



Project Number 732223

D6.5 The CROSSMINER Knowledge Base - Final Version

**Version 1.0
28 June 2019
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

Every effort has been made to ensure that all statements and information contained herein are accurate, however the CROSSMINER Project Partners accept no liability for any error or omission in the same.

© 2019 Copyright in this document remains vested in the CROSSMINER Project Partners.

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@aub.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@cw.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, 5 th 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 January 2019
0.2	First draft	17 March 2019
0.8	Second draft	12 June 2019
0.9	First release	24 June 2019
1.0	Final release	28 June 2019

Table of Contents

1	Introduction	1
1.1	Summary	1
1.2	Document Structure	2
2	Literature Review	4
2.1	Software Similarity	4
2.2	API Usage Recommendations	10
2.3	Library Recommendations	12
2.4	Mining StackOverflow to support software development	13
2.5	Classification of StackOverflow posts	14
2.6	Neural Networks in Software Engineering	14
3	The CROSSMINER Recommender Systems	16
4	Recommendation of project alternatives with similar APIs	18
4.1	Proposed Approach	18
4.2	Evaluation	19
5	API function calls and usage patterns recommendation	20
5.1	Overview	20
5.2	Architecture	21
5.3	Data Representation	22
5.4	Similarity Computation	23
5.5	API function calls recommendation	25
5.6	API usage patterns recommendation	26
5.7	Evaluation	27
5.7.1	Datasets	27
5.7.2	Methodology	27
5.8	Result Analysis	29
6	Third-party libraries recommendation	31
6.1	Overview	31
6.2	Architecture	32
6.3	Data Encoder	33
6.4	Similarity Calculator	33

6.5	Recommendation Engine	33
6.6	Evaluation	35
6.6.1	Dataset	35
6.6.2	Evaluation metrics	36
6.6.3	Evaluation Methodology	39
6.7	Result Analysis	40
6.8	Threats to Validity	46
6.9	Discussions	47
6.10	Conclusions and Future Work	47
7	Recommendation of StackOverflow Posts	49
7.1	Overview	49
7.2	Background and Motivations	50
7.3	Proposed Approach	51
7.3.1	Index Creation	52
7.3.2	Query Creation	54
7.3.3	Query Execution	56
7.4	Evaluation	56
7.4.1	Dataset	58
7.4.2	User studies	58
7.4.3	Evaluation metrics	58
7.4.4	Research questions	59
7.5	Experimental Results	59
7.6	Threats to validity	62
8	Categorization of Relevant API Discussions	63
8.1	Feed-forward Neural Networks	64
8.2	System Architecture	66
8.3	Evaluation	67
8.3.1	Datasets	67
8.3.2	Evaluation Metrics	67
8.4	Results	68
8.5	Threats to validity	68

9 Mining API Migration Patterns	70
9.1 Use Case	70
9.2 Proposed Approach	72
9.2.1 Architecture	72
9.2.2 Recommending relevant function calls and code snippets	73
9.3 Mining cross-project dependencies to discover API migration samples	74
9.3.1 AEThEReAL	74
9.3.2 Analysis Results	76
10 The Knowledge Base	79
10.1 Overview	80
10.2 Use Cases	80
10.3 Datasets	82
10.4 Technology Dependencies	82
10.5 REST API	83
10.5.1 Get analyzed projects	83
10.5.2 Get analyzed project by id	85
10.5.3 Get projects by metric provider platform id	86
10.5.4 Search analyzed projects	86
10.5.5 Add a new GitHub project to the analyzed projects	87
10.5.6 Store developer activity metrics	88
10.5.7 Get alternatives projects	89
10.5.8 Get relevant StackOverflow posts	91
10.5.9 Get third-party libraries	91
10.5.10 Get API function calls	92
10.5.11 Get API usage patterns	94
10.5.12 Get clustered projects	95
10.5.13 Get cluster containing a particular project	96
10.5.14 Get migration client pairs examples	97
10.5.15 Get clients using a particular library version	98
10.5.16 Get StackOverflow posts related to discussions about API migration	98
10.5.17 Get impact on library evolution	99
10.5.18 Get useful code snippets to migrate towards a new library version	100
11 Conclusions	102

Executive Summary

Implementing a new system by mining open source software (OSS) repositories helps reduce development effort and concurrently increase productivity. Furthermore, as the OSS ecosystem facilitates vibrant expert and user communities, developers can get practical supports which allow them to fix bugs as well as to find probable solutions to various issues alongside the development cycle. Nevertheless, to help developers effectively mine the existing data, it is crucial to equip them with suitable machineries. Under the context of Work Package 6 (WP6), we have been developing recommender systems to assist developers in building their software by mining OSS forges.

In this deliverable, we present the final implementation of the Knowledge Base designed in Task 6.1. We address the mandatory recommendations as specified in Deliverable D1.1. By exploiting the graph representation as well as recommendation engines developed in the previous phases of WP6, we are able to design and realize important recommendation engines which are capable of providing developers with useful supports while they are programming. In particular, we implement and integrate into the Knowledge Base the following types of recommendations: API function calls and usage patterns, third-party libraries, API migration, and StackOverflow post classification. Furthermore, we also finalize the tool for recommending StackOverflow posts.

1 Introduction

Open source software (OSS) allows developers to study, change, and improve the code free of charge. Code reusing is an intrinsic feature of OSS, and developing a new system by leveraging existing open source components reduces development effort, and thus being beneficial to the software life cycle. The benefits resulting from the reuse of properly selected open source projects are manifold including the fact that the system being implemented relies on open source code, “*which is of higher quality than the custom-developed code’s first incarnation*” [134]. In addition to source code, also metadata available from different related sources, e.g., communication channels and bug tracking systems, can be beneficial to the development process if properly mined [112]. However, without being equipped with suitable machinery, given a plethora of data sources, developers would struggle to look for and approach the sources that meet their demand, in the hope of transforming them to practical knowledge. Under the circumstances, the problem is not the lack of information but instead an information overload coming from heterogeneous and rapidly evolving sources. In particular, when developers join a new project, they have to typically master a huge number of information sources [34] (often at a short time). In other words, “*we are drowning in information but starved for knowledge.*”¹

In this sense, the deployment of systems which exploit existing data to assist and improve developer experience is of paramount importance. The realization of techniques and tools to build such systems has attracted a lot of attention from the research community recently. 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. The introduction of recommender systems to the domain of software development brings substantial benefits. A recommender system in software engineering (RSSE) is defined as “*... a software application that provides information items estimated to be valuable for a software engineering task in a given context*” [121]. Among others, recommender systems assist developers in navigating large information spaces and getting instant recommendations that might be helpful to solve the particular development problem at hand [96, 113]. Thus, RSSEs aim at giving developers recommendations, which can consist of different items including code examples, issue reports, reusable source code, possible third-party components, documentation, etc. In the scope of Work Package 6, we dedicate ourselves to develop various recommender systems to support developers. 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. Section 1.2 brings in the structure of the whole deliverable.

1.1 Summary

Within Work Package 6, we exploit cutting-edge information retrieval techniques to build recommender systems for mining software repositories. We develop tools and techniques to assist software developers in implementing their projects by means of an advanced Eclipse-based IDE. The recommendation engines are fed with metadata curated from different OSS forges and communication channels. Based on the CROSSMINER mining tools and on previous feedback, developers are able to select open source software and get real-time recommendations which are summarized as follows.

- *find a set of similar OSS projects* to the system being developed, with respect to different criteria, e.g., external dependencies, or API usage;
- *recommend components* that similar projects have included, for instance, a list of external libraries [83];

¹*John Naisbitt, researcher of future studies*

- *recommend code snippets* that show how an API is used in practice. These snippets provide developers with a deeper insight into the usage of the APIs being included;
- *suggest additional sources of information*, e.g., technical documents, tutorials, communication channels, etc., that are relevant to the code being developed, for instance by mining external experiences from StackOverflow;
- *identify API changes and their consequences*: changes of libraries will have a certain effect on the depending projects. It is necessary to notify developers and recommend amendments to preserve program compatibility.

Our work package consists of a unified framework being built on top of a graph representation model [90] that allows us to *compute similarities* and *incorporate various recommendation techniques*. In Deliverable D6.1, we drafted the first version of the CROSSMINER Knowledge Base. Afterwards, in Deliverable D6.2, we transformed the relationships among non-human artifacts, e.g., API utilizations, source code, interactions, and humans, e.g., developers into a mathematically computable format. This representation allows for the creation of CROSSSIM [90], a versatile tool for computing similarities among OSS projects. Being based on the infrastructure, a recommender system named CROSSREC for providing software developers with third-party libraries has been built in Deliverable D6.3 [92]. In this setting, CROSSSIM performs its computation using third-party libraries as the input features. Furthermore, to provide API function calls and usage patterns recommendation, we developed a context-aware recommender system, i.e., FOCUS [91],[96] where the similarities among OSS projects are computed by means of CROSSSIM with API function calls as features.

In Deliverable D6.4, we presented two independent tools, i.e., unsupervised and supervised clustering. Firstly, we exploited CROSSSIM 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. Secondly, apart from the unsupervised tool, we implemented OSCAN, a supervised neural network that groups a set of input vectors into clusters. 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.

The current deliverable, i.e., D6.5 presents the final implementation of the Knowledge Base. We address the mandatory recommendations as specified in Deliverable D1.1, including recommendation of API function calls and usage patterns, as well as third-party libraries. Furthermore, we API migration and classification of StackOverflow posts. Furthermore, we finalize the tool for providing StackOverflow post recommendation which has been partially solved in D6.3.

1.2 Document Structure

Deliverable D6.5 is structured into the following sections:

- Section 2 presents a literature review on the related work;
- Section 3 brings an overview of the recommender systems conceived within Work Package 6;
- In Section 4, we present the proposed approach for recommending project alternative with similar APIs;
- Section 5 introduces FOCUS, a recommender system for mining API function calls and usage patterns
- Afterwards, in Section 6, a recommender system for providing developers with third-party libraries, i.e., CROSSREC is introduced;
- In Section 7, we implement and evaluate SOrec, a recommender system for providing relevant StackOverflow posts, given an input context;

- Section 8 introduces in detail SCORE, a supervised classifier for categorizing StackOverflow posts;
- Section 9 presents *amAdvisor*, a recommender system to deal with API breaking changes by suggesting relevant migration patterns;
- Section 10 provides an exhaustive documentation of the REST API for the Knowledge Base;
- Finally, Section 11 summarizes our work and concludes the deliverable.

2 Literature Review

In this section, we summarize related work and associate our contributions to the literature in mining open source software repositories, discussion channels, as well as the application of recommender systems to tackle various issues in Software Engineering. In particular, Section 2.1 provides a comprehensive review on software similarity. Afterwards, we introduce notable studies on the topic of API function calls and third-party library recommendations in Section 2.2 and in Section 2.3, respectively. Section 2.4 reviews studies related to mining and usage of StackOverflow posts. Section 2.5 gives a summary of studies concerning the classification of StackOverflow posts. Finally, Section 2.6 provides an overview of neural networks as well as their application in Software Engineering.

2.1 Software Similarity

Having access to similar software projects is beneficial to the development process. By looking at a similar OSS project, developers learn how relevant classes are implemented, and in some certain extent, to reuse useful source code [131],[155]. Also, recommender systems rely heavily on similarity metrics to suggest suitable and meaningful items for a given item [38],[100],[131],[138]. As a result, similarity computation among software and projects has attracted considerable interest from many research groups. In recent years, several approaches have been proposed to solve the problem of software similarity computation. Many of them deal with similarity for software systems, others are designed for computing similarities among open source software projects. Depending on the set of mined features, there are two main types of software similarity computation techniques [29]:

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

This classification is used throughout this paper as a means to distinguish existing approaches with regards to the input information used for similarity computation. In this section, we provide a summary on three techniques for computing similarity among open source projects, i.e., MUDABLU [42], CLAN [82], and REPOPAL [155].

Together with a tool for automatically categorizing open source repositories, Garg *et al.* [42] propose an approach for computing similarity between software projects using source code. A pre-processing stage is performed to extract identifiers such as variable names, function names, and to remove unrelated factors such as comment. With the application of Latent Semantic Analysis (LSA) [67], software is considered as a document and each identifier is considered as a word. LSA is used for extracting and representing the contextual usage meaning of words by statistical computations applied to a large corpus of text. In summary, MUDABLU works in the following steps to compute similarities between software systems:

- (i) Extracts identifiers from source code and removes unrelated content;
- (ii) Creates an identifier-software matrix with each row corresponds to one identifier and each column corresponds to a software system;
- (iii) Removes unimportant identifiers, i.e., those that are too rare or too popular;
- (iv) Performs LSA on the identifier-software matrix and computes similarity on the reduced matrix using cosine similarity.

MUDABLUE has been evaluated on a database consisting of software systems written in C. The outcomes of the evaluation were compared against two existing approaches, namely GURU [80], and the SVM based method by Ugurel *et al.* [142]. The evaluation shows that MUDABLUE outperforms these observed algorithms with respect to precision and recall.

McMillan *et al.* propose CLAN, an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to packages class hierarchies [82]. CLAN works based on the document framework for computing similarity, semantic anchors, e.g., those that define the documents' features. Semantic anchors and dependencies help obtain a more precise value for similarity computation between documents. The assumption is that if two applications have API calls implementing requirements described by the same abstraction, then the two applications are more similar than those that do not have common API calls. The approach uses API calls as semantic anchors to compute application similarity since API calls contain precisely defined semantics. The similarity between applications is computed by matching the semantics already expressed in the API calls.

