute project similarities [57].

4.3 Clustering

In this section, we briefly recall two clustering algorithms, namely K-Medoids and CLARA as they are embedded in the CROSSMINER Knowledge Base as the clustering engine. We have chosen the two algorithms since they meet the requirements concerning effectiveness and efficiency which are of paramount importance in the context of clustering OSS projects.

4.3.1 K-Medoids

Clustering techniques have been widely used in different domains, for example to support document browsing and produce document summary [2]. For social networks, clustering is utilized in generating forecasts and recommendations. Among others, the K-Means algorithm has been widely used in clustering due to its simplicity. Nevertheless, the algorithm is prone to noise and outliers [53, 44]. K-Medoids was proposed to overcome the limitations of K-Means [31]. K-Medoids is a classical partitioning technique of clustering that groups n objects into k clusters known a priori. The algorithm deals with noise and outliers better compared to K-means since it minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances.

First, a set of initial medoids is generated randomly, then a medoid is selected as the object in the cluster that has minimum average distance to all objects in the cluster. Objects are assigned to the cluster with the closest medoid. K-Medoids performs clustering using a greedy strategy as follows:

- Step 1: Populate initial medoids by randomly selecting κ objects.
- Step 2: Assign each of the remaining objects to the cluster with the nearest medoid.
- Step 3: Calculate a new set of medoids. For each cluster, promote the object that has the smallest average distance to the other objects in the cluster to the new medoid. If there are no changes in the set of medoids, stop the execution and return the resulting clusters in Step 2. Otherwise go back to Step 2.

4.3.2 CLARA

CLARA (Clustering LARge Applications) has been proposed to cope with large datasets [32],[33]. The technique attempts to save processing time by finding medoids from subsets of data objects. CLARA shuffles various samples of objects and deploys K-Medoids on these samples to find the best set of medoids. The remaining objects are then assigned to their closest medoids. CLARA is briefly recalled as follows [33]:

Set $minD \leftarrow N$, N is a large enough number. Repeat the following steps for 5 times:

- Step 1: Choose randomly a set of $40 + 2\kappa$ objects. Apply K-Medoids on this set to find κ medoids.
- Step 2: Assign the remaining objects to the cluster whose medoid is closest to them.
- Step 3: Compute the average distance $avgD$ of the clustering solution in Step 2. If $avgD < minD$ then $minD \leftarrow avgD$ and select κ medoids in Step 1 as the current medoids. Go back to Step 1.

4.4 Evaluation

Once the similarity matrix for a set of projects has been computed by means of CROSSSIM, it is possible to cluster them. We evaluated the clustering performance of K-Medoids and CLARA on datasets consisting of 570 projects collected from GITHUB. The dataset contains a list of labeled repository by topics. We study the performance of proposed clustering techniques in combination with two similarity measures. The evaluation

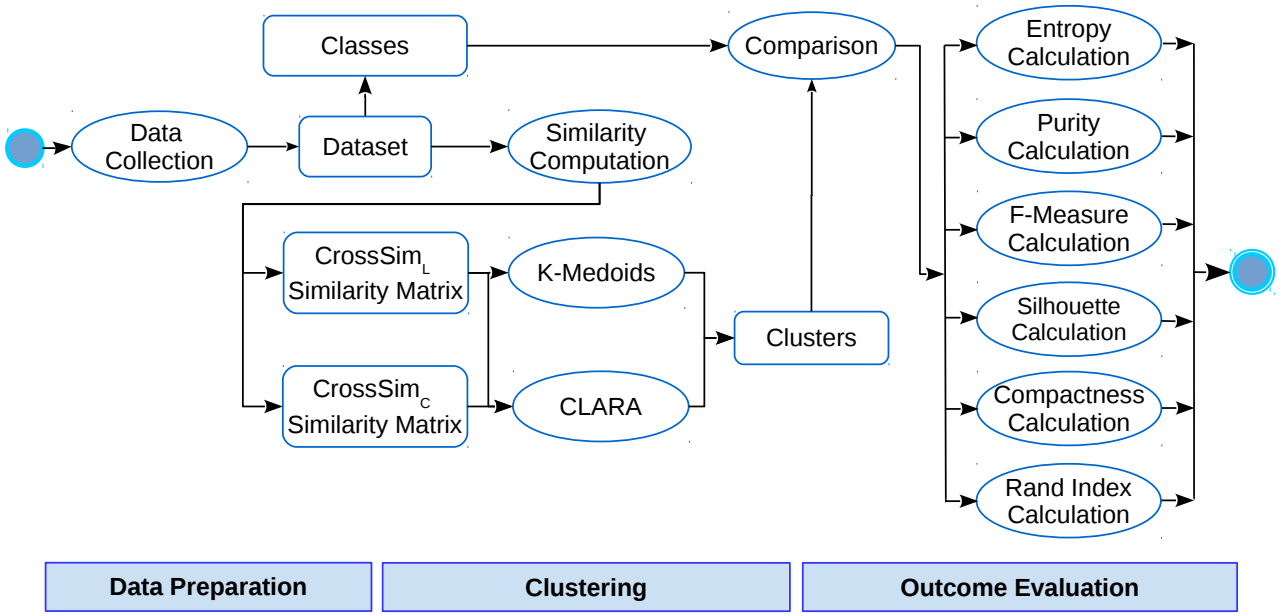


Figure 5: The evaluation process.

process is depicted in Figure 5. First, data is collected from various sources, such as GITHUB, SourceForge to create a dataset of decent quality to serve as input for similarity computation.

The related phases are detailed in the rest of this section.

4.4.1 Dataset

GITHUB recently launched a new functionality that allows developers to specify a description for each repository, aiming to facilitate future search queries. Topics are defined as any *"word or phrase that roughly describes the purpose of a repository and the type of content it encapsulates."*[7]. In this sense, topics are a means to describe a project in term of its intended purpose, subject area, affinity groups, or other important qualities.

We mined a dataset of 570 Java repositories that belong to the 47 most-used topics. In the data collection phase, we tried to cover a wide range of possibilities by taking into consideration that many repositories in GITHUB are of low quality, which is especially true when they do not have many stars. Thus, we consider only projects that have been starred by at least 20 developers. We believe that such projects are of decent quality since they have been acknowledged by a considerable large number of developers. The labels for a repository are assigned the main topics to which the repository belongs. A summary of the datasets is given in Table 3. The projects spread across a wide range of topics and among others *aws* is the topic containing more elements: 54 projects belong to this topic. Meanwhile *latex* is the topic with least projects: just 3 projects belong to this category.

[7] `https://githubengineering.com/topics/`

No.	Topic	# of repo	No.	Topic	# of repo	No.	Topic	# of repo
1	ai	5	**17**	data-structures	5	**33**	latex	3
2	angular	13	**18**	data-visualization	6	**34**	postgresql	26
3	arduino	3	**19**	deep-learning	9	**35**	privacy	15
4	aws	54	**20**	deployment	7	**36**	raspberry-pi	14
5	azure	17	**21**	discord	8	**37**	redux	9
6	blockchain	17	**22**	documentation	8	**38**	scala	31
7	bootstrap	17	**23**	ethereum	13	**39**	server	29
8	bot	12	**24**	game-engine	15	**40**	serverless	9
9	chrome	3	**25**	git	20	**41**	shell	5
10	cli	8	**26**	go	3	**42**	swift	6
11	clojure	4	**27**	google	8	**43**	telegram	4
12	cms	7	**28**	google-cloud	9	**44**	csharp	6
13	code-quality	7	**29**	graphql	29	**45**	css	9
14	compiler	9	**30**	hacktoberfest	7	**46**	jquery	9
15	continuous-integration	11	**31**	html	12	**47**	kubernetes	36
16	cryptocurrency	7	**32**	image-processing	6			

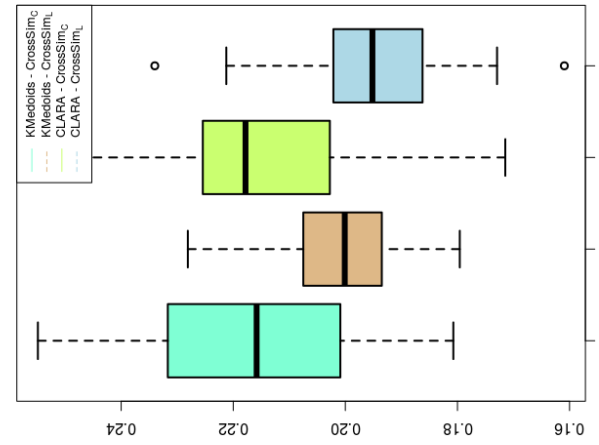
Table 3: A summary of the collected dataset.