Using a complete software application as input, CLAN represents source code files as a term-document matrix (TDM). A TDM is used to store the features of a set of document and it is a matrix where a row corresponds to a document and a column represents a term [30]. Each cell in the matrix is the frequency that the corresponding term appears in the document. By CLAN, a row contains a unique class or package and a column corresponds to an application. SVD is then applied to reduce the dimension of the matrix. Similarity between applications is computed as the cosine similarity between vector in the reduced matrix. CLAN has been tested on a dataset with more than 8,000 SourceForge² applications and shows that it qualifies for the detection of similar applications [82].

MUDABLUE and CLAN are comparable in the way they represent software and source code components like variables, function names or API calls in a term-document matrix and then apply LSA to find the similarity and to category the softwares. However, CLAN has been claimed to help obtain a higher precision than that of MUDABLUE as it considers only API calls to represent software systems. As shown later in this paper, CLAN is more efficient than MUDABLUE as it produces recommendations in a much shorter time.

In contrast to many previous studies that are generally based on source code [42],[76],[82], RepoPal [155] is a high-level similarity metric and takes only repositories metadata as its input. With this approach, two GitHub³ repositories are considered to be similar if: (i) They contain similar README.MD files; (ii) They are starred by users of similar interests; (iii) They are starred together by the same users within a short period of time. Thus, the similarities between GitHub repositories are computed by using three inputs: readme file, stars and the time gap that a user stars two repositories.

Considering two repositories r_i and r_j , the following notations are defined: (i) f_i and f_j are the readme files with t being the set of terms in the files; (ii) $U(r_i)$ and $U(r_j)$ are the set of users who starred r_i and r_j , respectively; and (iii) $R(u_k)$ is the set of repositories that user u_k already starred. There are three similarity indices as follows:

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

$$sim_f(r_i, r_j) = CosineSim(f_i, f_j) \quad (1)$$

²SourceForge: <https://sourceforge.net/>

³About GitHub: <https://github.com/about>

Stargazer-based similarity The similarity between a pair of users u_k and u_l is defined as the Jaccard index [57] of the sets of repositories that u_k and u_l have already starred: $sim_u(u_k, u_l) = Jaccard(R(u_k), R(u_l))$. The star-based similarity between two repositories r_i and r_j is the average similarity score of all pairs of users who already starred r_i and r_j :

$$sim_s(r_i, r_j) = \frac{1}{|U(r_i)| \cdot |U(r_j)|} \sum_{\substack{u_k \in U(r_i) \\ u_l \in U(r_j)}} sim_u(u_k, u_l) \quad (2)$$

Time-based similarity It is supposed that if a user stars two repositories during a relative short period of time, then the two repositories are considered to be similar. Based on this assumption, given that $T(u_k, r_i, r_j)$ is the time gap that user u_k stars repositories r_i and r_j , the time-based similarity is computed as follows:

$$sim_t(r_i, r_j) = \frac{1}{|U(r_i) \cap U(r_j)|} \sum_{u_k \in U(r_i) \cap U(r_j)} \frac{1}{|T(u_k, r_i, r_j)|} \quad (3)$$

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

$$sim(r_i, r_j) = sim_f(r_i, r_j) \times sim_s(r_i, r_j) \times sim_t(r_i, r_j) \quad (4)$$

REPOPAL has been evaluated against CLAN using a dataset of 1,000 Java repositories [155]. Among them, 50 were chosen as queries. *Success Rate*, *Confidence* and *Precision* were used as the evaluation metrics. Experimental results in the paper show that REPOPAL produces better quality metrics than those of CLAN.

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

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

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

To detect clone among Android apps, Wang *et al.* propose WuKong [145] which employs a two-phase process as follows. The first phase exploits the frequency of Android API calls to filter out external libraries. Afterwards, a fine-grained phase is performed to compare more features on the set of apps coming from the first phase. For each variable, its feature vector is formed by counting the number of occurrences of variables in different contexts (Counting Environments - CE). An m -dimensional Characteristic Vector (CV) is generated using m CEs, where the i -th dimension of the CV is the number of occurrences of the variable in the i -th CE. For each code segment, CVs for all variables are computed. A code segment is represented by an $n \times m$ Characteristic Matrix (CM). For each app, all code segments are modelled using CM, yielding a series of CMs and they are considered as the features for the app. The similarity between two apps is computed as the proportion of similar code segments. The similarity between two variables v_1 and v_2 is computed using cosine similarity [140],[141] between their feature vectors. Evaluations on more than 100,000 Android apps collected from 5 Chinese app markets show that the approach can effectively detect cloned apps [145]. CROSSSIM is also able to deal with low-level features as by WuKong if such features are integrated into the graph.

Lo *et al.* develop TagSim⁴, a tool that leverages tags to characterize applications and then to compute similarity between them [77]. Tags are terms that are used to highlight the most important characteristics of software systems [150] and therefore, they help users narrow down the search scope. TagSim can be used to detect similar applications written in different languages. Based on the assumption that tags capture better the intrinsic features of applications compared to textual descriptions, TagSim extracts tags attached to an application and computes their weights. This information forms the features of a given software system and can be used to distinguish it from others. The technique also differentiates between important tags and unimportant ones based on their frequency of appearance in the analyzed software systems. The more popular a tag across the applications is, the less important it is and vice versa. Each application is characterized by a feature vector, and each entry corresponds to the weight of a tag the application has. Eventually, the similarity between two applications is computed as the cosine similarity [140],[141] between the two vectors. To evaluate TagSim, more than a hundred thousands of projects have been collected and analyzed [77]. A total of 20 queries were used to study the performance of the algorithm in comparison with CLAN. The authors also performed a user study to manually analyze the extent to which two applications are similar. The experimental results show that TagSim helps achieve better performance in comparison to CLAN.

Inspired by CLAN, Linares-Vásquez *et al.* develop CLANdroid for detecting similar Android applications with the assumption that similar apps share some semantic anchors [73]. Nevertheless, in contrast to CLAN, CLANdroid works also when source code is not available as it exploits other high-level information. By extending the scope of semantic anchors for Android apps, starting from APK (Android Package) CLANdroid extracts quintuple features, i.e., identifiers, intents from source code, API calls and sensors from JAR files, and user permissions from the *AndroidManifest.xml*⁵ specification. This file is a mandatory component for any Android app and it contains important information about it. For each feature, a feature-application matrix is built, resulting in five different matrices. Latent Semantic Indexing is applied to all the matrices to reduce the dimensionality. Afterwards, similarity between a pair of applications is computed as the cosine similarity between their corresponding feature vectors from the matrix. Users can query for similar apps with a given app by specifying which feature is taken into consideration. Evaluations have been performed to study which semantic anchors are more effective [73]. The authors also analyze the impact of third-party libraries and obfuscated code when detecting similar apps, since these two factors have been shown to have significant impact on reuse in Android apps and experiments using APKs. The evaluation on a dataset shows that computing similarity based on API helps produce higher recall. According to the experimental results, the feature sensor is ineffective in computing similarity. By comparing with a ground-truth dataset collecting from Google Play, the study

⁴For the sake of clarity, in this deliverable we give a name for the algorithms that have not been originally named

⁵<https://developer.android.com/guide/topics/manifest/manifest-intro.html>

gives some hints on the mechanism behind the way Google Play recommends similar apps. CROSSSIM is relevant to CLANdroid since it can work with low-level features by representing function calls, API calls in the graph as we already demonstrated in our recent work [94],[96].

With the aim of finding apps with similar semantic requirements, SimApp has been developed to exploit high-level metadata collected from apps markets [29]. If two apps implement related semantic requirements then they are seen as similar. Each mobile application is modeled by a set of features, so called modalities. The following features are incorporated into similarity computation: *Name*, *Category*, *Developer*, *Description*, *Update*, *Permissions*, *Images*, *Content rating*, *Size* and *Reviews*. For each of these features, a function is derived for each of the features to calculate the similarity between applications. The final similarity score for a pair of apps is a linear combination of the multiple kernels with weights. Through the use of a set of training data, the optimal weights are determined by means of online learning techniques.

AnDarwin is an approach that applies Program Dependence Graphs to represent apps [33], and feature vectors are then clustered to find similar apps. Locality Sensitive Hashing is used to find approximate near-neighbors from a large number of vectors. AnDarwin works according to the following stages: (i) It represents each app as a set of vectors computed over the app's Program Dependence Graphs; (ii) Similar code segments are found by clustering all the vectors of all apps; (iii) It eliminates library code based on the frequency of the clusters; (iv) Finally, it detects apps that are similar, considering both full and partial app similarity. AnDarwin has been applied to find similar apps by different developers (cloned apps) and groups of apps by the same developer with high code reuse (rebranded apps). An evaluation using more than 200,000 apps from different Android markets demonstrated that the system can effectively detect cloned apps.

LibRec is a tool that provides developers them with library recommendations to help developers leverage existing libraries [138]. LibRec suggests the inclusion of libraries that may be useful for a given project using a combination of rule mining and collaborative filtering techniques. It finds a set of relevant libraries, based on the current set of libraries that a project already uses. *Association rule* mining is applied to find similar libraries that co-exist in many projects to extract libraries that are commonly used together. The component then rates each of the libraries based on their likelihood to appear together with the currently used libraries. A *collaborative filtering* technique is applied to search for top most similar projects and recommends libraries used by these projects to a given project. Given a project, similarity is computed against all projects and top similar projects are selected. The libraries used by the top similar projects are used as recommendations based on a score computed according to their popularity. Considering a set of projects and a set of libraries each project is characterized by a feature vector using the set of libraries it includes. The similarity between two projects is the cosine similarity between their feature vectors.

A summary of all the similarity metrics introduced in this section is depicted in Table 1. Most low-level similarity algorithms attempt to represent source code (and API calls) in a term-document matrix and then apply SVD to reduce dimensionality. The similarity is then computed as the cosine similarity between feature vectors. Among others, MUDABLU, CLAN, and CLANdroid belong to this category. In contrast, high-level similarity techniques do not consider source code for similarity computation. They characterize software by exploiting available features such as descriptions, user reviews, and *README.MD* file. The similarity is computed as the cosine similarity of the corresponding feature vectors. For computing similarity between mobile applications, other specific features such as images and permissions are also incorporated. A current trend in these techniques is to exploit textual content to compute similarity, e.g., in AppRec [16], SimApp [29], TagSim [77]. A main drawback with this approach is that, same words can be used to explain different requirements or the other way around, the same requirements can be described using different words [42]. So it might be the case that two textual contents with different vocabularies still have a similar description or two files with similar vocabularies contain different descriptions.

	Source code				Metadata					
	MUDABlue [42]	CLAN [82]	CLANdroid [73]	WuKong [145]	SimApp [29]	AnDarwin [33]	TagSim [77]	LibRec [138]	RepoPal [155]	
Considered features										Descriptions
Dependencies	-	-	-	✓	-	-	-	✓	-	Set of third-party libraries that a project includes
API Calls	✓	✓	✓	✓	-	✓	-	-	-	API function calls that appear in the source code of the analyzed projects. They are used to build term-document matrices and then to calculate similarities among applications
Functions	✓	-	-	-	-	✓	-	-	-	Functions defined in a project's source code
Stars	-	-	-	-	-	-	-	-	✓	The GitHub star events occurred for each analyzed projects
Timestamps	-	-	-	-	-	-	-	-	✓	The point of time when a user stars a repository
Statements	✓	-	-	-	-	-	-	-	-	Source code statements
Identifiers	✓	-	✓	✓	-	-	-	-	-	All artifacts related to source code, such as variable names, function names, 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 <i>README.MD</i> files, used to describe the functionalities of an open source project
Tags	-	-	-	-	-	-	✓	-	-	The tags that are used by OSS platforms, e.g. SourceForge to classify and characterize an OSS project
Updates	-	-	-	-	✓	-	-	-	-	The newest changes made to the considered applications
Permissions	-	-	✓	-	✓	-	-	-	-	This feature is available by mobile apps. It specifies the permission of an app to handle data in a smartphone
Screenshots	-	-	-	-	✓	-	-	-	-	This feature is available by mobile apps. It is a picture representing an app
Contents	-	-	-	-	✓	-	-	-	-	Each app has content rating to describe its content and age appropriateness
Size	-	-	-	-	✓	-	-	-	-	Some similarity metrics assume that two apps whose size is considerably different cannot be similar
Reviews	-	-	-	-	✓	-	-	-	-	All user reviews for an app are combined in a document
Intents	-	-	✓	-	-	-	-	-	-	For a mobile app, an intent is description for an operation to be performed
Sensors	-	-	✓	-	-	-	-	-	-	In mobile devices, sensors can provide raw data to monitor 3-D device movement or positioning, or changes in the environment. A set of features can be built from sensors to characterize an app
Used techniques										
TDM & LSA	✓	✓	✓	-	-	-	-	-	-	TDM [30] and LSA [68] are generally used in combination to model the relationships between API calls/identifiers and software systems and to compute the similarities between them
COS	✓	✓	✓	✓	✓	-	✓	✓	✓	Cosine similarity [140],[141] is widely used in several algorithms for computing similarities among vectors
JCS	-	-	-	-	-	✓	-	-	✓	Jaccard index [57] is used for computing similarity between two sets of elements

Table 1: Summary of the similarity algorithms and their features.

The matching of words in the descriptions as well as source code to compute similarity is considered to be ineffective as already stated in [82]. To overcome this problem, the application of a synonym dictionary like WordNet [84] is beneficial. Nevertheless, there is an issue with the approaches like REPOPAL where readme files are used for similarity computation. Since in general the descriptions for software projects are written in different languages, the comparison of readme files in different languages should yield dissimilarity, even though two projects may be similar. SimApp [29] is the only technique that attempts to combine several high-level information into similarity computation. It eventually applies a machine learning algorithm to learn optimal weights. The approach is promising, nevertheless it is only applicable in the presence of a decent training dataset, which is hard to come by in practice.