4.4.2 Experimental Settings

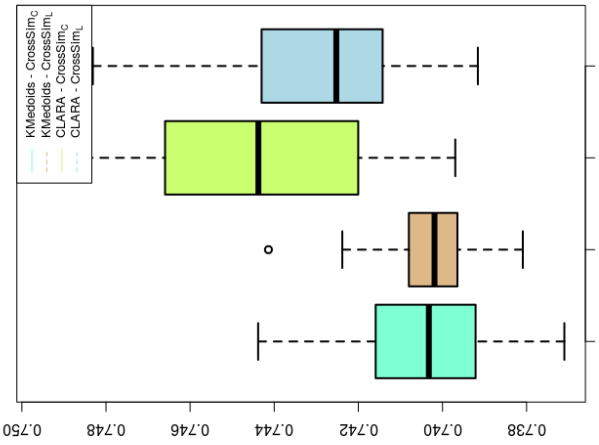
Using the datasets described in Table 3, we performed independent experiments, each corresponds to applying one distance measure, i.e., CROSSSIM$_L$ and CROSSSIM$_C$ and a clustering algorithm, i.e., K-Medoids or CLARA to the repository set. From the distance scores, CLARA, and K-medoids were applied to produce clusters. To aim for randomness, every clustering experiment was run in 50 independent trials.

4.4.3 Results

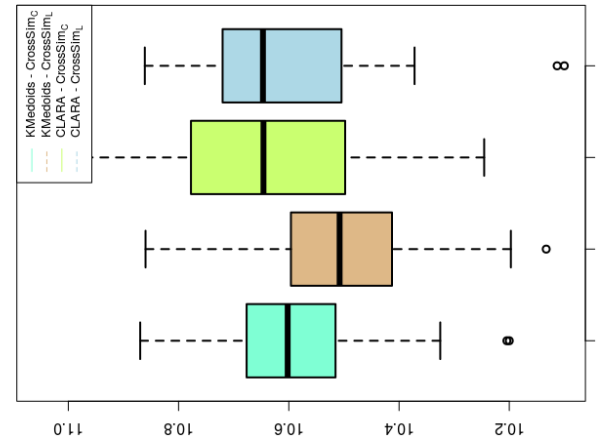
The results we obtained after performing K-Medoid and CLARA using CROSSSIM$_L$ and CROSSSIM$_C$ as the similarity computation engine are depicted in Figure 6 and Figure 7. As shown in Figure 6(a), K-Medoids helps obtain a better Entropy for both CROSSSIM$_L$ and CROSSSIM$_C$ compared to CLARA. However, the Purity by applying CLARA is superior to those of K-Medoids, using either CROSSSIM$_L$ or CROSSSIM$_C$ as the similarity measure.



(c) F-Measure.

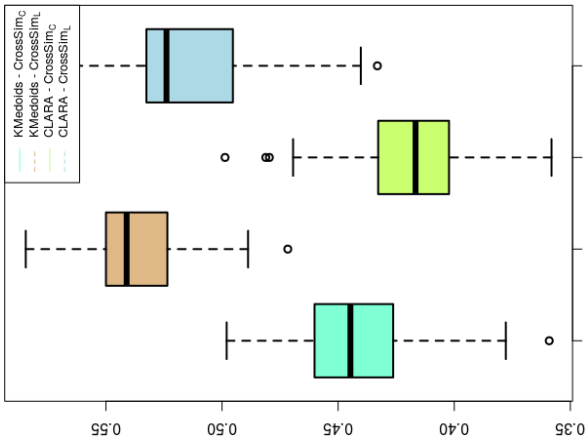


(b) Purity.

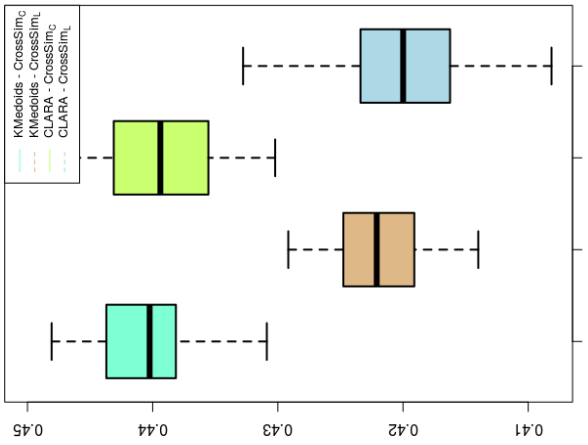


(a) Entropy.

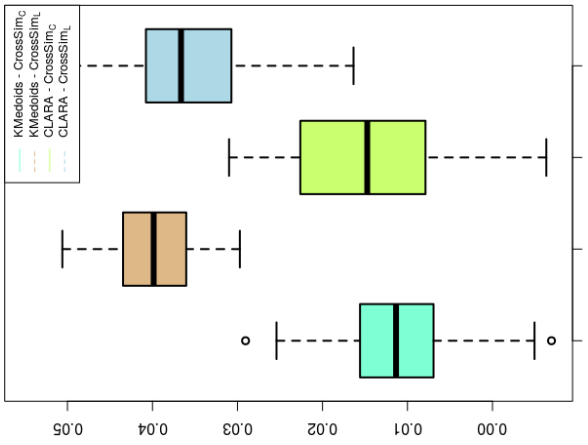
Figure 6: External validation.



(c) Rand Index.



(b) Compactness.



(a) Silhouette.

Figure 7: Internal validation.

Concerning F-Measure (see Figure 6(c)), CROSSSIM$_C$ is beneficial to the application of K-Medoids and CLARA since the F-Measure for these cases is superior to that when CROSSSIM$_L$ is used as the similarity computation engine. This implies that using as a combination of third-party libraries, star events, and development activities as features for computing similarity as presented in Section 4.2.1 is more effective than using just third-party libraries following Section 4.2.2.

The silhouette values are shown in Figure 7(a). Following Equation 8, a higher score implies a better Silhouette. Thus, in contrast to F-Measure, we see that CROSSSIM$_L$ helps obtain a better Silhouette value than CROSSSIM$_C$ for both K-Medoids and CLARA. This suggests that if we want to achieve a high accuracy (F-Measure) by using a combination of CROSSSIM$_C$ with either K-Medoids or CLARA, we will suffer from a deficit in Silhouette.

The same trend for Silhouette is also witnessed by Compactness (Figure 7(b)), where a combination of CROSSSIM$_C$ with either K-Medoids and CLARA brings a better compactness than when CROSSSIM$_L$ is used instead. However, CROSSSIM$_L$ is beneficial to Rand Index as shown in Figure 7(c). By combining CROSSSIM$_L$ with K-Medoids and CLARA, a superior Rand Index is obtained compared to the case when CROSSSIM$_L$ is used to compute similarity.

In summary, we see that CROSSSIM$_L$ contributes to obtaining better Accuracy, i.e., Precision, Recall, as well as Rand Index. There is no big difference in performance of K-Medoids and CLARA for all testing configurations.

5 Supervised Clustering using a Neural Network

In some open source platforms, such as SourceForge[8] or Maven[9], projects are normally assigned to specific categories, with the aim of facilitating the search process. The labeling is manually done, i.e., when developers start creating or uploading a project, they will specify one or more categories to the project[10]. The labels should be selected so as to provide a comprehensive summary to the project's functionalities. Figure 8 depicts an example of the SourceForge project named `Java HTTP Proxy Library`[11]. According to its description, the project *"assists in recording and troubleshooting http by providing an implementation of a proxy and having extensible classes to handle post/pre processing which can be used to carry out custom actions or logging of the transactions,"* and thus it has been assigned to four categories, i.e., *Browsers*, *HTTP Servers*, *Networking*, and *Quality Assurance*.

These categories are a means to help developers narrow down the search scope and efficiently approach the project. For instance, when another developer looks up the *Browsers* category, she will end up having a list of projects related to this topic, and among others, `Java HTTP Proxy Library` will also be shown up.

Figure 8: An example of a SourceForge project and its categories.

Recently, GitHub has deployed a similar functionality which allows developers to manually assign projects to categories [12],[13]. Furthermore, given a set of repositories, GitHub also recommends probable topics by automatically analyzing their content. Nevertheless, such a recommendation is still a semi-automatic process as it is incumbent upon GitHub administrators to decide whether to make use of the recommended topics or not[14].

[8] `https://sourceforge.net/`
[9] `https://mvnrepository.com/`
[10] `https://sourceforge.net/p/forge/documentation/Create%20a%20New%20Project/?version=30`
[11] `https://sourceforge.net/projects/wpg-proxy/`
[12] `https://github.com/showcases/search`
[13] `https://help.github.com/articles/classifying-your-repository-with-topics/`
[14] `https://help.github.com/articles/about-topics/`

It is our firm belief that the prescribed information related to projects and their corresponding categories is meaningful: it reflects the perception of humans towards the relationship between projects and categories. We hypothesize that projects' features, such as source code, metadata can be exploited to automatically group projects into categories. In other words, it is reasonable to cluster OSS projects by attempting to simulate humans' cognition towards the projects-categories relationship, using the available data.

In this section, we present our proposed approach to support supervised clustering of OSS projects, i.e., by directly learning from labeled data. We propose OSCAN, a tool for **O**pen Source **S**oftware Projects **C**lustering using a Fuzzy **A**RTMAP **N**eural network. In Section 5.1, we introduce background studies related to Fuzzy ARTMAP neural network and possible applications. Afterwards, in Section 5.2 we detail the implementation of OSCAN. Finally, Section 5.3 presents a preliminary evaluation of our proposed approach on a SourceForge dataset.

5.1 Fuzzy ARTMAP

Fuzzy ARTMAP (FAM) [8],[9] is a neural network algorithm based on the Adaptive Resonance Theory (ART) [21]. A FAM is a classifier that accepts a set of vectors as input and returns a set of clusters as output. Originally, FAM was designed to work with unsupervised clustering, however given that labeled data is available, i.e., there are projects and the corresponding categories, it is possible to apply the supervised learning technique [87]. In this section, we recall Fuzzy ARTMAP neural network for supervised learning as a base for further presentation.

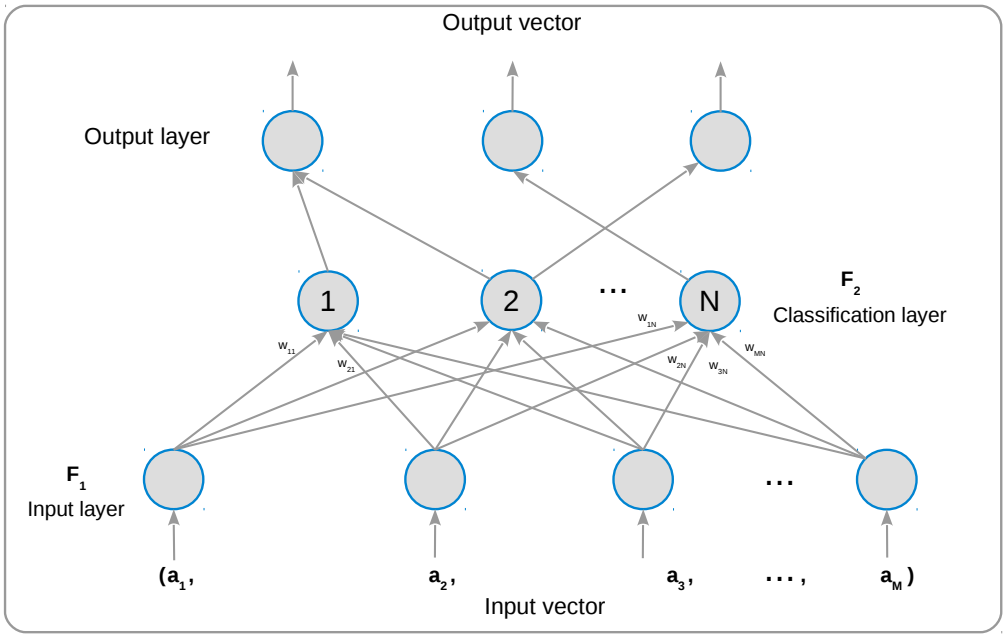


Figure 9: A Fuzzy ARTMAP neural network.

An example of a Fuzzy ARTMAP neural network is depicted in Figure 9. Such a network consists of three layers. F_1 is the input layer and it accepts data in the form of a vector, and every node in F_1 connects to all nodes in F_2. An F_2 node j learns a prototype vector $w_j = \{w_{j1}, w_{j2}, \ldots, w_{jN}\}$ that represents a recognition category. The last layer F_3 associates each F_2 category to a certain class.

The input data for this network is a set of vectors, each has the form of a = $\{a_1, a_2, \ldots, a_m\} \in \mathbb{R}^m$, where m is the number of input features, a_i is either 0 or 1, and 1 implies that the object possesses the feature and 0 otherwise. The learning phase is the process that FAM selects a winning node in F_2 and adapts the corresponding weights to fit in the input vector. Before going into details, to facilitate further presentation, we introduce the following definitions.

- Before being fed to the network, an input vector a is converted into its complement-coded format I as given below.

$$I = [a, a^c] \equiv [a_1, a_2, \ldots, a_m, a_1^c, a_2^c, \ldots, a_m^c] \tag{13}$$

  where $a_i^c = 1 - a_i; i = \overline{1, m}$
- For each neuron j in the second layer, there is a weight vector w_j associated with it and w_j is called the prototype of neuron j.

$$w_j = [w_{1j}, w_{2j}, \cdots, w_{Mj}] \tag{14}$$

  where $M = 2 \times m$. Initially, all entries of a weight vector are set to 1.
- The fuzzy AND operator is defined as follows:

$$(a \wedge b)_i \equiv min(a_i, b_i) \tag{15}$$

  Equation (15) implies that the fuzzy AND favors the smaller value.
- The norm of a vector is the sum of its entries:

$$|a| \equiv \sum_{i=1}^{m} |a_i| \tag{16}$$

  The definition in Equation (16) implies that $|I| = m$.
- An F_2 neuron is said to be committed if it wins and is assigned with labels from an input vector.
- The vigilance factor $\rho \in [0, 1]$ is used to control the threshold that an input vector is considered to be associated with a prototype.
- α is a choice parameter, $\alpha \in [0.001, 10]$, and it determines the number of categories, i.e., the number of neurons in the F_2 layer.
- Epoch is a complete round of introducing all input vectors to the neural network [8]. The number of epochs can be empirically set, depending on the granularity of the outcome.

The main steps of a Fuzzy ARTMAP are summarized as follows [9]:

- For each input vector I, find a neuron J whose prototype w_J matches with the vector. J is said to be the winner for I.
- If neuron J satisfies the vigilance requirement, the learning process will be triggered.
- Otherwise, select another winner by using an increased value of the vigilance threshold ρ.
- If the labels of a match with those of J, modify w_J to make it better reflect a.
- If no neuron can be considered as winner, create a new neuron and set its prototype to be the complement-coded input vector I.
- The process continues for all input vectors with a number of iterations (epochs).

In summary, by an epoch, each input vector is fed to the neural network which then performs computation to assign the vector to an F_2 node. A node is said to win if its weight vector and the input vector are similar, i.e.,

by exceeding a certain threshold [8],[9]. The winning node is assigned the class(es) of the input vector and the learning process happens, that means the weight vector is adjusted so as to be closer to the input vector. A detail description of both learning and testing phases is presented in Section 5.2.

5.2 OSCAN: A Neural Network to Cluster OSS Projects

OSCAN has been implemented using a simplified version of Fuzzy ARTMAP proposed by Vakil-Baghmisheh et al. [79]. The OSCAN architecture is depicted in Figure 10. There are two phases: learning and testing. Essentially, learning is the process of adjusting the weight vectors of the second layer. This is done by performing computation on the labeled data to learn the weights for all nodes in the F_2 layer. Whereas by the testing phase, the weights are used to classify input vectors. First, data is collected from different OSS platforms, such as GITHUB, SourceForge ①. Afterwards, it is passed through the `Data Extractor` module to be converted to binary vectors together with labels ②. The `Weight Calculator` computes F_2 layer weights using the input vectors ③. The final outcome of this phase is a neural network represented by concrete values of w_j ⑤. In the testing phase, we exploit the neural network to cluster projects without labels. OSCAN accepts input data as projects' artifacts ④, e.g., source code and converts into the form of vectors and labels using the `Data Extractor` ②. The neural network previously learned ⑤ is used to cluster the input vectors. Eventually, a set of clusters is produced ⑥.

In practice, learning is the building process where the system is trained with data collected from OSS platforms. Meanwhile, testing corresponds to the deployment phase, i.e., the system is used to cluster projects whose classes are unknown.

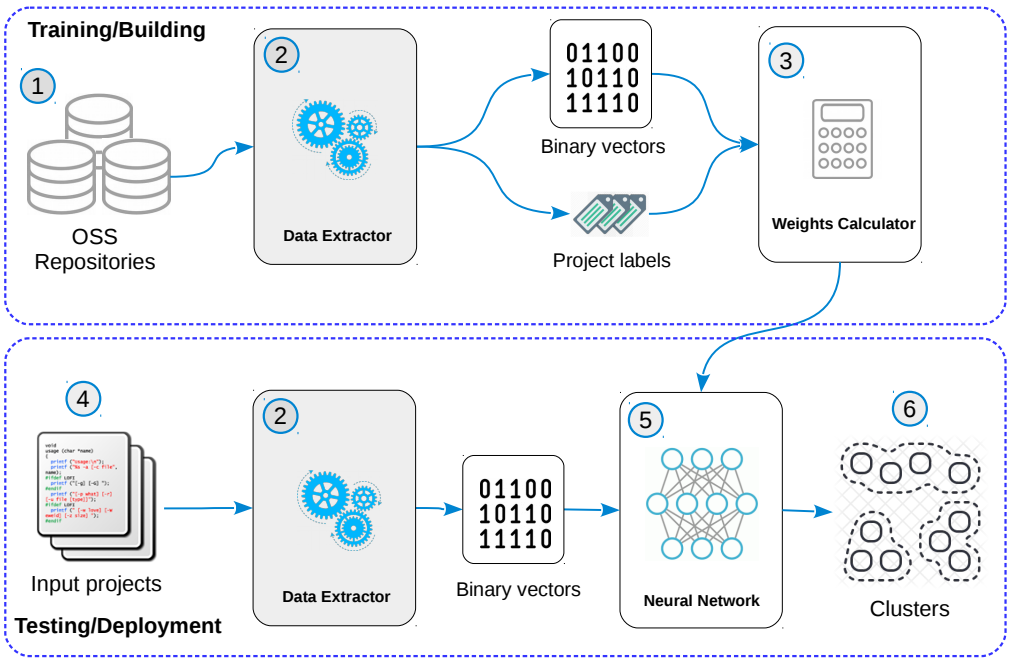


Figure 10: The OSCAN architecture.