2.2 API Usage Recommendations

MAPO has been developed to mine API usage patterns from client code projects [157]. The system analyzes source files to collect API usage information and groups the API methods into clusters. Afterwards, it mines API usage patterns from the clusters, ranks them according to the similarity with developer context, and eventually recommends complete API code snippets to developers.

Moreno et al. introduce MUSE, a practical tool to recommend code examples related to a specific function [87]. MUSE parses source code to extract method usage, it simplifies examples and detects clones to group similar code snippets. Furthermore, it is able to rank recommendation outcomes according to various characteristics, i.e., reusability, understandability, and popularity.

Wang et al. proposed UP-Miner, aiming at reducing redundancy as well as covering a wide range of API usage patterns from source code [146]. From an input API method, the technique automatically finds all usage patterns and returns related code snippets.

Strathcona [53] is a recommendation tool, which analyzes developer's context from the structural point of view and suggests a possible implementation related to the task that she is developing. Strathcona uses six heuristics based on inheritance hierarchy, field types method calls, and object usage in order to build the query. The built query is then executed on a repository containing all possible usage of the APIs and it is built automatically from the context. Finally, Strathcona retrieves code examples, which can be navigated by the developer both graphically and in a textual way.

Fowkes et al. introduce PAM (Probabilistic API Miner), a parameter-free probabilistic approach to mine API usage patterns [41]. PAM uses the structural Expectation-Maximization (EM) algorithm to infer the most probable API patterns from client code, which are then ranked according to their probability. PAM outperforms both MAPO and UP-Miner (lower redundancy and higher precision).

The NCBUP-miner (Non Client-based Usage Patterns) [126] is a technique that identifies unordered API usage patterns from the API source code, based on both structural (methods that modify the same object) and semantic (methods that have the same vocabulary) relations. The same authors also propose MLUP [125], which is based on vector representation and clustering, but in this case client code is also considered. MLUP is used for mining multi-level API usage patterns, which are clusters of methods that co-exist in a method performing a specific functionality. As input, the technique takes the source code and extracts the relevant methods of the considered API. Each API public method is characterized by a vector, where each entry corresponds to a client method. Clustering techniques are then used to group API methods that are usually used together by projects.

Table 2 provides a summary on different approaches for mining API usage and their corresponding characteristics.

Name/Authors	Similarity Computation	Clustering	Dataset	Output
MAPO [157]	Class name, Method name, API call sequences	Data-driven	Java projects	API usage patterns
UP-Miner [146]	SeqSim technique similar sequences	Two clustering on frequent sequences with BIDE [147]	C# projects	Probabilistic graph of API
CLAMS [60]	Longest Common Subsequence, Distance matrix	LCS [51], HDBSCAN [25] techniques AP-TED for top rank snippets	Client projects from 6 popular Java libraries	API usage patterns
APIRec[88]	Association-based model with fine-grained code changes	Association-based change inference model	50 GitHub Java projects	Most frequent API calls
APIMiner extension[86]	Structural similarity among API	FP-growth with WEKA tool [49]	Android projects	Enhanced documentations
Niu et. al[102]	Object usages social network with co-existence relation	Co-existence relation with modularity index method call similarity with Gamma index	Android project	API usage patterns
CodeBroker[151]	Latent semantic analysis [67]	Discourse model, user model	Java core libraries	Three layered information relevant tasks, signature and JavaDoc

Table 2: Summary of API usage recommendation techniques.

Niu et al. extract API usage patterns using API class or method names as queries [103]. They rely on the concept of object usage (method invocations on a given API class) to extract patterns. The approach outperforms UP-Miner and Codota,⁶ a commercial recommendation engine in terms of coverage, performance, and ranking relevance.

Sourcerer [75] is a code search tool mainly based on Lucene. Source codes are indexed according to the included keywords and by considering fingerprints, which summarize code snippets in vectors and give information about the syntactical aspects of the code. Sourcerer maps also the developers of each snippet in a matrix consisting of developer-document entries. This kind of process gives further information about the developers who write the code: in particular, Sourcerer can categorize developers with best skills according to their contributions.

The authors in [78] introduce CodeHow, a code search engine specifically conceived to parse APIs online documentation by analyzing the user's query. As the first step, the tool retrieves and parses information coming from the documentation by applying the most classical text preprocessing techniques, i.e., text normalization, stop words removal and stemming. Then, CodeHow finds similarities between the user's query and the API related to it. To identify the relevant APIs among all retrieved ones, the authors propose the adoption of an information retrieval technique, called the Extended Boolean Model (EBM), and they make use of Elasticsearch as the main indexing and searching platform.

We developed FOCUS [96] a recommender system for mining API calls and usage patterns. The system represents mutual relationships among projects using a tensor and exploits a collaborative-filtering technique to mine API calls [129]. The main advantage of the system is that it does not depend on any specific set of libraries to generate API calls. Furthermore, FOCUS scales well with large datasets exploiting the context-aware collaborative-filtering technique that helps effectively remove irrelevant API function calls.

2.3 Library Recommendations

Teyton et al. introduce LibTic, an approach to identify relevant experts of Java libraries among GitHub developers by automatically mining software repositories [136]. Thus, LibTic can also be used to identify experts that can be contacted to ask questions concerning the usage of libraries.

LibFinder is an approach developed by *Ouni et al.* and it is based on a multi-objective search-based algorithm for supporting developers in searching for "useful" libraries when they implement a new software system [105]. The authors suggest that embedding library semantics in third-party library recommendation can increase efficiency [105].

The problem of third-party library recommendation has been well defined by *Thung et al.* [138]. In this work, LibRec was proposed to help developers discover existing libraries that may be useful for a given project using a combination of rule mining and collaborative filtering techniques [131]. Considering a set of projects $P = (p_1, p_2, \dots, p_m)$ and a set of libraries $L = (l_1, l_2, \dots, l_n)$, i.e. $\vec{p}_i = (I_i(l_1), I_i(l_2), \dots, I_i(l_n))$, where $I_i(l_r)$ is the inclusion of library l_r in project p_i . $I_i(l_r) = 1$ if l_r is used in p_i , otherwise $I_i(l_r) = 0$, the similarity between two projects is computed as the cosine of the angle between the two vectors. Given a project, similarity is computed against all projects and the most similar projects are selected. A performance evaluation was conducted on 500 GitHub projects to validate the proposed hypotheses. LibRec has been considered as baseline for evaluating the performance of the new approach that we have conceived in CROSSMINER for recommending libraries as presented in Sec. 6.

⁶<https://www.codota.com/>

2.4 Mining StackOverflow to support software development

StackOverflow can be exploited to support coding activities by providing developers with messages and code snippets therein that are relevant to the query explicitly or implicitly defined by the user. In this deliverable, we develop SOrec, a tool for finding StackOverflow posts that match with an input code snippet. SOrec imposes various measures on both the data collection and query phases. Furthermore, to improve efficiency Apache Lucene [1], an information retrieval library is used to index the textual content and code coming from StackOverflow. This section reviews related work in mining StackOverflow and associate these studies with SOrec.

Zagalsky *et al.* [152] present *Example overflow*, which allows developers to search for code snippets starting from provided keywords, which in turn are used by the system for retrieving code snippets from a local SO dump. Similarly to our approach, the search function is based on Apache Lucene even though the outcome of *Example overflow* consists of embeddable code, whereas SOrec is able to retrieve full posts that are related to the user context.

Ponzanelli *et al.* [111] propose *Seahawk*, a tool able to retrieve SO discussions, which are linked to the source code being developed. The search mechanism exploits code similarity techniques essentially based on TD-IDF. The SOrec search mechanisms are instead based on different boosting features that are considered when creating queries and when executing them atop of Apache Lucene.

Prompter [113] has been devised to extend Seahawk by focusing on the query creation phase and on the model, which is used to rank the retrieved posts. The tool relies on public search engines i.e., Google and Bing to retrieve SO messages. Further than focusing on the query creation phase, SOrec identifies also different aspects that are used to properly create data indexes and to accordingly define the queries by exploiting boosting mechanisms provided by Apache Lucene.

An approach to recommend a ranked list of pairs of SO questions and answers based on the user query consisting of a list of terms has been proposed by *de Souza et al.* [36]. Furthermore, the approach also allows one to classify SO posts according to defined labels like *how-to*, *debug corrective*, etc. The main difference with SOrec relies on the way queries are defined. In particular, in SOrec queries consist of the whole developer context, instead of only lists of terms explicitly defined by the user as in [36].

Rigby and Robilliard [117] propose ACE (Automated Code Extractor) that mines an input SO dump in order to find relevant elements in the code. ACE relies on a fully text-based analysis mechanism to identify and create indexes of the so-called salient element in the code. Differently to SOrec, ACE uses island parsers based on a set of regular expressions to approximate Java qualified statements (i.e., package definitions, class names, and so on).

StackOverflow data has been analyzed also with the aim of assessing documentation coverage for specific APIs. For instance, in the empirical study presented in [107] authors show that StackOverflow contains questions and answers covering 87% of the Android APIs. To perform the analysis, authors conceived a supporting platform to maintain an API index i.e., a dataset to keep track of the links between retrieved SO posts and related elements of the API being considered.

In this deliverable, we implement and evaluate SOrec, a recommender system for searching for relevant StackOverflow posts. SOrec distinguishes itself from current approaches that deal with the mining of StackOverflow as it addresses different phases of the whole searching process, i.e., Index Creation, Query Creation and Query Execution. To this end, SOrec attempts to effectively exploit the well-defined indexing and searching mechanisms provided by Lucene to increase the exposure of queries to the indexed data. More details of our proposed approach are presented in Section 7.

2.5 Classification of StackOverflow posts

Discussions in online forums contain useful instructions on how to leverage a specific API. However, such communication is normally dispersed among various threads, and the grouping of similar API discussions into a single category will free developers from the search effort. In this section, we review some of the most notable studies that deal with this issue.

Hou and Mo [54] perform an investigation of how well machine learning algorithms can be applied to categorize API discussions into specific topics according to their content. A Naïve Bayes classifier has been exploited to classify API discussions into API specific topics. Furthermore, the authors also investigated the impacts of various feature selection methods on classification accuracy, including stop words removal, words splitting. The approach was evaluated by using three groups of posts, consisting of 46, 158, and 833 documents, respectively. The authors also concluded that the classification performance is largely affected given that documents can be classified with multi labels.

CASE is an automated classifier that works by caching a subset of terms whose affinity scores to each category are the highest, and then building a classifier based on the cached terms [158]. CASE considers different inputs for its computation, i.e., text, code, and the combination of both. Starting from a post, the system performs some pre-processing steps to convert into a feature vector. First, it parses each post into token, removes stop words and stems the remaining tokens. Afterwards, CASE represents each post using bag-of-words. To reduce the number of features, a term cache algorithm was then proposed to select the most representative terms. CASE has been evaluated using the 3 datasets curated by *Hou and Mo* [54]. The experiment results show that CASE achieves accuracy scores of 0.69, 0.77, and 0.96 for the 3 datasets and this means that CASE clearly outperforms the approach proposed by *Hou and Mo* [54].

Beyer et al. [15] present an automated classifier using two machine learning algorithms, namely Random Forest and Support Vector Machines. The obtained model can be used to aid developers in browsing SO discussions or researchers in building recommenders based on SO. A dataset of 500 StackOverflow posts has been curated and manually classified into seven categories. The evaluation has been conducted using various experimental configurations with respect to the representation of the input data. The experimental results show that when Random Forest is used together with phrases as input data, the proposed models obtain the best classification performance. In particular, an average precision and recall of 0.88 and 0.87.

In this deliverable, we implement and evaluate SCORE, a system for the automated classification of SO posts into independent categories (see Section 8). The final aim is to provide developers with posts discussing a same topic to a given API. This is highly useful since it helps developers quickly approach the set of most relevant SO posts. We compared SCORE with three state-of-the-art tools, i.e., the ones mentioned above [15],[54],[158], and we demonstrate that our proposed tool outperforms the baselines.

2.6 Neural Networks in Software Engineering

The ability to learn from labeled data allows neural networks to have a wide range of applications. The work in [17] presents an overview of neural networks for pattern recognition. A multi-purpose algorithm based on a single deep convolutional neural network (CNN) for solving various problems, e.g., face detection, face alignment, smile detection is introduced in [115]. The authors in [154] review different applications of neural networks also identifying the characteristics that make them suitable for forecasting tasks. The work also identifies issues related to the selection of number hidden layers as well as the number of neurons.

For classification, neural networks have demonstrated their suitability in various application domains. An approach to classify individual characteristics in behavioural sciences using a neural network is proposed

in [116]. The authors in [6] investigate two different types of neural networks for classification, i.e., back-propagation and probabilistic neural network and find out that only the latter is suitable for the detection of novel patterns. A series of experiments with convolutional neural networks for sentence-level classification tasks is reported in [63].

A deep neural network (DNN) has been exploited to find similar software applications [89]. Apart from the input and output layers, two hidden layers are used to derive the high-level concepts for each project. 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 system produces an N -dimension vector, where each entry corresponds to the likelihood of the project being classified to a given category. The proposed approach has been evaluated by comparing with some standard machine learning approaches, i.e., naive Bayes and classical neural networks using ten-fold cross-validation. 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.

Le Clair et al. [69] develop a neural text classification technique to deal with the representation of source code. The approach takes into consideration the fact 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. A word embedding technique is exploited to assign a single category to a project. Each software 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 integer numbers. A word is represented as a vector of integers and it is used to feed a neural network and to produce categories. After the preprocessing phase, a neural network which is made of three layers is used to classify the input data. The first layer, so-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.