5.2.1 The training phase

The learning phase is depicted in the upper part of Figure 10. It is the process that the neuron adjusts its weights to be closer to the input vector. This is the ability to memorize and learn from a piece of input data, the neural network assigns a vector to a cluster and by the next time, a highly similar input vector will be assigned to the same cluster.

1. Step 1: A new input vector a is converted to its complement-coded form and fed to the network. Compute second layer activities:

$$T_j = \frac{|I \wedge w_j|}{\alpha + |w_j|}, j = 1, \ldots, N \tag{17}$$

2. Step 2: Category choice: A node J is selected as the winner if it obtains the maximum value:

$$T_J = max\{T_j; j = 1, \ldots, N\} \tag{18}$$

3. Step 3: If the winning neuron J is not committed, then the input vector is assigned as its prototype $w = I$ and the current epoch continues with a new input vector (Back to Step 1).

4. Step 4: If the chosen neuron satisfies the match criterion:

$$\frac{|I \wedge w_j|}{|I|} \geq \rho \tag{19}$$

   then resonance is said to occur and the learning ensues, i.e., go to Step 5. Otherwise, J is marked as not suitable for input vector a during the current epoch. The process tries to seek a better one by going back to Step 3.

5. Step 5: Learning happens to adjust the winning neuron's weights to better reflect the input vector.

$$w_J^{(new)} = \beta(I \wedge w_J^{(old)}) + (1 - \beta)w_J^{(old)} \tag{20}$$

   β is the learning rate and normally set to 1 by many existing studies [79],[87]. The process moves back to Step 1 and introduces a new input vector.

5.2.2 The testing phase

As shown in the lower part of Figure 10, by the testing phase, there is no predefined label for input vectors and it is necessary to find proper labels for them. The testing phase resembles the training one in some steps. However, it is more simple since there is no label matching as well as no weight adapting. Furthermore, testing is done in only one epoch. In particular, the testing phase is conducted as follows.

1. Step 1: A new testing input vector a is converted to its complement-coded form and fed to the network. Compute second layer activities.

2. Step 2: A neuron J with the maximum value T_J is selected as the winner.

3. Step 3: The input vector is assigned the label(s) of the winner J. Go back to Step 1 with a new input vector.

5.2.3 Category proliferation

The ρ value in Equation (19) contributes to the number of neurons needed for layer F_2. A low value of ρ will increase the number of vectors that meet the match criterion and thus prolonging the overall processing time. Meanwhile, a high ρ value will make more difficult for an input vector to find a suitable neuron, and therefore a new neuron needs to be created. Given the circumstance, the number of neuron in the second layer may explode and lead to the so-called problem *category proliferation* [64]. We derive the approach proposed by Zhang et al. [87] to reduce the number of necessary categories. Before adding a new category, a threshold is set so as all input vectors that exceed this threshold will be selected. The algorithm estimates the class posterior probability for each category and selects the one with the highest value. The network resorts to creating a new category only when no category exists for the input pattern.

5.3 Evaluation

This section presents an evaluation of OSCAN on a set of OSS projects. In Section 5.3.1 we introduce the dataset exploited in the evaluation and in Section 5.3.2, we explain the experimental settings. Finally, the outcomes of the experiments are given in Section 5.3.3.

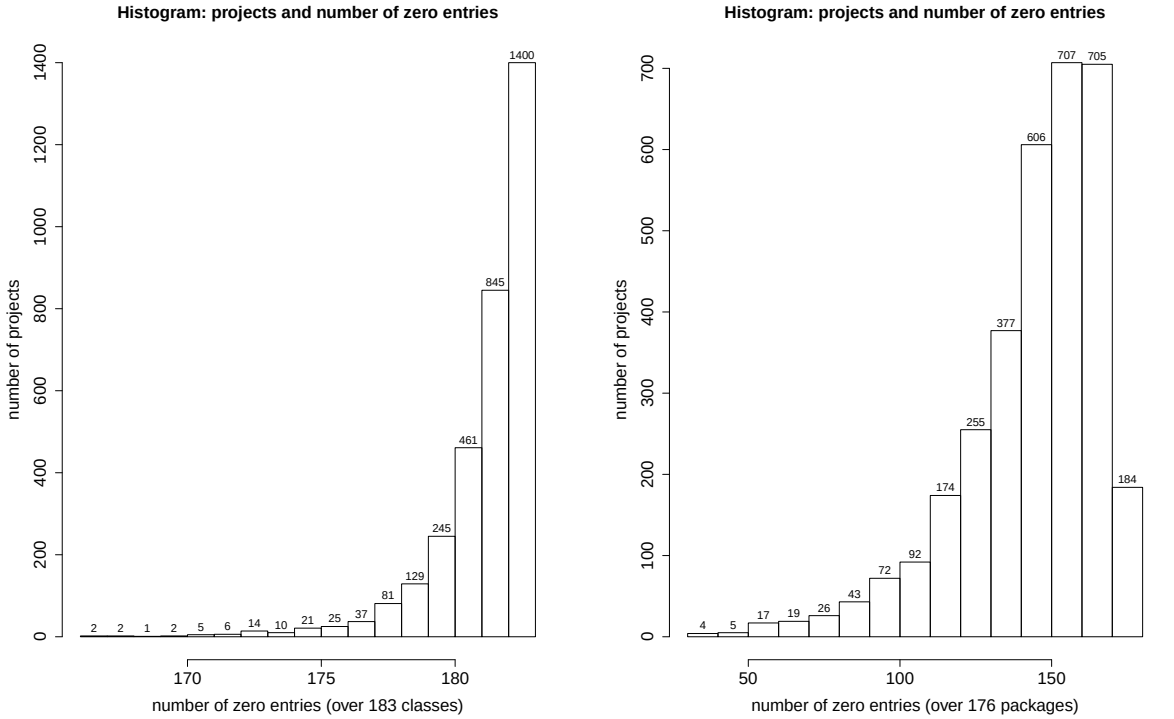
5.3.1 Dataset

The evaluation of OSCAN necessitates labeled data where projects are assigned with specific categories. We investigate whether the datasets accompanied the publications in [40],[47] are suitable for the evaluation of OSCAN. There are two independent datasets, one from SourceForge and the other from Sharejar, both provided in the WEKA `ARFF` file format[15] (WEKA is a software library providing various machine learning algorithms). The former contains $3,286$ projects and the latter has 745 projects. For each `ARFF` file, right after the `@data` tag there are several rows and each corresponds to one vector representing a project. Each entry specifies the occurrence of the corresponding feature, i.e., 1 means that the feature is present in the project, 0 otherwise. By carefully observing the files, we realized that the SourceForge dataset is of poor quality, many projects just contain all 0 in their entries. Figure 11(a) and Figure 11(b) show the number of zero entries that a project vector contains. For instance, with the `sourceforge-classes.arff` file (Figure 11(a)), there are $1,400$ projects which have all entries as zero (42.6%). We suppose that such projects are useless for any classification task since they contain no practical information, i.e., entropy is equal to 0.

Eventually, by applying the same checking method, we found out that the Sharejar dataset contains decent labeled data, i.e., most projects have non-zero entries together with zero entries. Thus, we decided to get rid of the SourceForge dataset and exploit the one from Sharejar for our evaluation. Table 4 provides a summary of the categories and their cardinality of the Sharejar dataset [40]. Among others, the Chat & SMS category is the largest one, with 320 projects whereas Programming is the smallest category with 10 projects. It is worth noting that one project may belong to more than one category (as depicted in Figure 8).

In the Sharejar dataset, there are three independent types of features: Packages, Methods, and Classes, and they are stored in `sharejar-packages.arff`, `sharejar-methods.arff`, and `sharejar-classes.arff`, respectively. We conduct different experiments on the files to measure the recommendation performance with respect to various features.

[15] `https://www.cs.waikato.ac.nz/ml/weka/arff.html`



(a) File `sourceforge-classes.arff` (b) File `sourceforge-packages.arff`

Figure 11: A summary of the SourceForge dataset.

No.	Category	# of projects	No.	Category	# of projects
1	Chat & SMS	320	**8**	Music	50
2	Dictionaries	30	**9**	Science	20
3	Education	90	**10**	Utilities	190
4	Free Time	120	**11**	Emulators	30
5	Internet	180	**12**	Programming	10
6	Localization	20	**13**	Sports	40
7	Messengers	50			

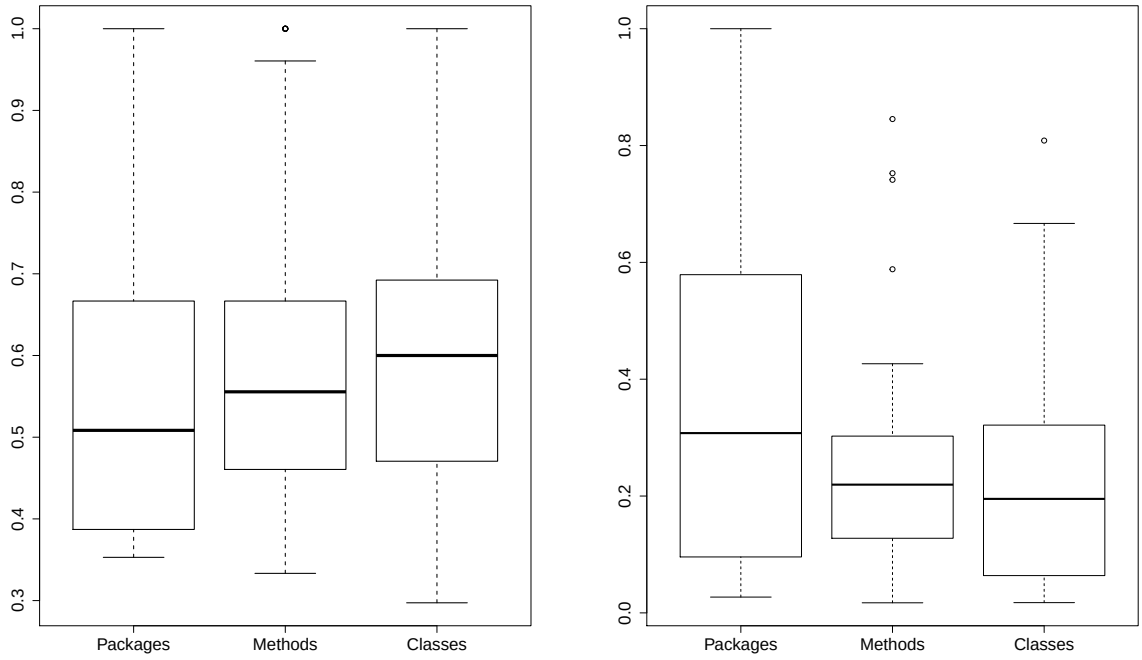
Table 4: A summary of the projects in the Sharejar dataset [40].

5.3.2 Experimental Settings

We separately use packages, methods, and classes as features to provide as input for OSCAN. The aim is to study which type of features will bring a better clustering performance. Similar to the evaluation in [47] on the Sharejar dataset, we apply *five-fold cross validation* [83], i.e., four parts of the data are used for training the remaining part is used for testing. The evaluation has been conducted 5 times and by each iteration, 4 parts with projects and their labels are used to train OSCAN and the remaining part is used as testing data. For each testing project, its classes are removed to use as ground-truth data. The outcome of the testing phase is a set of clusters with labels. We compare the obtained clusters with the ground-truth classes to see how well OSCAN assigns projects to clusters.

	Entropy			Purity		
Fold	Packages	Methods	Classes	Packages	Methods	Classes
1	**2.42**	4.05	4.05	0.52	0.65	**0.79**
2	**2.23**	3.81	3.27	0.48	0.81	**0.97**
3	**2.36**	3.32	2.53	0.63	0.76	**0.78**
4	**1.19**	1.58	1.21	0.75	0.78	**0.81**
5	**1.89**	1.86	3.32	0.59	0.61	**0.90**

Table 5: Entropy and Purity for the Sharejar dataset.



(a) Precision

(b) Recall

Figure 12: Precision and Recall for the Sharejar dataset.

5.3.3 Results

Different from the experiments presented in Section 4.4 in this evaluation, we are interested in how well the clusters produced by OSCAN match the original classes. Thus, we consider the external validation metrics, i.e., Entropy, Purity, Precision, and Recall. Table 5 shows the Entropy and Purity scores obtained from the experiments by using packages, methods, and classes as features for the clustering process.

Entropy: As a lower entropy represents a better clustering solution, we see that using packages as features brings a better Entropy for all testing folds. For instance, the maximum value for Entropy is 2.42 for Packages. However the corresponding scores for Methods and Classes are 4.05. This means that the clusters produced by using packages as features are more related to the original classes.

Purity: A better performance is obtained when Classes are used as features. The best purity score is 0.90 and it is obtained in Fold 5 where OSCAN uses vectors of class name as input data.

Accuracy: Figure 12(a) and Figure 12(b) show the precision and recall scores obtained from the evaluation outcomes, respectively. Figure 12(a) suggests that the precision yielded by using packages, methods and classes as features is comparable. A slightly better Precision is seen when classes are used as input for clustering. Meanwhile with Recall, using Packages as features helps bring a considerably better performance. For instance, in Figure 12(b) a Recall of 0.58 is obtained and it is the maximum value among others when packages are used as feature.

Altogether, we see that using packages as the input feature for OSCAN helps obtain a better clustering performance compared to other types of input data. This is somewhat consistent with the finding by McMillan et al. [47] where the authors state that package names are the most useful set of features.

5.4 Discussions

In this section, we presented OSCAN as a way to learn clustering from labeled data. In our evaluation, the experimental results show that OSCAN is able to produce final clusters that in turn match with those assigned by humans. Nevertheless, the overall performance is still not very high as expected. The reason for this is due to the lack of data as there were only 745 projects available for the training process. In practice, we expect to improve the performance by feeding the neural network with more well-defined data. For future work, we are going to study the performance of OSCAN by using more data from different sources, such as GITHUB. Furthermore, we will investigate additional types of features as input for the clustering process by exploiting the infrastructure developed from other partners' packages as well as from our previous working phases. Among others, we anticipate the following types of features: API function calls, third-party libraries.

In the context of Work Package 6, the application of OSCAN paves the way for further developments. In particular, a neural network has the potential to be deployed also for providing recommendations [25],[29]. For the next phase of our work package, we will be addressing the use cases related to API migration and the application of a neural network seems to be feasible.

A conventional recommender system works with explicit ratings, i.e., the preference of a customer towards an item is clearly specified with real scores. However, in case there is not enough user-item interaction information, or in other words no ratings are present, implicit ratings (such as interest towards an item during a *session*, demonstrated as clicking and viewing) can also be exploited to generate recommendations. To this end, each click on an item by the user is recorded and serves as the input for the recommendation [25]. The proliferation of deep learning techniques in recent years has opened a new era of recommender systems [29]. Among others, a *session-based recommender system* can be equipped with a deep neural network, i.e., one with several convolutional and standard layers of processing units, to handle sequential data, such as the mouse clicking events. This type of systems is applicable to mining OSS repositories, it can be applied to recommend artifacts that are time-constraint, such as software evolution.

6 REST API

In this section we discuss the integration of the clustering tools presented in the previous section into the CROSSMINER Knowledge Base. The component diagram initially presented in Deliverable D6.1: *Architecture specification of the CROSSMINER Knowledge Base* is depicted in Figure 13 by showing the light green the components that have been completely implemented. The ones that need more development and investigation efforts are shown in orange.

By focusing on the goals of this deliverable, the `ClusterCalculator` component in the one that builds clusters by relying on similarity functions and it is the implementation counterpart for Section 4. As depicted in Figure 13 three clustering algorithms have been implemented. `ClusterCalculator` is responsible for calculating clusters of analyzed artifacts as discussed in the previous sections. To this end similarity functions are used as implemented by the `SimilarityCalculator` component.