DeepAPI [47] is a deep-learning method used to generate API usage sequences given a query in natural language. The learning problem is encoded as a machine translation problem, where queries are considered the source language and API sequences the target language. Only commented methods are considered during the search. The same authors [46] present CODenn (Code-Description Embedding Neural Network), where, instead of API sequences, code snippets are retrieved to the developer based on semantic aspects such as API sequences, comments, method names, and tokens.

In this deliverable, we exploit a feed-forward neural network to build SCORE, a supervised classifier to categorize StackOverflow posts. An evaluation using various datasets demonstrates that the tool is able to learn from manually classified data and effectively categorize incoming unlabeled data, obtaining a high prediction performance thus outperforming two baselines.

3 The CROSSMINER Recommender Systems

In recent years, recommender systems have gained momentum in various disciplines. For instance, in customer service systems recommendation techniques are implemented as a means to help users make an appropriate choice [153]. In the entertainment industry, recommender systems are used to suggest movies or music to users, according to their personal preferences [31],[38],[99]. To name a few, Netflix [45], Amazon [74], and LinkedIn [149] are among the most notable online services that launch the recommendation feature, with the ultimate aim of tailoring services to customers' needs [104].

In Software Engineering, recommender systems have become popular as they are deployed to provide developers with resources that are considered to be useful for their current development tasks [43],[92]. This feature contributes towards the reduction in the time spent to discover and understand artifacts from OSS repositories, thus fostering re-usability and productivity [121].

Our approach is based on recommender systems that provide recommendations to developers with regards to their development context. We derive recommendation techniques from the mechanisms implemented for e-commerce systems [74]. There, given a customer, products that have been purchased by similar customers are recommended to her [130]. Similarly, given a software project, we recommend artifacts that exist in projects that are similar to it. A *content-based recommender system* works by recommending to a developer various artifacts, e.g., code snippets, API method invocations and external libraries that are similar to the projects being developed [108]. A *collaborative-filtering recommender system* recommends artifacts to a developer based on the artifacts used by developers with similar behaviors [129, 131].

Figure 1 depicts an overview of the approach we have been promoting in the context of the CROSSMINER project. We feed a Knowledge Base with data curated from OSS forges. The OSS ecosystem representer transforms the collected metadata into a mathematically computable format which then serves as input for the Similarity Calculator and the Recommender systems. The goal is to support the development of new software systems by relying on existing and reusable open source components. Such recommendations are produced by mining heterogeneous sources of information including source code repositories, bug tracking systems, forums, and Q&A systems like StackOverflow [2].

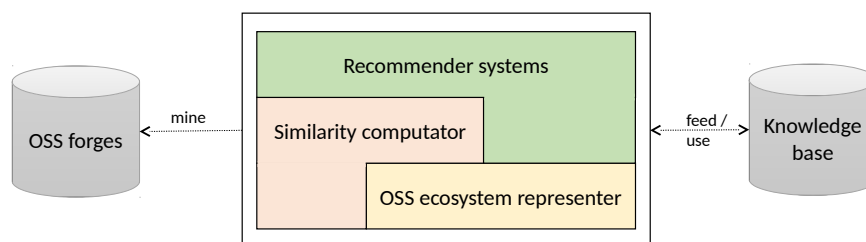


Figure 1: The main components underpinning the CROSSMINER recommendation systems.

The development of recommendation providers as shown in Figure 1 has been realized by relying on existing recommender system techniques. In particular, collaborative-filtering recommender systems (CFRSs) have been employed to suggest developers additional third-party libraries that should be included in the system being developed. Meanwhile, to conceive recommendations consisting of API function calls and source code snippets, we make use of a context-aware recommender system (CARS). Collaborative-filtering recommender systems work on the premise that “*if customers agree about the quality or relevance of some items, then they will likely agree about other items*” [131]. A CFRS exploits a 2-D matrix to represent the relationships between users and items and computes missing ratings. The CF technique can be applied in mining OSS repositories,

as long as we can find suitable mappings from the domain of product recommendation. A possible mapping is as follows, if we consider *projects* as *customers*, and *libraries* as *products*, then we can recommend third-party libraries using the CF technique. Instead of recommending products, we recommend third-party libraries to projects using an analogous mechanism: “if projects share some libraries in common, then they will probably share other common libraries.”

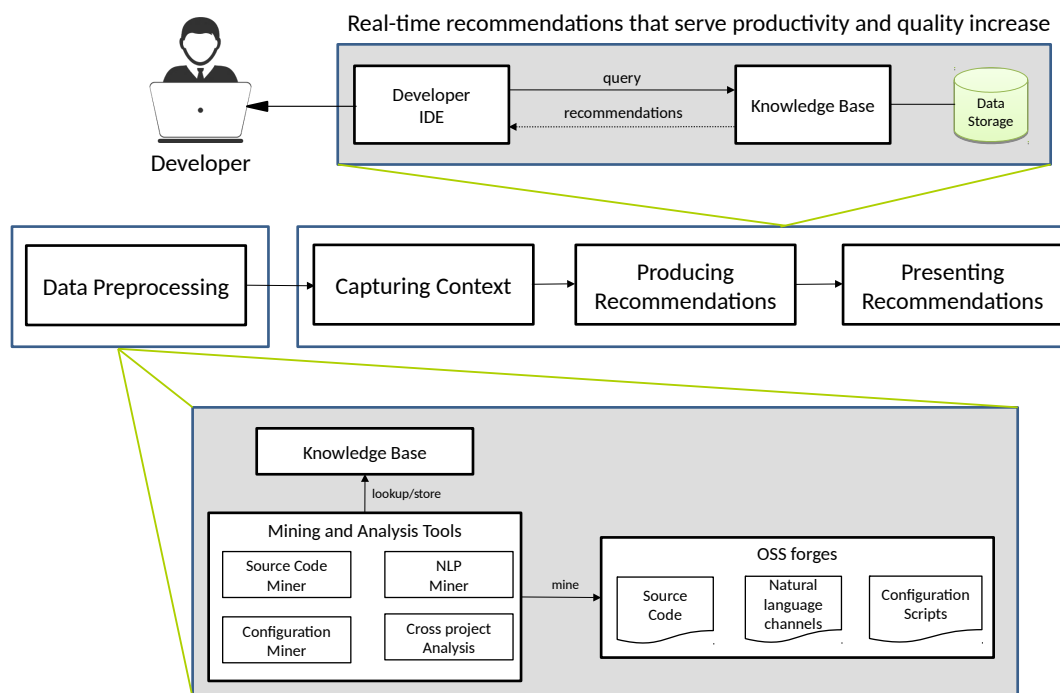


Figure 2: A high-level view of the CROSSMINER architecture.

The proposed solution is made up of four main modules as depicted in Figure 2. The Data Preprocessing module contains tools that extract metadata from OSS repositories. Data can be of different types, such as: source code, configuration, or cross project relationships. Natural language processing (NLP) tools are also deployed to analyze developer forums and discussions. The collected data is used to populate a knowledge base which serves as the core for the mining functionalities. By capturing developers' activities (Capturing Context), an IDE is able to generate and display recommendations (Producing Recommendations and Presenting Recommendations)

4 Recommendation of project alternatives with similar APIs

In this section, we present an approach for recommending projects with similar APIs. Based on the model for computing similarity among OSS repositories presented in Deliverable D6.2, i.e., CROSSSIM [90], in this section we derive a technique that allows for finding similar projects with respect to the third-party library usage, given a specific project.

4.1 Proposed Approach

Computing similarities among software systems is considered to be a difficult task [29],[82]. In addition, the miscellaneousness of artifacts in OSS repositories as well as their cross relationship makes the similarity computation even more complicated. For recommender systems in general, the ability to measure the similarity between items plays an important role in obtaining relevant recommendations [48]. Intuitively, for software mining recommender systems, the measurement of similarities between artifacts, e.g. projects, dependencies, code snippets, or even developers shall also be a critical factor. Nevertheless, the computation of similarities between software systems/open source projects in particular has been identified as a thorny issue [82]. Furthermore, considering the miscellaneousness of artifacts in OSS repositories, similarity computation becomes very complicated as many artifacts and several cross relationships prevail.

To enable both the representation of different OSS projects and the calculation of their similarity, a *graph-based* model has been conceived. Specifically, the model has been chosen since it allows for flexible data integration and facilitates numerous similarity metrics [18]. We consider the community of developers together with OSS projects, libraries and all mutual interactions as an *ecosystem*. Graphs are then used for representing different types of relationships in the OSS ecosystem.

We exploit the tool developed in Task 6.2, i.e., CROSSSIM [90] for this purpose. However, in order to meet the requirements specific to this type of recommendation, we impose some changes to the original graph. Since CrossSim offers a flexible representation of the OSS ecosystem, only third-party libraries are used as feature. Furthermore instead of using *isUsedBy* we reverse the direction of one edge, resulting in *includes*. As an example, we consider a set of four projects $P = \{p_1, p_2, p_3, p_4\}$ and a set of libraries $L = \{lib_1=junit:junit; lib_2=commons-io:commons-io; lib_3=log4j:log4j; lib_4=org.slf4j:slf4j-api; lib_5=org.slf4j:slf4j-log4j12\}$. By observing the *pom.xml* files⁷ of the projects in P , we discovered the following inclusions: $p_1 \ni lib_1, lib_2$; $p_2 \ni lib_1, lib_3$; $p_3 \ni lib_1, lib_3, lib_4, lib_5$; $p_4 \ni lib_1, lib_2, lib_4, lib_5$. The graph for the set of projects is depicted in Figure 3.

We adopt the approach proposed by Di Noia *et al.* [38] to compute the similarities among OSS graph nodes, as it has been successfully exploited by many studies to do the same task [22],[61]. Among other relationships, two nodes in a graph are deemed to be similar if they point to the same node with the same semantic edge. For instance, the graph in Figure 3 depicts the relationships among 4 projects and 5 third-party libraries. There, we see that p_3 and p_4 are highly similar since they both point to three nodes representing lib_1, lib_4, lib_5 . This reflects the actual relationship of the two projects by the view of Software Engineering: they are similar since they implement common functionalities by using common libraries [82]. By the view of recommendation algorithms, given a project p , suggested libraries for p should come from the most similar projects to p [104].

Using this metric, the similarity between two project nodes p and q in an OSS graph is computed by considering their feature sets [38]. Given that p has a set of neighbour nodes $(lib_1, lib_2, \dots, lib_n)$, the features of p are

⁷The file *pom.xml* defines all project dependencies with external Maven libraries (<https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>)

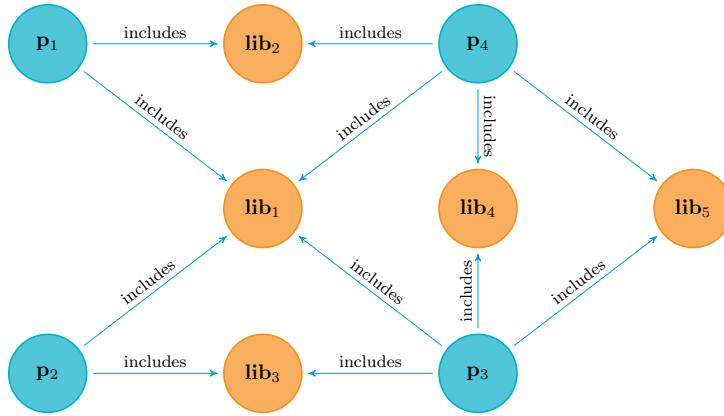


Figure 3: Graph representation for projects and libraries

represented by a vector $\vec{\phi} = (\phi_1, \phi_2, \dots, \phi_n)$, with ϕ_i being the weight of node lib_i . It is computed as the *term-frequency inverse document frequency* value as follows:

$$\phi_i = f_{lib_i} * \log\left(\frac{|P|}{a_{lib_i}}\right) \quad (5)$$

where f_{lib_i} is the number of occurrence of lib_i with respect to p , it can be either 0 and 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*. According to Eq. 5, node lib_1 in Figure 3 has a low weight compared to other nodes since it is pointed by all four project nodes. In practice, this is comprehensible since *junit:junit* is a very popular dependency and thus it should have a less important role in characterizing a project.

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

$$sim(p, q) = \frac{\sum_{i=1}^n \phi_i \times \omega_i}{\sqrt{\sum_{i=1}^n (\phi_i)^2} \times \sqrt{\sum_{i=1}^n (\omega_i)^2}} \quad (6)$$

where n is the cardinality of the set of libraries that p and q share in common [38]. Intuitively, p and q are represented as vectors in an n -dimensional space and Eq. 6 measures the cosine of the angle between the two vectors.

4.2 Evaluation

The proposed technique is used to find project alternatives with similar APIs, given an input project. Interestingly, it is also applicable to the module for computing project similarity by the third-party library recommender system, which is going to be detailed later on in this deliverable. Thus, rather than performing a separate performance evaluation for this technique, we decided to combine it with the one for third-party library recommendation in Section 6.

5 API function calls and usage patterns recommendation

The proliferation of OSS repositories in recent years has substantially changed the way people develop software. Developers are free to navigate a large information space to search for and re-use artifacts that benefit their development tasks. Among other activities, embedding API function calls or concrete usage patterns allows for simplifying integration and thus speeding up the development process. Nevertheless, searching for suitable APIs to be integrated is an uphill task due to the plethora of information available at several OSS repositories. We aim at assisting developers in mining relevant API function calls and usage patterns from open source projects. We built a context-aware collaborative-filtering system that exploits the cross relationships among different artifacts in OSS projects to represent them in a graph and eventually to predict the inclusion of additional API invocations. An evaluation on a dataset curated from the Maven repository shows that our proposed approach obtains a good performance with respect to various quality indicators. We believe that the deployment of a context-aware recommender system to provide APIs recommendation is meaningful and promising.