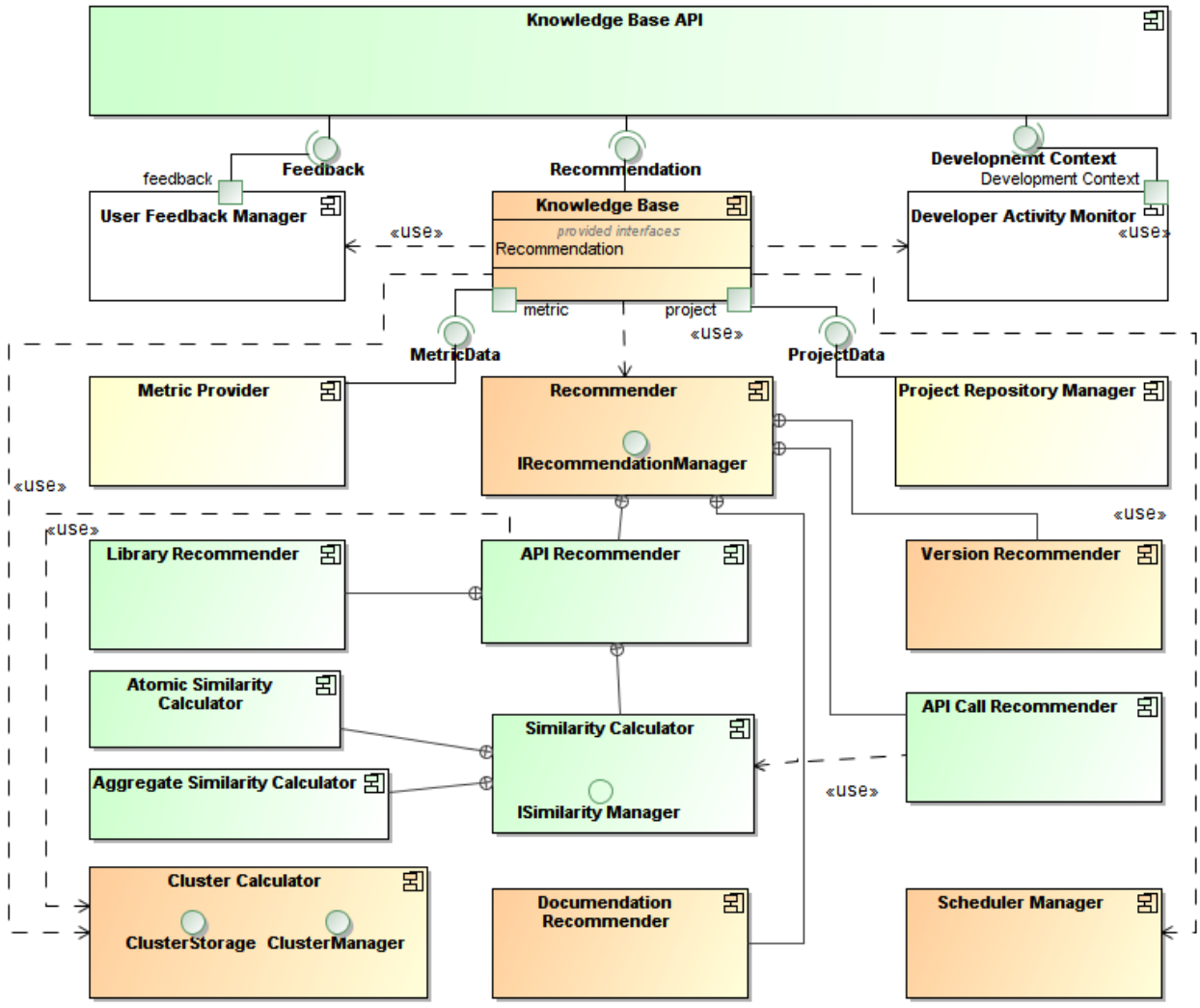


Figure 13: Knowledge Base component diagram

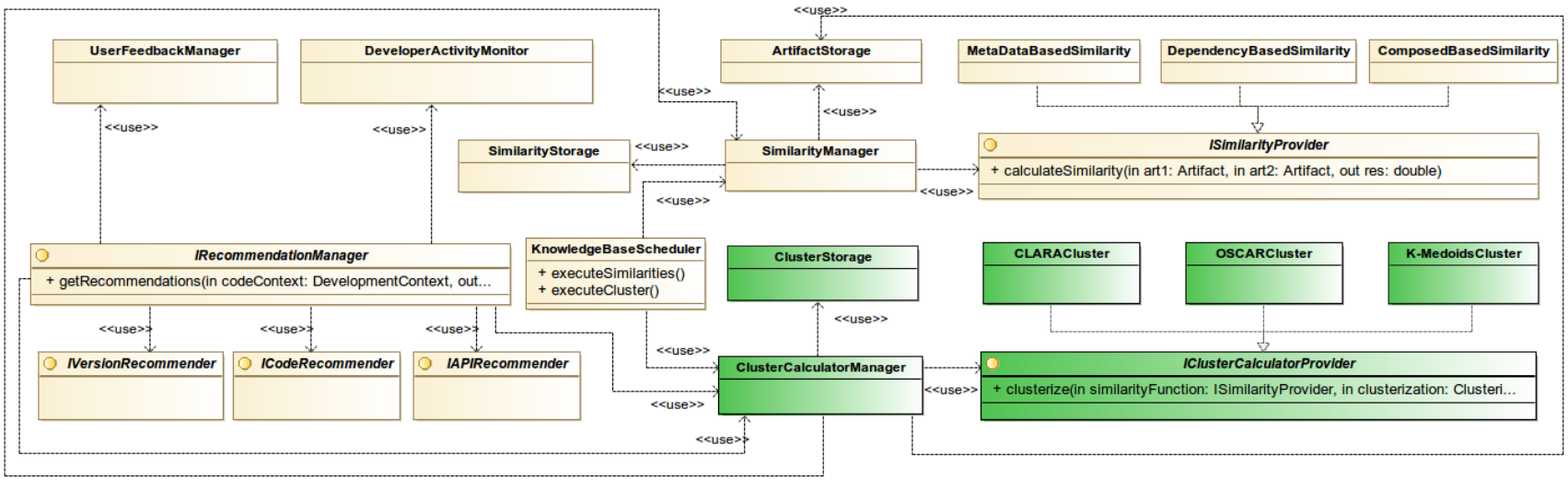


Figure 14: The Knowledge Base Development View.

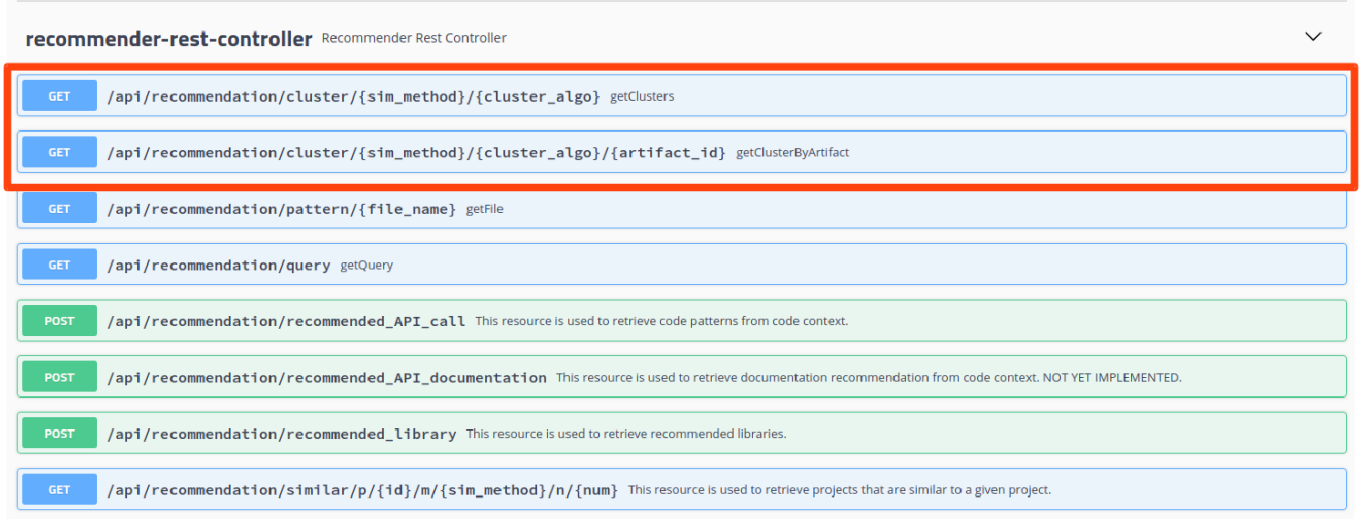


Figure 15: REST API - Swagger Documentation.

In particular, the deployment view depicted in in Figure 13 shows the actual software design of the cluster calculator component. The static structure of the system is illustrated in Figure 14. The `IClusterCalculatorProvider` interface is implemented by three different classes (each one implements the cluster algorithms proposed in Section 4 and 5). `ClusterStorage` relies on Spring data[16] which is fully integrated into the Spring framework. Spring Data MongoDB[17] is employed to make use of a MongoDB data store.

In the following subsections, we present the REST APIs that have been developed in order to enable the adoption of clustering components from the other related ones, and especially from the Web dashboards under development in Work Package 7. The implementation of the Knowledge Base as shown in Figure 14 is available at the CROSSMINER internal repository[18].

6.1 Get clustered projects

The CROSSMINER Knowledge Base can be invoked through a dedicated API. In this section, we present the implementation of the API by focusing on the methods related to the clustering features. We have integrated Swagger and OpenApi specifications into the Knowledge Base, which is accessible at `/swagger-ui.html` (see Figure 15). The Knowledge Base architecture automatically generates OpenAPI specification and Swagger interface starting from code annotations. The currently supported clustering operations (highlighted in red in Figure 15) of the Knowledge Base API are presented as follows.

Description: This resource is used to retrieve clusters of projects.

HTTP Method	**HTTP url**
GET	`cluster/{sim_method}/{cluster_algo}`
Path parameters	
Name	**Description**
`{sim_method}`	results are computed by using the similarity function specified as parameter, which can be *Compound*, *CrossSim*, *Dependency*, *Readme*, *RepoPalCompound*, *RepoPalCompoundV2*, *OSCAN* or *CrossRecSimilarity*.
`{cluster_algo}`	results are computed by using the clustering algorithm specified as parameter, which can be *CLARA*, *K-Medoids* or *HCLibrary*.

Example: `cluster/CrossSim/Clara` is a call example of this API method and the result is shown below.

```
[
{
"clusterization": {
"clusterizationDate": 1544376657311,
"similarityMethod": "CrossSim",
"clusterAlgorithm": "Clara",
"id": "5c0d5151fe03927a05d2055a"
},
"mostRepresentative": {
"id": "5c0d5150fe03927a05d203ce",
"name": "AlipayOrdersSupervisor-GUI",
"description": "GUI of AlipayOrdersSupervisor, implemented in Java and Swing",
```

[16] http://projects.spring.io/spring-data/
[17] http://projects.spring.io/spring-data-mongodb/
[18] https://github.com/crossminer/scava/tree/dev/knowledge-base

```
"fullName": "thundernet8/AlipayOrdersSupervisor-GUI",
"year": 0,
"active": true,
"html_url": "https://github.com/thundernet8/AlipayOrdersSupervisor-GUI",
"clone_url": "https://github.com/thundernet8/AlipayOrdersSupervisor-GUI.git",
"git_url": "git://github.com/thundernet8/AlipayOrdersSupervisor-GUI.git",
"size": 0,
"master_branch": "master",
"readmeText": "...",
"dependencies": [
"com.darcula:darcula-lnf",
...
],
"starred": [
{
"login": "thundernet8",
"datestamp": "2017-08-16T17:39:24Z"
},
...
]
},
"artifacts": [
{
"id": "5c0d5150fe03927a05d203d2",
"name": "RssToMobiService",
"description": "A rss to mobi service",
"fullName": "yanghua/RssToMobiService",
"year": 0,
"active": true,
"html_url": "https://github.com/yanghua/RssToMobiService",
"clone_url": "https://github.com/yanghua/RssToMobiService.git",
"git_url": "git://github.com/yanghua/RssToMobiService.git",
"master_branch": "master",
"readmeText": "...",
"dependencies": [
...
],
"starred": [
...
]
},
{
...
},
{
"clusterization": {
...
}
]
```


6.2 Get cluster containing a particular project

Description: This resource is used to retrieve clusters of projects.

Example: `cluster/CrossSim/Clara/5c0d5150fe03927a05d203d2` is a call example of this API method and the result is shown below.


```
{
"clusterization": {
"clusterizationDate": 1544376657311,
"similarityMethod": "CrossSim",
```

HTTP Method	HTTP url
GET	`cluster/{sim_method}/{cluster_algo}/{artifact_id}`
Path parameters	
Name	**Description**
`{sim_method}`	results are computed by using the similarity function specified as parameter, which can be *Compound*, *CrossSim*, *Dependency*, *Readme*, *RepoPalCompound*, *RepoPalCompoundV2*, *OSCAN* or *CrossRecSimilarity*.
`{cluster_algo}`	results are computed by using the clustering algorithm specified as parameter, which can be *CLARA*, *K-Medoids* or *HCLibrary*.
`{artifact_id}`	The id of the artifact.


```
"clusterAlgorithm": "Clara",
"id": "5c0d5151fe03927a05d2055a"
},
"mostRepresentative": {
"id": "5c0d5150fe03927a05d203ce",
"name": "AlipayOrdersSupervisor-GUI",
"description": "GUI of AlipayOrdersSupervisor, implemented in Java and Swing",
"fullName": "thundernet8/AlipayOrdersSupervisor-GUI",
"year": 0,
"active": true,
"html_url": "https://github.com/thundernet8/AlipayOrdersSupervisor-GUI",
"clone_url": "https://github.com/thundernet8/AlipayOrdersSupervisor-GUI.git",
"git_url": "git://github.com/thundernet8/AlipayOrdersSupervisor-GUI.git",
"size": 0,
"master_branch": "master",
"readmeText": "...",
"dependencies": [
"com.darcula:darcula-lnf",
...
],
"starred": [
{
"login": "thundernet8",
"datestamp": "2017-08-16T17:39:24Z"
},
...
]
},
"artifacts": [
{
"id": "5c0d5150fe03927a05d203d2",
"name": "RssToMobiService",
"description": "A rss to mobi service",
"fullName": "yanghua/RssToMobiService",
"year": 0,
"active": true,
"html_url": "https://github.com/yanghua/RssToMobiService",
"clone_url": "https://github.com/yanghua/RssToMobiService.git",
"git_url": "git://github.com/yanghua/RssToMobiService.git",
"master_branch": "master",
"readmeText": "...",
"dependencies": [
...
],
"starred": [
...
]
},
...
```

```
54 ]
55 }
```

7 Conclusions

In this deliverable, we presented the results related to the unsupervised tool of OSS projects. By reviewing some of the most notable studies concerning software similarities and clusterization, we were able to summarize the main features and requirements of a software clustering tool. Based on the observation, we came up with two independent tools, i.e., unsupervised and supervised clustering. Firstly, we exploited CROSSSIM which has been designed and implemented in D6.2 and D6.3 as the software similarity tool to compute similarities among OSS projects and feed as input of the clustering process. We made use of two algorithms, i.e., K-Medoids and CLARA as the clustering engine. The algorithms have been chosen since they meet the requirements concerning effectiveness and efficiency in clustering OSS projects. An evaluation has been conducted on a dataset with 570 projects collected from GITHUB and shows that both K-Medoids and CLARA generate relevant clusters compared to predefined ground-truth data.

Secondly, apart from the unsupervised tool, we also implemented OSCAN, a supervised neural network that groups a set of input vectors into clusters. Based on the observation that the prescribed information related to projects and the corresponding categories reflects humans' perception towards the relationship between projects and categories. We hypothesize that projects' features, such as source code, metadata can be exploited to automatically group projects into categories. We attempt to cluster OSS projects by simulating humans' cognition towards the projects-categories relationship, using the data manually given by developers. OSCAN is highly advantageous given that labeled data is available for the training process. This is true once data is collected from dedicated sources such as GITHUB and SourceForge where projects have been manually classified by humans. A preliminary evaluation on a Sharejar dataset [40] of 745 demonstrates that OSCAN obtains good clustering performance with respect to various quality metrics, i.e., Precision, Recall, Entropy, and Purity. Moreover, we anticipate another application of OSCAN to solve the problem of recommending API migrations.

7.1 Fulfilling the related CROSSMINER Requirements

We analyze the possibility of the recommendation functionalities implemented in this deliverable together with similarity computations done in Deliverable D6.2 to support related requirements defined in the CROSSMINER project.

Table 6 lists all the CROSSMINER use case requirements that are related to the clustering functionality in this deliverable. The status of the requirements is also specified in Table 7 and Table 8 with the aim of distinguishing requirements that are already satisfied (marked with ●) by means of the developed tools from those that are not supported yet (marked with ○) or that are partially supported (marked with ◐).

- U148: The clustering tools presented in the previous section can be used to group libraries used in projects. The outcome can be used to suggest to developers stacks of open source applications.
- U149: In this deliverable, CROSSSIM has been exploited to compute similarities and to provide input for the clustering module. Currently, we can use different source code artifact as features, e.g., method names, function names.

7.2 Future Work

In the last deliverable of WP6, i.e., D6.5 which is due on month M30, we are going to provide the final implementation of the Knowledge Base designed in Task 6.1 including the APIs designed in Task 6.3 in Deliverable

Name	Status
GetProjectAlternativesWithSimilarAPIs	●
GetProjectAlternativesWithSimilarSize	○
GetProjectAlternativesWithSimilarTopics	◐
GetProjectAlternativesWithSimilarQuality	○
GetProjectsByUsedComponents	◐
GetAPIUsageDiscussions	●
GetAPIUsagePatterns	●
GetRecommendedDeps	●
GetRecommendedDocs	◐
GetAPIBreakingUpdates	○
GetRequiredChanges	○

Table 6: Implementation status of the required recommendations as presented in D6.1. Early stage: ○; Half done: ◐; Fully done: ●

Req. No.	Requirement	Priority	Status
D65	The cross-project relationships miners shall be able to create cluster of projects with respect to an extensible set of project similarity functions	SHALL	●

Table 7: Work Package 6 requirement in the focus of this deliverable (from D1.1).

Req. No.	Requirement	Priority	Status
U148	Able to propose common clusters of collaborating libraries of projects to suggest to developers stacks of open source applications	SHALL	◐
U149	Able to categorize Code Snippets based on the project involved	SHALL	●

Table 8: Use Case Requirements referring Work Package 6 and in the focus of this deliverable (from D1.1).

D6.5: *The CROSSMINER Knowledge Base - Final Version*. In particular, we are going to address all the mandatory recommendations as specified in Deliverable D1.1. A summary of the current implementation status of the required recommendations is depicted in Table 6. There, four types of recommendations are fully done, three of them are half done and four of them are early stage and planned to be completed at M30. The remaining requirements which are not fully implemented, i.e., those denoted either with ◐ or ○ in Table 6, will be fully addressed in D6.5. Among others, we put more emphasis on the problem of API migration and detecting API breaking updates. Furthermore, we will finalize the StackOverflow post recommendation which has been partly solved in D6.3.