5.1 Overview

Third-party libraries are interface to runnable and reusable code snippets, they can be embedded in source code projects independently from environment code [120]. For software developers, integrating external components into source code being developed is a daily routine since this helps accelerate the development process as well as increase reliability. Rather than programming from scratch, developers look for libraries that implement the desired functionalities and integrate them into their existing projects [92]. For such libraries, API function calls are the entry point which allows one to invoke the offered functionalities. However, in order to exploit a library to implement the required feature, programmers need to consult various sources, e.g. API documentation to see how a specific API instance is utilized in practice. Nevertheless, from these external sources, there are only texts providing generic syntax or simple usage of the API, which may be less relevant to the current development context. In this sense, concrete examples of source code snippets that indicate how specific API function calls are deployed in actual usage, are of great use [87].

Two major types of API mining methods have been identified [60]. By the first type, a system suggests a ranked list of API invocations and the programmer exercises her own discretion in choosing suitable function calls. On one side, this scheme gives more freedom to the integration process, on the other side, it lacks details of control flow and might be suitable only for skilled developers. By the second type of recommendations, the developer gets real code snippets that implement specific functionalities and she can directly embed the most relevant snippet into the current function. Such recommendation is considered to be more practical and helpful. However, since an API function can be used in different contexts depending on the purposes, simply recommending a code snippet that contains some API functions may not be the solution, it can be irrelevant to the context as a whole [157].

Clustering has been considered as the *de facto* mechanism for finding similar source code snippets, aiming to remove redundant items [60],[157],[146]. Nevertheless, a substantial amount of redundancy is still witnessed by approaches that rely on clustering [41]. Furthermore, existing studies seem not to fully exploit the context data on which the developer is working to mine the most relevant API usages. In our view, the current project together with all API function calls, provide a precise insight into the development context and they should be properly incorporated into the recommendation process. Within the scope of Work Package 6, we develop FOCUS, a context-aware collaborative filtering recommender system that mines OSS repositories to provide developers with API Function Calls and Usage patterns [96]. FOCUS concurrently supports two functionalities: API function calls recommendation and API usage patterns recommendation.

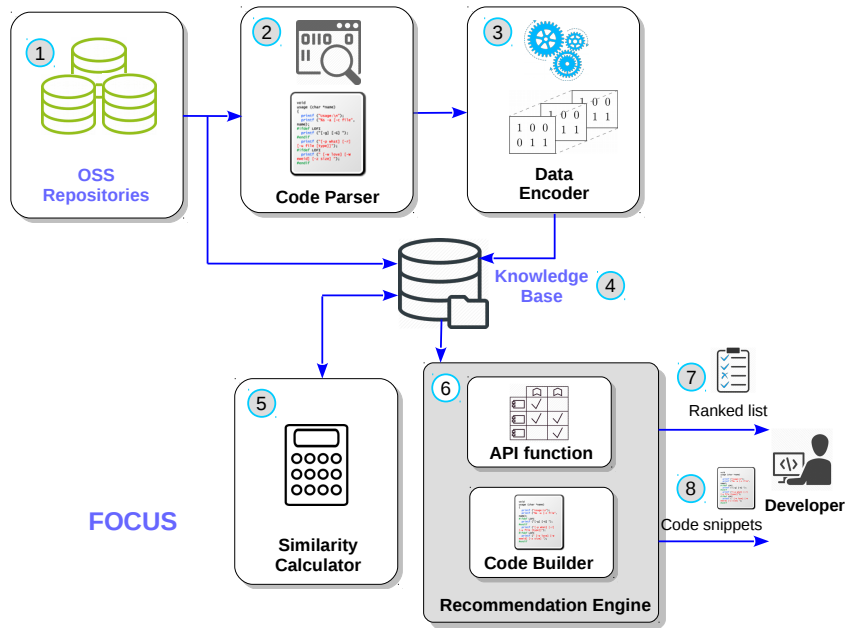


Figure 4: Overview of the FOCUS architecture.

Our work distinguishes itself from other existing approaches in terms of the following aspects. First, it employs a new model to represent the mutual relationships in source code. Second, given a project, FOCUS collaboratively mines API usage from the most similar projects. It attempts to narrow down the search scope and to *focus on* only the most relevant API usages by *collaboratively filtering out* irrelevant items. Furthermore, to the best of our knowledge, existing studies rely heavily on the manual analysis of recommendation outcome by human to validate performance. Nonetheless, such an analysis is labor intensive, time consuming and prone to subjective perception [23]. We get rid of a user study and propose an approach to automate the evaluation process by means of cross validation, splitting a dataset into independent subsets, namely *training data* and *testing data*. In this sense, the main contributions of this chapter are summarized as follows:

- Deploy a context-aware collaborative-filtering technique for recommending API function calls and usage patterns;
- Introduce an approach for evaluating the outcomes of API function calls and usage patterns recommendation without relying on a user study;
- Evaluate the performance of the proposed approach on a dataset curated from the Maven repository⁸.

5.2 Architecture

Different from other existing approaches which normally rely on clustering to find API calls, FOCUS utilizes a context-aware collaborative-filtering technique to search for invocations from closely relevant projects. The FOCUS architecture is depicted in Figure 4. By connecting to OSS repositories ①, the Code Parser ② extracts source code projects to obtain relevant metadata. For each project, its source code is also stored to the database for further retrieval. The Data Encoder ③ gets input metadata and represents it in a *mathematically computable format*. The similarities among projects, declarations are computed by the Similarity

⁸<https://mvnrepository.com>

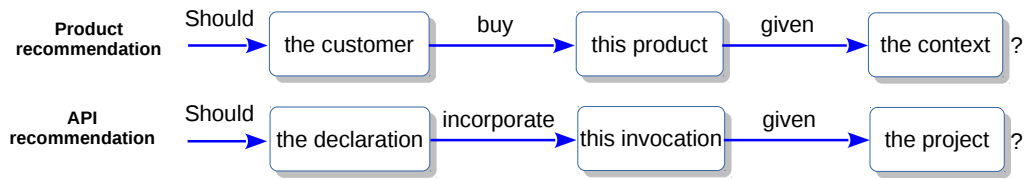


Figure 5: Comparison.

Calculator ⑤. The Recommendation Engine ⑥ exploits the similarity scores and generates recommendations. In particular, the API Generator returns a ranked list of invocations ⑦, whereas the Code Builder queries the Knowledge Base to get real source code snippets ⑧ and recommends to the developer.

The following terms are introduced to pave the way for further presentation:

- Method invocation (or invocation): a function call from an external API, i.e. i_k ;
- Method declaration (or declaration): a single source code unit, i.e. a function/procedure, that contains various invocations from different APIs, i.e. $d_r \ni (i_1^r, i_2^r, \dots, i_l^r)$ to realize a specific functionality;
- Software project (or project): a complete, standalone source code unit that consists of a set of declarations $p \ni (d_1, d_2, \dots, d_n)$ to perform a particular job.

By a context-aware collaborative-filtering technique, given a customer who needs recommendations on what additional products should be put into the shopping cart, the intuition is to collaboratively deduce the presence of prospective items from *those that have been purchased by similar customers in comparable contexts* [28]. Here, similar customers and contexts play the role of a *filter* that helps narrow down the search scope. Instead of searching in all projects, the search engine only looks for invocations from similar projects, thus yielding the feature *collaborative filtering*. To solve the problem of API usage recommendation, an analogous mechanism is applied. By considering the following mappings: *projects–contexts*, *declarations–customers*, and *invocations–products*, we are able to transform the model applied in products recommendation into a solution for recommending API usages. Intuitively, as illustrated in Figure 5, we simulate the situation when a customer (a declaration) wonders if she (it) should buy (incorporate) a product (an invocation) given a concrete context (the current project). Given an active declaration, we search for prospective invocations from *those in similar declarations belonging to comparable projects*.

5.3 Data Representation

Figure 6 depicts an example with the Java code snippets of a method declaration named `clone()`. For a project, FOCUS works on the basis of declarations and invocations. In the first place, it is necessary to directly extract those artifacts from source code. We chose Rascal M³ [56] since it allows to parse various types of input data. Given a project, the tool is used to extract *relevant* declarations and invocations, i.e. those that come from third-party libraries.

For each declaration or invocation, Rascal M³ returns the full name of data type for all the parameters. Though this seems trivial at first glance, it is useful since it helps distinguish invocations with same name, but containing different types of input parameters. The metadata extracted for the method introduced in Figure 6 is depicted in Figure 7. The declaration is represented as `ClientRequestImpl/clone(javax.ws.rs.core.MultivaluedMap)` by Rascal M³.

```
private static MultivaluedMap<String, Object> clone(MultivaluedMap<String, Object> md) {
    MultivaluedMap<String, Object> clone = new OutBoundHeaders();
    for (Map.Entry<String, List<Object>> e : md.entrySet()) {
        clone.put(e.getKey(), new ArrayList<Object>(e.getValue()));
    }
    return clone;
}
```

Figure 6: Method declaration **clone()**.

```
java/util/Iterator/next()
com/sun/jersey/core/header/OutBoundHeaders/OutBoundHeaders()
java/util/Iterator/hasNext()
java/util/Set/iterator()
java/util/ArrayList/ArrayList(java.util.Collection)
javax/ws/rs/core/MultivaluedMap/entrySet()
javax/ws/rs/core/MultivaluedMap/put(java.lang.Object,java.lang.Object)
java/util/Map$Entry/getValue()
```

Figure 7: Metadata extracted from **clone()**.

From the extracted metadata, we represent the relationships among declarations and invocations using a user-item matrix. In this matrix, each row represents a declaration and each column represents an invocation. A cell is set to 1 if the declaration in the corresponding row contains the invocation in the column, otherwise it is set to 0. As an example, Figure 8 shows the user-item matrix of project p_1 with 4 declarations, i.e. $p_1 \ni (d_1, d_2, d_3, d_4)$ and 4 invocations, i.e. (i_1, i_2, i_3, i_4) .

$$\begin{matrix} & i_1 & i_2 & i_3 & i_4 \\ \begin{matrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{matrix} & \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{matrix}$$

Figure 8: Representation of the relationship between declarations and invocations

To capture the intrinsic relationships among various projects, declarations and invocations, we come up with a 3-D user-item-context matrix. For example, Figure 9 depicts a set of three OSS projects $P = (p_a, p_1, p_2)$ representing by three slices with 4 declarations and 4 invocations in total. Project p_1 has already been introduced in Figure 8 and for the sake of readability, the column and row labels are removed from all the slices in Figure 9. There, p_a is the *active project* and it has an *active declaration*. *Active* here means the artifact, e.g. project, declaration, being considered/developed. Both p_1 and p_2 are complete projects and called *background data* since they are already available and serve as a base for the recommendation process. In practice, it is expected that we can store as much complete projects as possible since the more background data we have, the higher is the possibility we are able to mine relevant invocations.

5.4 Similarity Computation

By exploiting the collaborative-filtering technique, the presence of additional invocations is deduced from the most similar projects. Given an active declaration in an active project, it is essential to find a set of similar

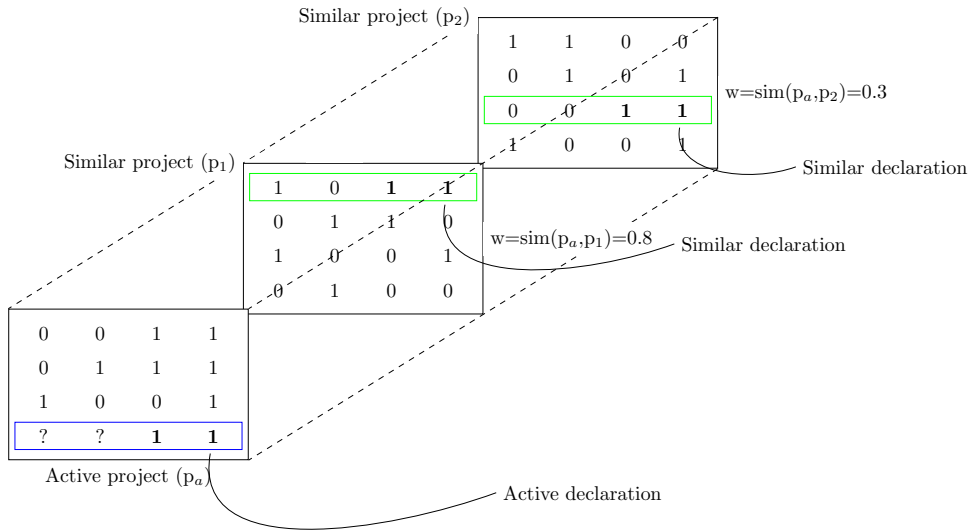


Figure 9: 3-D matrix representation of the project-declaration-invocation relationship.

projects, and then most similar declarations in those similar projects. To compute similarities, we exploit the graph representation to model the relationships among projects and invocations introduced in [90]. We employ a similar representation proposed in Section 4, however using a weighted directed graph. Each node in the graph represents either a project or an invocation, if project p contains invocation i , then there is a directed edge from p to i . The weight for an edge $p \rightarrow i$ represents the number of invocations i that occur in project p . Figure 10 depicts the graph for the set of projects introduced in Figure 9. For instance, p_a has 4 declarations and all of them have i_4 as an invocation. As a result, the edge $p_a \rightarrow i_4$ has a weight of 4. In the graph, a question mark represents missing information, since for the active declaration in p_a , it is not clear if invocations i_1 and i_2 also belong to it or not.