References

[1] Rock: A robust clustering algorithm for categorical attributes. In *Proceedings of the 15th International Conference on Data Engineering*, ICDE '99, pages 512–, Washington, DC, USA, 1999. IEEE Computer Society.

[2] Charu C. Aggarwal and ChengXiang Zhai. A survey of text clustering algorithms. In Charu C. Aggarwal and ChengXiang Zhai, editors, *Mining Text Data*, pages 77–128. Springer, 2012.

[3] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, January 1991.

[4] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang. Clustering mobile apps based on mined textual features. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 38:1–38:10, New York, NY, USA, 2016. ACM.

[5] Kamel Alreshedy, Dhanush Dharmaretnam, Daniel M. German, Venkatesh Srinivasan, and T. Aaron Gulliver. Scc: Automatic classification of code snippets. *CoRR*, abs/1809.07945, 2018.

[6] Pavel Berkhin. A survey of clustering data mining techniques. In Jacob Kogan, Charles Nicholas, and Marc Teboulle, editors, *Grouping Multidimensional Data*, pages 25–71. Springer, 2006.

[7] 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.

[8] G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds, and D. B. Rosen. Fuzzy artmap: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3(5):698–713, Sept 1992.

[9] Gail A. Carpenter and Stephen Grossberg. The handbook of brain theory and neural networks. chapter Adaptive Resonance Theory (ART), pages 79–82. MIT Press, Cambridge, MA, USA, 1998.

[10] 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.

[11] William W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*, ICML'95, pages 115–123, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[12] Ana Emília Victor Barbosa Coutinho, Emanuela Gadelha Cartaxo, and Patrícia Duarte de Lima Machado. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal*, 24:407–445, 2014.

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

[14] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. Linked open data to support content-based recommender systems. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 1–8, New York, NY, USA, 2012. ACM.

[15] J. Escobar-Avila, M. Linares-Vásquez, and S. Haiduc. Unsupervised software categorization using bytecode. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 229–239, May 2015.

[16] William S. Evans, Christopher W. Fraser, and Fei Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, Dec 2009.

[17] Jaroslav Fowkes and Charles Sutton. PAM: Probabilistic API Miner. `https://github.com/mast-group/api-mining`. last access 24.08.2018.

[18] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 254–265, New York, NY, USA, 2016. ACM.

[19] Eibe Frank and Remco R. Bouckaert. Naive bayes for text classification with unbalanced classes. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Knowledge Discovery in Databases: PKDD 2006*, pages 503–510, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[20] 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.

[21] Stephen Grossberg. Adaptive resonance theory: How a brain learns to consciously attend, learn, and recognize a changing world. *Neural Netw.*, 37:1–47, January 2013.

[22] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. *SIGMOD Rec.*, 27(2):73–84, June 1998.

[23] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111, June 2012.

[24] Marti A. Hearst. Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28, July 1998.

[25] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. *CoRR*, abs/1511.06939, 2015.

[26] A. Huang. Similarity measures for text document clustering. pages 49–56, 2008.

[27] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

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

[29] Alexandros Karatzoglou and Balázs Hidasi. Deep learning for recommender systems. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*, RecSys '17, pages 396–397, New York, NY, USA, 2017. ACM.

[30] George Karypis, Eui-Hong (Sam) Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, August 1999.

[31] L Kaufman and Peter Rousseeuw. Clustering by means of medoids. In *Statistical Data Analysis Based on the L1 Norm and Related Methods*, pages 405–416. North-Holland; Amsterdam, 1987.

[32] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: an introduction to cluster analysis*. Wiley, 1990.

[33] Leonard Kaufman and Peter J. Rousseeuw. *Finding groups in data : an introduction to cluster analysis*. Wiley, New York, 1990.

[34] Saif Ur Rehman Khan, Sai Peck Lee, Raja Wasim Ahmad, Adnan Akhunzada, and Victor Chang. A survey on test suite reduction frameworks and tools. *Int. J. Inf. Manag.*, 36(6):963–975, December 2016.

[35] Ashraf M. Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In Geoffrey I. Webb and Xinghuo Yu, editors, *AI 2004: Advances in Artificial Intelligence*, pages 488–499, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[36] Thomas K Landauer. *Latent semantic analysis*. Wiley Online Library, 2006.

[37] Alexander LeClair, Zachary Eberhart, and Collin McMillan. Adapting neural text classification for improved software categorization. *CoRR*, abs/1806.01742, 2018.

[38] António Menezes Leitão. Detection of redundant code using r2d2. *Software Quality Journal*, 12(4):361–382, Dec 2004.

[39] 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.

[40] Mario Linares-Vásquez, Collin Mcmillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empirical Softw. Engg.*, 19(3):582–618, June 2014.

[41] 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.

[42] 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.

[43] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software*, 73(2):227 – 244, 2004. Applications of statistics in software engineering.

[44] Oded Maimon and Lior Rokach, editors. *Clustering Methods*, pages 321–352. Springer US, Boston, MA, 2005.

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

[46] 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.

[47] Collin McMillan, Mario Linares-Vasquez, Denys Poshyvanyk, and Mark Grechanik. Categorizing software applications for maintenance. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 343–352, Washington, DC, USA, 2011. IEEE Computer Society.

[48] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25, New York, NY, USA, 2010. ACM.

[49] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[50] J. Misra, K. M. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus. Software clustering: Unifying syntactic and semantic features. In *2012 19th Working Conference on Reverse Engineering*, pages 113–122, Oct 2012.

[51] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, Piscataway, NJ, USA, 2015. IEEE Press.

[52] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: An overview. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2:86–97, 01 2012.

[53] Raymond T. Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1003–1016, September 2002.

[54] Anh Tuan Nguyen and Tien N. Nguyen. Automatic categorization with deep neural network for open-source java projects. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 164–166, Piscataway, NJ, USA, 2017. IEEE Press.

[55] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio. CrossSim: Exploiting Mutual Relationships to Detect Similar OSS Projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 388–395, Aug 2018.

[56] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. Knowledge-aware recommender system for software development. In *Proceedings of the 1st Workshop on Knowledge-aware and Conversational Recommender System*, KaRS 2018, New York, NY, USA, 2018. ACM.

[57] 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, Rome, Italy, May, 28-30, 2018.*, 2018.

[58] 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.

[59] 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.

[60] P.T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns - to appear. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019.

[61] 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.

[62] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 522–531, Piscataway, NJ, USA, 2013. IEEE Press.

[63] 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 the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111, New York, NY, USA, 2014. ACM.

[64] Farhad Pourpanah, Chee Peng Lim, and Junita Mohamad Saleh. A hybrid model of fuzzy artmap and genetic algorithm for data classification and rule extraction. *Expert Syst. Appl.*, 49(C):74–85, May 2016.

[65] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[66] C. Ragkhitwetsagul, J. Krinke, and B. Marnette. A picture is worth a thousand words: Code clone detection based on image similarity. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, pages 44–50, March 2018.

[67] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519, Aug 2018.

[68] W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[69] Eréndira Rendón, Itzel Abundez, Alejandra Arizmendi, and Elvia M. Quiroz. Internal versus external cluster validation indexes. *Int. Journal of Compt. and Comm.*, 5:27–34, March 2011.

[70] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[71] 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.

[72] 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.

[73] Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. Cataloging github repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pages 314–319, New York, NY, USA, 2017. ACM.

[74] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, Jan 2004.

[75] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. In *6th ACM SIGKDD, World Text Mining Conference*, 2000.

[76] Yee Whye Teh, Michael I. Jordan, Matthew J. Beal, and David M. Blei. Hierarchical dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[77] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191, Oct 2013.

[78] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal*, 19(2):295–331, Jun 2011.

[79] Mohammad-Taghi Vakil-Baghmisheh and Nikola Pavešić. A fast simplified fuzzy artmap network. *Neural Processing Letters*, 17(3):273–316, Jun 2003.

[80] S. Vargas-Baldrich, M. Linares-Vásquez, and D. Poshyvanyk. Automated tagging of software projects using bytecode and dependencies (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 289–294, Nov 2015.

[81] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006*, 2006.

[82] 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.

[83] Tzu-Tsung Wong. Performance Evaluation of Classification Algorithms by K-fold and Leave-one-out Cross Validation. *Pattern Recognition*, 48(9):2839–2846, 2015.

[84] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, Jun 2015.

[85] Harry Zhang. The optimality of naive bayes. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*. AAAI Press, 2004.

[86] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Min. Knowl. Discov.*, 1(2):141–182, January 1997.

[87] Yongquan Zhang, Hongbing Ji, and Wenbo Zhang. TPPFAM: Use of threshold and posterior probability for category reduction in fuzzy ARTMAP. *Neurocomputing*, 124:63 – 71, 2014.

[88] 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.

[89] Ying Zhao, George Karypis, and Usama Fayyad. Hierarchical clustering algorithms for document datasets. *Data Min. Knowl. Discov.*, 10:141–168, March 2005.