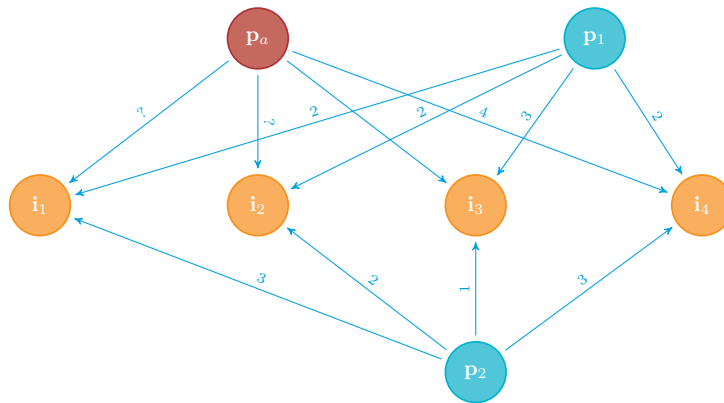


Figure 10: Graph representation of projects and invocations.

The similarity between two graph nodes p and q is computed by considering their feature sets [38]. Given that p has a set of neighbour nodes (i_1, i_2, \dots, i_l) , the features of p are represented by a vector $\vec{\phi} = (\phi_1, \phi_2, \dots, \phi_l)$,

with ϕ_k being the weight of node i_k . It is computed as the *term-frequency inverse document frequency* value as follows:

$$\phi_k = f_{i_k} * \log\left(\frac{|P|}{a_{i_k}}\right) \quad (7)$$

where f_{i_k} is the number of occurrence of i_k with respect to p , it is the weight of the edge $p \rightarrow i_k$; $|P|$ is the number of projects in the collection; a_{i_k} is the number of projects connecting to i_k . Eventually, the similarity between two projects p and q with their corresponding feature vectors $\vec{\phi} = \{\phi_k\}_{k=1,\dots,l}$ and $\vec{\omega} = \{\omega_j\}_{j=1,\dots,m}$ is computed as given below:

$$\text{sim}_\alpha(p, q) = \frac{\sum_{t=1}^n \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^n (\phi_t)^2} \times \sqrt{\sum_{t=1}^n (\omega_t)^2}} \quad (8)$$

The similarities among method declarations are calculated using the Jaccard similarity index as given below:

$$\text{sim}_\beta(d, e) = \frac{|\mathbb{F}(d) \cap \mathbb{F}(e)|}{|\mathbb{F}(d) \cup \mathbb{F}(e)|} \quad (9)$$

where $\mathbb{F}(d)$ and $\mathbb{F}(e)$ are the sets of invocations of declarations d and e , respectively.

For instance, in Figure 9 there is an active project p_a with an active declaration located at the bottom of the matrix, marked with a rectangle. By performing the similarity computation, we obtain $\text{sim}_\alpha(p_a, p_1) = 0.8$ and $\text{sim}_\alpha(p_a, p_2) = 0.3$. Furthermore, by using Eq. 9 we find two similar declarations to the active one, and they are also marked with a rectangle in p_1 and p_2 . The similar projects and declarations are used to compute a score for the missing cells. We are going to describe this process in the next section.

5.5 API function calls recommendation

In Figure 9, the active project p_a already includes three declarations, and the developer is working on the fourth declaration which corresponds to the last two-dimensional matrix. p_a consists of only two invocations, represented as the last two columns of the matrix, the cells with the value of 1. The first two cells are marked with a question mark (?), indicating that it is not clear whether these two invocations should also be incorporated into p_a . This simulates a real development scenario where the developer just started working on a new function, she has added two function calls and now expects recommendations on additional invocations. This is the point where FOCUS comes into play. The recommendation engine attempts to predict which invocations the active declaration should include by computing the missing ratings in the corresponding slice (two-dimensional matrix) by means of the following formula [28]:

$$r_{d,i,p} = \bar{r}_d + \frac{\sum_{e \in \text{topsim}(d)} (R_{e,i,p} - \bar{r}_e) \cdot \text{sim}_\beta(d, e)}{\sum_{e \in \text{topsim}(d)} \text{sim}_\beta(d, e)} \quad (10)$$

Equation 10 is used to compute a real score for the cell representing method invocation i in declaration d of project p ; $\text{topsim}(d)$ is the set of top similar declarations of d ; $\text{sim}_\beta(d, e)$ is the similarity between d and declaration e , computed using Eq. 9; \bar{r}_d and \bar{r}_e are average ratings of d and e , respectively. $R_{e,i,p}$ is understood

as the combined rating of declaration d for invocation i in all similar projects. It is computed as given below [28]:

$$R_{e,i,p} = \frac{\sum_{q \in \text{topsim}(p)} r_{e,i,q} \cdot \text{sim}_\alpha(p, q)}{\sum_{q \in \text{topsim}(p)} \text{sim}_\alpha(p, q)} \quad (11)$$

where $\text{topsim}(p)$ is the set of top similar projects of p ; $\text{sim}_\alpha(p, q)$ is the similarity between p and a similar project q , computed using Eq. 8. Equation 11 implies that more weight is given to project with a higher similarity. That means the combined rating for a cell in the active project is more affected by those from the most similar projects. In practice, it is reasonable since given a project, its similar projects seem to contain more relevant API function calls than less similar projects. Using Eq. 10 we are able to compute all the missing ratings in the active project and obtain a ranked list of invocations with scores in descending order. The list is then provided to the developer as recommendations.

5.6 API usage patterns recommendation

Using FOCUS The ranked list of items provided by FOCUS is suggested as the outcome of the recommendation engine. However, it is up to developers to decide how those invocations shall be included. On one side, this scheme gives developers freedom to proceed with the integration process. On the other side, it does not provide enough details to realize a runnable method. In this sense, this type of recommendations might be suitable only for resourceful developers. Thus, to further facilitate the development activities, we also implement a tool for recommending real code snippets.

From the ranked list of recommendations, the *top-N* items are selected and used as query to search the database for relevant declarations. The Jaccard index is used to compute similarity between a set of query items and a declaration represented by metadata consisting set of invocations stored in the Knowledge Base (see Eq. 9). For each query, we search for declarations that contain as many of the invocations in the query as possible. Eventually, the original source code corresponding to the matched metadata is retrieved from the Knowledge Base and recommended to the developer.

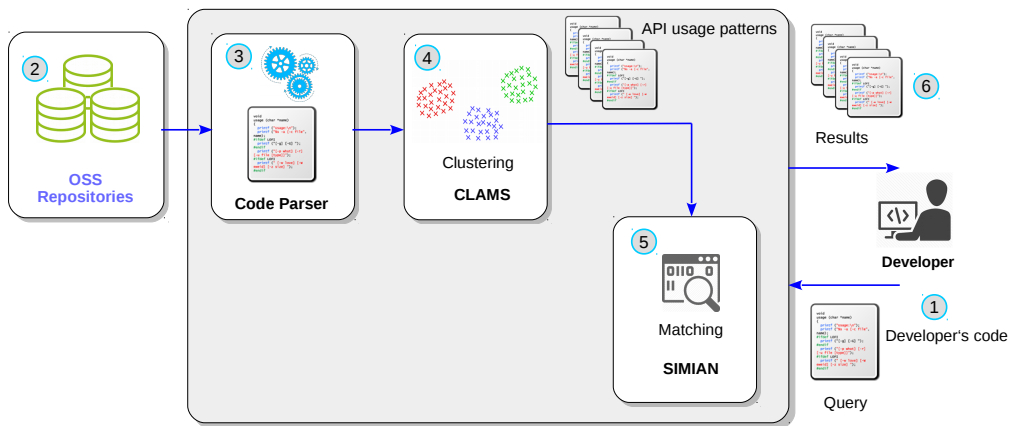


Figure 11: API usage patterns recommendation using a combination of CLAMS and Simian.

Using CLAMS and Simian To further enrich the Knowledge Base with various recommendation options, we propose an approach to suggest API usage patterns. The process applied to generate usage patterns is shown in Figure 11. Open source projects are scraped and used as input for the `Code Parser` which then extracts metadata. From this data, CLAMS [60] is applied to find API usage patterns. As input for the recommendation process, the context where the developer is working on (represented as a file containing source code) is fed to the system. Using Simian⁹, the code is matched against the patterns generated by CLAMS to find the most similar patterns. After this phase, a ranked list of usage patterns is presented to the developer.

This recommendation tool has been successfully integrated into the Knowledge Base and is available as a REST API.

5.7 Evaluation

Performing a user study has been accepted as the *de facto* method to validate an API usage recommendation tool [87, 102, 157]. Nevertheless, such a user study is not only cumbersome but also highly susceptible to errors and individual perception. We introduce a validation approach that automates the evaluation by relying solely on data. We simulate a real developer at different stages of the development process on a dataset curated from OSS repositories beforehand. The following subsections explain the evaluation in more detail.

5.7.1 Datasets

A set of 3.600 jar files was randomly collected from the Maven repository¹⁰ and named as **Dataset#1**. The `Data Parser` is used to extract relevant metadata from the projects. However, declarations that contain less than 8 invocations are discarded. Our assumption is that declarations with a low number of invocations are not helpful and shall not be considered. We checked **Dataset#1** manually and noticed that many invocations originate from a same project, i.e. they differ only in their version numbers. For instance, *ant-1.6.5.jar* and *ant-1.9.3.jar* are actually different versions of project *ant*. The collaborative-filtering technique works well given that highly similar projects exist, since it just “copies” invocations from similar methods in the very similar projects (see Eq. 10). Since too similar projects may introduce a bias against the recommendation process, we decided to create a second dataset. From **Dataset#1**, for projects with same prefix but different version numbers, we randomly selected only one among them and removed all the others. The removal results in a dataset consisting of 1.600 and we named it **Dataset#2**. Evaluation is performed on both datasets to see how well FOCUS recommends API invocations with respect to different input data.

5.7.2 Methodology

By using the datasets, we are able to validate the usefulness of the tool without resorting to a manual analysis of the recommendation outcome. A dataset is split into *training data* and *testing data* and ten-fold cross validation is conducted to validate the performance of FOCUS [32]. In particular, the dataset is split in N independent sets, $N-1$ sets are used for training and 1 for test. The experiment is repeated 10 times, each using a different set for test. The results are averaged over N . This scheme allows us to perform evaluation even on larger datasets.

Table 3 gives an example of recommendation results provided by FOCUS. On the left side of the table, there are all invocations of an active declaration. We keep only one invocation as testing data and take out the others

⁹<https://www.harukizaemon.com/simian/>

¹⁰<https://mvnrepository.com/>

	<i>Invocations</i>	<i>Rank</i>	<i>FOCUS</i>
Ground-truth	org.json4s.package\$.JObject()	1	scala.collection.immutable.List.map(scala.Function1,scala.collection.generic.CanBuildFrom)
	org.json4s.native.JsonParser\$\$anonfun\$1\$\$anonfun\$org\$json4s\$native\$JsonParser\$\$anonfun\$reverse-\$1\$2/2(org.json4s.native.JsonParser\$\$anonfun\$1)	2	scala.collection.immutable.List\$.canBuildFrom()
	scala.collection.immutable.List.reverse()	3	org.json4s.JsonAST\$.JArray.arr()
	scala.collection.immutable.List.map(scala.Function1,scala.collection.generic.CanBuildFrom)	4	scala.Option.isEmpty()
	org.json4s.JsonAST\$.JObject\$.apply(scala.collection.immutable.List)	5	scala.collection.immutable.Map\$.apply(scala.collection.Seq)
	org.json4s.JsonAST\$.JArray\$.apply(scala.collection.immutable.List)	6	org.json4s.package\$.MappingException.MappingException(java.lang.String)
	org.json4s.JsonAST\$.JArray.arr()	7	scala.collection.mutable.StringBuilder.StringBuilder()
	scala.collection.immutable.List\$.canBuildFrom()	8	org.json4s.package\$.JObject()
	org.json4s.package\$.JArray()	9	java.lang.Object.equals(java.lang.Object)
Testing	org.json4s.JsonAST\$.JObject.obj()	10	org.json4s.JsonAST\$.JObject\$.apply(scala.collection.immutable.List)

Table 3: Example of recommendation results.

and use as ground-truth data. On the left side of table, there are *top-10* items recommended by FOCUS. Among 10 retrieved items, 5 of them are found in the ground-truth data. The example shows that FOCUS is able to retrieve relevant invocations, given that only one invocation is available as input data.

We simulate different stages of a real development process to study if the system is applicable, considering a programmer who is developing a software project p . At the point of consideration, the developer already finished a number of method declarations δ , and she is now working on the active declaration d_a . For this function, the developer has just finished writing a specific number of invocations, let's say π , and she has to complete d_a . The two parameters δ , π are used to stimulate different stages of the development process. In practice, δ is low at an early stage and increases over the course of time. From a recommender system point of view, we expect to provide the developer with suitable suggestions that match the context well in various phases.

To thoroughly evaluate the performance of our proposed approach, we utilize ten-fold cross validation [65]. A dataset is split into 10 equal parts, i.e. *folds*, and an evaluation is conducted in 10 rounds. In each round of validation, nine folds are used for training and the remaining fold is for testing. Given a testing project, we call the number of method declarations as $\#dec$ and we conduct a preliminary evaluation by considering the following configurations:

Configuration Conf#1: $\delta = \#dec/2-1$, $\pi = 1$. One declaration is used as testing data, only half of the remaining declarations are used as training data and the other half are removed. For the testing declaration, only one invocation is provided as input for recommendation, and the rest is used as ground-truth data. This configuration mimics a scenario when the developer is at an early stage of the development process and therefore, only limited context data is available for feeding the recommendation engine.

Configuration Conf#2: $\delta = \#dec-1$, $\pi = 1$. One method declaration is selected as testing data, all the remaining declarations are used as training data. Similar to **Conf#1**, by the testing declaration only one invocation is kept and all the others are taken out to use as ground-truth data. This represents the stage when the developer almost finishes implementing the project.

It is expected that the proposed system can recommend items that eventually match with those stored as ground-truth data.

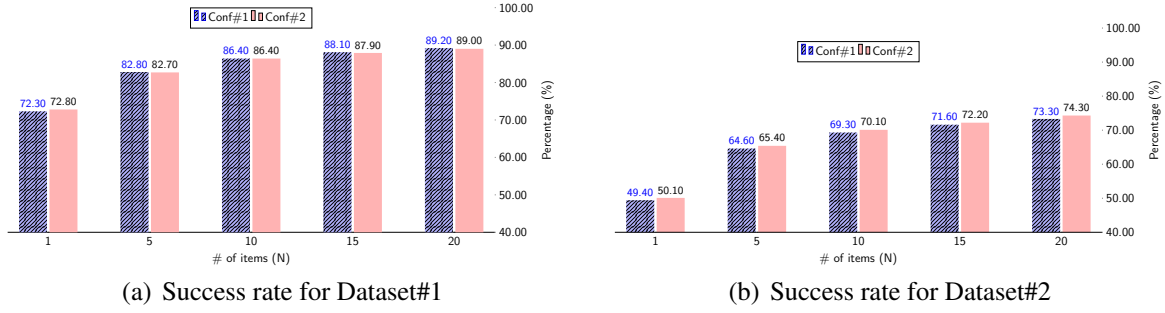


Figure 12: Success rate.

5.8 Result Analysis

Figure 12(a) and Figure 12(b) depict the success rate obtained for different cut-off values, i.e. $N = \{1, 5, 10, 15, 20\}$ for both configurations **Conf#1** and **Conf#2**. We investigate the outcome by each dataset, i.e. **Dataset#1** and **Dataset#2** separately. Interestingly, there are no big differences in success rate between two configurations for all values of N . This demonstrates that FOCUS is able to generate relevant recommendations also when only limited context data is available. By comparing the two figures we see that FOCUS produces better matches given that more similar projects are available, as in **Dataset#1** there exist projects with different version numbers.

For a small N , i.e. $N = 1$, that means when the developer expects a very brief list of recommendations, the system is still able to provide matches. For example, with **Dataset#1**, the success rates of **Conf#1** and **Conf#2** are 72.30% and 72.80%, respectively. Meanwhile, the outcome by **Dataset#2** is much lower for both configurations with $N = 1$, the success rates of **Conf#1** and **Conf#2** are 49.40% and 50.10%, respectively. This suggests that the performance of FOCUS improves considerably, given that more similar projects are available.

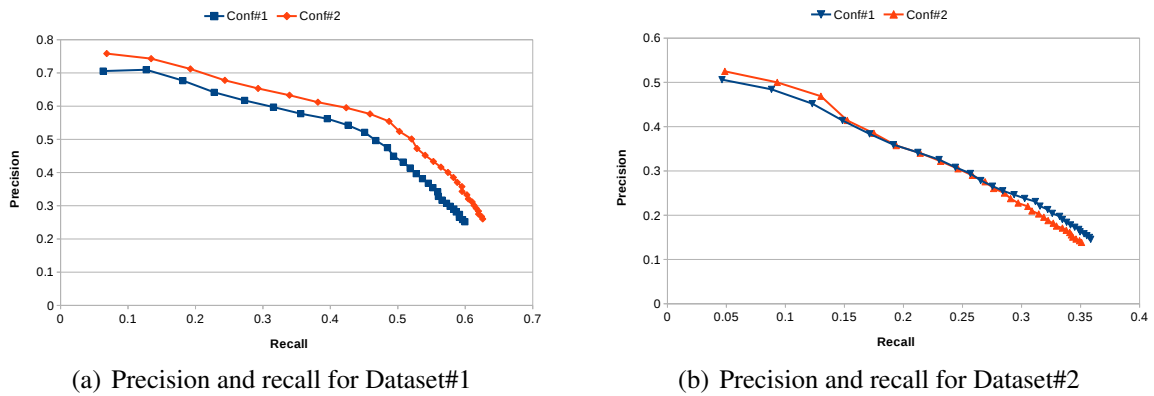


Figure 13: Accuracy.

Next, we investigate the accuracy of both configurations by varying the cut-off value N from 1 to 30 to get $Precision@N$ and $Recall@N$ and sketch the Precision-Recall curves as shown in Figure 13(a) and Figure 13(b). Considering **Dataset#1**, we witness the same trend as by success rate, since there are no big distinctions between accuracies for the configurations with all cut-off values N . Only a small difference in the

accuracy of **Conf#1** and **Conf#2** can be seen in **Dataset#1**. With this dataset, **Conf#2** has a better accuracy as the corresponding curve is closer to the top right corner. It is understandable since by **Conf#2**, there is more background data available for recommendation.

The experimental results show that FOCUS recommends API function calls with high success rates. Nevertheless, the performance of FOCUS needs to be further studied through additional experiments considering various configurations. This is considered as a future work and we will be back to fully investigate the use cases in Deliverable D6.5.

6 Third-party libraries recommendation

In OSS repositories, information piles up along with the development activities by developers as well as the interaction among users, committers, projects. Source code, project history, and communication archives are among the main artifacts that add up to the accumulations of information. A store of such kind leads to several well-defined issues, e.g. information overload, information heterogeneity, context-sensitive technical information, and software evolution [121].

To facilitate the development activities, programmers frequently consult data available at OSS repositories to look for reusable artifacts such as source code [137], API documentation [112], or API elements [121]. Among others, finding and reusing third-party libraries are activities that programmers regularly perform during the development phase. Third-party libraries are highly advantageous to the development activities since they provide programmers with several tailor-made functionalities [71]. Instead of programming from scratch, one only needs to search for libraries that implement the desired functionalities and embed into the existing code [121]. In many Android apps, third-party libraries are a staple element [79], a recent study shows that sub-packages from external libraries account for 60% of code in Android software [145].

Nevertheless, due to the heterogeneity of resources and their corresponding dependencies, developers need to spend a lot of effort to search for relevant items [121]. Despite the necessity, very little work has been conducted concerning the techniques that facilitate the search for suitable libraries from OSS repositories. Most of the related studies address the issue of finding code snippets [137] or API function calls [139]. To the best of our knowledge, LibRec [138] is among the most advanced techniques for library recommendation to support OSS developers. It was designed to find relevant libraries, based on the current set of items that a project already includes. LibRec has been demonstrated to be able to recommend project libraries with a high *success rate*. However, as shown later in this chapter, the performance of the approach can be improved with respect to different quality metrics.

6.1 Overview

We developed CROSSREC, a framework that exploits **C**ross **P**rojects **R**elationships among **O**pen **S**ource **S**oftware **R**epositories to build a **R**ecommender **S**ystem. With regards to the rich metadata infrastructure available at OSS repositories, we represent the cross relationships among them using the graph model, so as to compute similarities among various artifacts. We propose a novel approach utilizing a collaborative-filtering technique [131] to recommend third-party libraries to support OSS developers. Originally, the technique was developed for e-commerce systems to exploit the relationships among users and products to predict the missing ratings of prospective items [74],[105]. The technique is based on the premise that *“if users agree about the quality or relevance of some items, then they will likely agree about other items”* [131]. Our approach is derived from this to solve the problem of library recommendation. Instead of recommending goods or services to customers, we recommend third-party libraries to projects using an analogous mechanism: *“if projects share some libraries in common, then they will probably share other common libraries.”* In a nutshell, we work towards the search for an appropriate answer for the question: *“Which third-party libraries should I further include in my current project?”* Thus, CROSSREC aims at supporting software developers who have already included some libraries in the new projects being developed, and expect to get recommendations on which additional libraries should be further incorporated (if any). To this end, the main contributions of this chapter are as follows:

- Introducing a representation model to describe the relationship among projects and third-party libraries in OSS repositories and to compute similarities;

- Building a collaborative-filtering recommender system to assist software developers with the selection of suitable third-party libraries;
- Proposing a set of metrics for the measurement of quality of suggestions provided by a library recommender system. Besides *success rate*, we consider also *accuracy*, *sales diversity*, *novelty* as they have been widely utilized in evaluating recommender systems [99],[121];
- Performing an empirical study on the performance of CROSSREC and LibRec with a dataset of 1.200 GitHub Java projects utilizing the above mentioned quality metrics.

6.2 Architecture

In this section the CROSSREC approach is presented. It provides a library recommendation functionality, which is meaningful to open source software developers since it allows them to search for third-party libraries that may be useful for their current project. The architecture of CROSSREC is shown in Figure 14 and consists of the software components supporting the following activities:

- Representing the relationships among projects and libraries retrieved from existing repositories;
- Computing similarities to find projects, which are similar to that under development;
- Recommending libraries to projects using a collaborative-filtering technique.

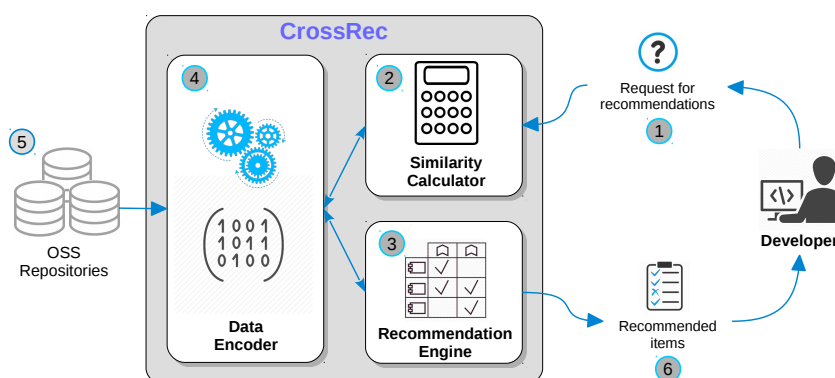


Figure 14: CROSSREC Architecture.

In particular, as shown in Figure 14, the developer interacts with the system by sending a request for recommendations. The request contains a list of libraries that are already included in the current project. The Data Encoder collects *background data* from OSS repositories, represents them in a *mathematically computable format*, which is then used as a base for other components of CROSSREC. The Similarity Computation module calculates similarities among projects to find the most similar ones to the given project. The Recommendation Engine gets the list of *top-k* similar projects and executes recommendation techniques to generate a ranked list of *top-N* libraries. Finally, the recommendations are sent back to the developer. Background data can be collected from different OSS platforms like GitHub¹¹, Eclipse¹², BitBucket¹³. The current version of CROSSREC supports data extraction from GitHub, even though the support for additional platforms is already under development.

¹¹GitHub <https://www.github.com>

¹²Eclipse <https://projects.eclipse.org>

¹³BitBucket <https://www.bitbucket.com>

The source code implementing CROSSREC is available for download at [39]. In the following, the components Data Encoder, Similarity Calculator, and Recommendation Engine are singularly described.

6.3 Data Encoder

A recommender system for online services is based on three key components, namely *users*, *items* and *ratings* [104, 129]. A *user-item ratings matrix* is built to represent the mutual relationships among the components. Specifically, in the matrix a user is represented by a row, an item is represented by a column and each cell in the matrix corresponds to a rating given by a user for an item [104]. For library recommendation, instead of users and items, there are projects and third-party libraries, and a project may include various libraries to implement desired functionalities.

We derive an analogous user-item ratings matrix to represent the *project-library inclusion* relationships, denoted as \ni . In this matrix, each row represents a project and each column represents a library. A cell in the matrix is set to 1 if the library in the column is included in the project specified by the row, otherwise it is set to 0. For the sake of clarity and conformance, we still denote this as a user-item ratings matrix throughout this document.

For explanatory purposes, we refer to the set of projects introduced in Section 4. There is a set of four projects $P = \{p_1, p_2, p_3, p_4\}$ together with a set of libraries $L = \{lib_1=junit:junit; lib_2=commons-io:commons-io; lib_3=log4j:log4j; lib_4=org.slf4j:slf4j-api; lib_5=org.slf4j:slf4j-log4j12\}$ and the following inclusions: $p_1 \ni lib_1, lib_2$; $p_2 \ni lib_1, lib_3$; $p_3 \ni lib_1, lib_3, lib_4, lib_5$; $p_4 \ni lib_1, lib_2, lib_4, lib_5$. Accordingly, the user-item ratings matrix built to model the occurrence of the libraries is depicted in Figure 15.

$$\begin{matrix} & \begin{matrix} lib_1 & lib_2 & lib_3 & lib_4 & lib_5 \end{matrix} \\ \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{matrix} & \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{matrix}$$

Figure 15: An example of a user-item ratings matrix to model the inclusion of third-party libraries.

6.4 Similarity Calculator

The Recommendation Engine of CROSSREC works on the basis of an analogous user-item ratings matrix. To provide inputs for this module, the first task of CROSSREC is to perform similarity computation to find the most similar projects to a given project. We adopt the approach presented in Section 4 as the mechanism for computing similarities among projects with respect to the included third-party libraries.

6.5 Recommendation Engine

The representation using a user-item ratings matrix allows for the computation of missing ratings [104, 4]. Depending on the availability of data, there are two main ways to compute the unknown ratings, namely *content-based* [108] and *collaborative-filtering* [85] recommendation techniques. The former exploits the relationships among items to predict the most similar items. The latter computes the ratings by taking into account the set of items rated by similar customers. A collaborative-filtering recommender system suggests

products that customers similar to the customer being considered have already purchased. There are two main types of collaborative-filtering recommendation: *user-based* [156] and *item-based* [129] techniques. As their names suggest, the user-based technique computes missing ratings by considering the ratings collected from similar users. Alternatively, the item-based technique does the same task by using the similarities among items [32]. In a recent work [93], we exploit user-based and item-based collaborative-filtering (CF) techniques to build a book recommender system. In contrast to many existing studies which state that the item-based CF technique outperforms the user-based CF technique [24, 59, 106], we found out that there is no distinct winner between them. Moreover, we confirm that the performance of a CF recommender system may be good with regards to some quality metrics, but not to some others.

q_1	$\left(\begin{array}{ccccc} * & * & * & * & * \end{array} \right)$
q_2	$\left(\begin{array}{ccccc} * & * & * & * & * \end{array} \right)$
q_3	$\left(\begin{array}{ccccc} * & * & * & * & * \end{array} \right)$
p	$\left(\begin{array}{ccccc} * & ? & * & * & ? \end{array} \right)$

Figure 16: Computation of missing ratings using the user-based collaborative-filtering technique.

In the context of CROSSREC, the term *rating* is understood as the occurrence of a library in a project and computing missing ratings means to predict the inclusion of additional libraries. The project that needs prediction for library inclusion is called the *active project*. By the matrix in Figure 16, p is the active project and an asterisk (*) represents a known rating, either 0 or 1, whereas a question mark (?) represents an unknown rating and needs to be predicted.

Thanks to the availability of the cross relationships as well as the possibility to compute the similarities among OSS projects by means of the graph representation, in our work we exploit the user-based collaborative-filtering technique as the engine for recommendation [55, 156]. Given an active project p , the inclusion of libraries in p can be deduced from projects that are similar to p . In particular, the user-based collaborative-filtering technique predicts a missing rating by considering the most similar projects to p . The computation is summarized as follows [24]:

- Compute the similarities between the active project and all projects in the collection;
- Select *top-k* most similar projects;
- Predict ratings by means of those collected from the most similar projects.

The rectangles in Figure 16 imply that the row-wise relationships between the active project p and the similar projects q_1, q_2, q_3 are exploited to compute the missing ratings for p . The following formula is used to predict if p should include l (or $p \ni l$) [104]:

$$r_{p,l} = \bar{r}_p + \frac{\sum_{q \in \text{topsim}(p)} (r_{q,l} - \bar{r}_q) \cdot \text{sim}(p, q)}{\sum_{q \in \text{topsim}(p)} \text{sim}(p, q)} \quad (12)$$

where \bar{r}_p and \bar{r}_q are the average rating of p and q , respectively; q belongs to the set of *top-k* most similar projects to p , denoted as $\text{topsim}(p)$. For a testing project p , \bar{r}_p is equal to 1 since the ratings for all testing libraries are 1. $\text{sim}(p, q)$ is the similarity between the active project and a project q , and it is computed using Equation 8.

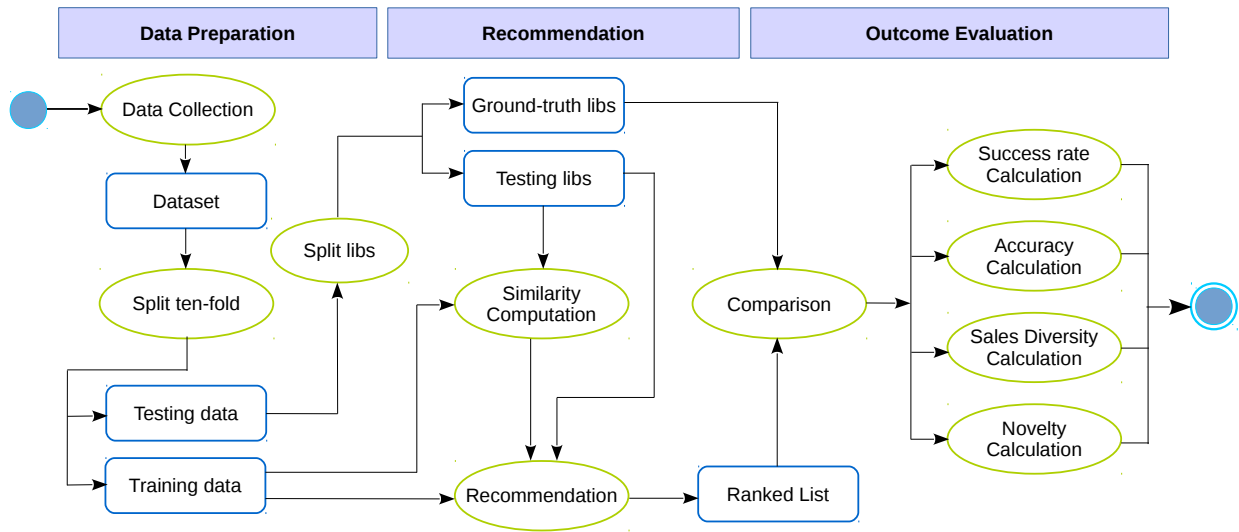


Figure 17: Evaluation Process.

CROSSREC has been designed as a flexible framework and it can integrate different similarity metrics as well as recommendation engines into its core. The current implementation of CROSSREC supports the technique presented in Section 4 as the similarity computation technique and the user-based collaborative-filtering mechanism as the recommendation engine. However, it is feasible to incorporate other similarity algorithms as well as recommendation techniques as long as they are suitable to work on the metadata available in OSS repositories.

In order to evaluate the performance of CROSSREC, we defined and applied the evaluation process shown in Figure 17. In particular, through private communications, the authors of LibRec provided us with its source code implementation, thus allowing us to execute it directly. Using the available implementation, we conducted a comprehensive evaluation on LibRec and CROSSREC to see if and how well they are able to suggest suitable third-party libraries. The activities of the evaluation process shown in Figure 17 and the artifacts that were produced during its execution are discussed in the following.

6.6 Evaluation

In the following, we describe the planning of our evaluation, whose *goal* is to evaluate the performance of CrossRec, and to compare it with the state-of-the-art approach LibRec.

6.6.1 Dataset

By means of the GitHub API¹⁴ we collected a dataset consisting of 1,200 Java projects. Among these projects, only 7 of them have been forked from other projects. Such original projects have been excluded from the dataset as their forked ones share highly similar libraries, and this may introduce bias in the recommendation outcomes. We represent the distribution of projects with respect to the number of forks, commits and pull requests in Figure 18. Most projects have a low number of pull requests, i.e., lower than 100, however many

¹⁴GitHub REST API v3: <https://developer.github.com/v3/>

of them have a large number of forks and commits. Forking is a means to contribute to the original repositories [58]. Furthermore, there is a strong correlation between forks and stars [20]. A project with a high number of forks means that it gets attention from the OSS community. In this sense, having many forks can be considered as a sign of a well-maintained and received project. Meanwhile, as commits have an impact on the source code [10], the number of commits is also a good indicator of how a project has been developed. We mined dependency specification by means of `code.xml` or `.gradle` files.¹⁵ Figure 19(a) and Figure 19(b) provide a summary of the dataset. Although the figures look quite similar, they convey completely different information: Figure 19(a) gives an overview on the distribution of libraries across the projects. Most of the libraries, i.e., 12,962, are used by a small number of projects, whereas only 10 libraries are extremely popular by being included in more than 200 projects. By carefully investigating the dataset, we also see that most projects contain a small number of dependencies, i.e., 48% of the projects include less than 20 libraries and just 15% of them include more than 100 libraries. The number of dependencies a project includes is depicted in Figure 19(b). Most projects contain a small number of dependencies: 580 projects include less than 20 libraries and 180 projects include more than 100 libraries.

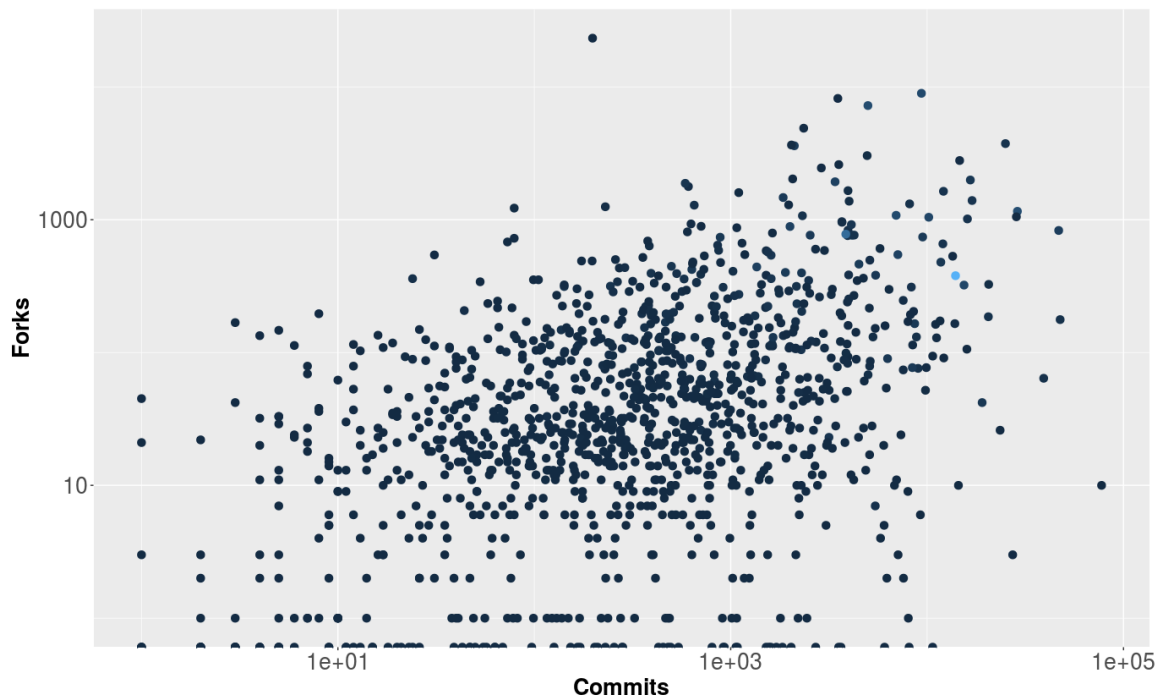
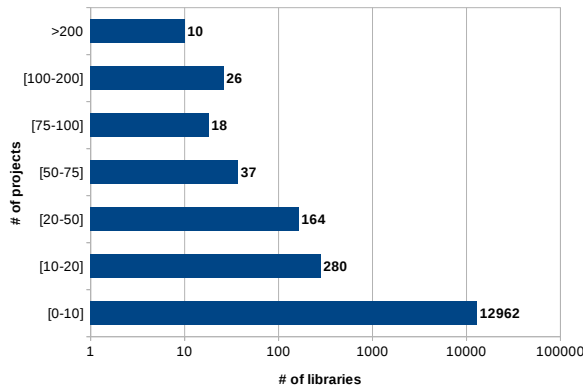


Figure 18: The projects and their number of forks, commits and pull requests.

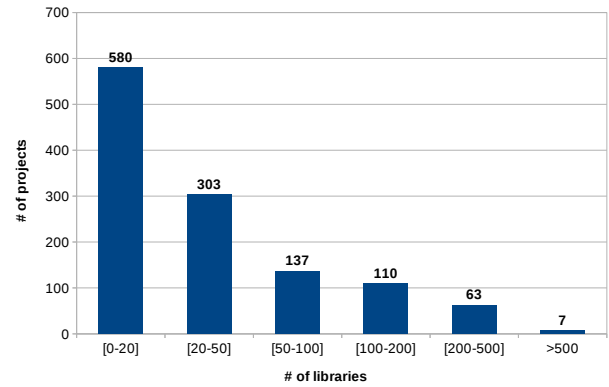
6.6.2 Evaluation metrics

We introduce a set of metrics used for evaluating the outcomes of a recommender system in the context of mining OSS repositories [92]. Given a query, the outcome of the recommendation process is a ranked list of items that are considered to be relevant for the query. For instance, a system that recommends third-party libraries for a given project returns a list in descending order of similarity scores corresponding to libraries

¹⁵The files `pom.xml` and with the extension `.gradle` are related to management of dependencies by means of Maven (<https://maven.apache.org/>) and Gradle (<https://gradle.org/>), respectively.



(a) The distribution of libraries in projects



(b) The number of libraries included in projects

Figure 19: A summary of the dataset.

	Library	Frequency
1	junit:junit	969
2	org.slf4j:slf4j-api	473
3	log4j:log4j	369
4	com.google.guava:guava	306
5	commons-io:commons-io	298
6	org.slf4j:slf4j-log4j12	275
7	commons-lang:commons-lang	227
8	commons-codec:commons-codec	215
9	org.mockito:mockito-core	213
10	javax.servlet:servlet-api	204

Table 4: Most frequent libraries in the considered dataset.

[138]. To validate the performance of a recommender system, we need a *training* and a *testing* dataset [32]. The former is used to build the model whereas the latter is used to validate the outcome. Considering a project that needs library recommendation, the graph model is used to compute similarities and then to find most k similar projects. The outcome of the recommendation is a ranked list of third-party libraries. Normally, a developer pays attention only to the *top-N* items. We use k and N as parameters for further evaluations.

We recall the following metrics that can be used to evaluate the performance of a recommender system in the context of mining software repositories given the presence of training and testing datasets. First, for a clear presentation of the metrics considered during the outcome evaluation, the following notations are defined:

- N is the cut-off value for the list of recommended items and k is the number of neighbour projects considered for the recommendation process;
- For a testing project p , the ground-truth dataset is named as $GT(p)$;
- $REC(p)$ is the *top-N* items recommended to p . It is a ranked list in descending order of real scores, with $REC_r(p)$ being the library in the position r ;
- If a recommended item $i \in REC(p)$ for a testing project p is found in the ground truth of p (i.e., $GT(p)$), hereafter we call this as a library *match* or *hit*.

Using this notation, the metrics utilized to measure the recommendation outcomes are explained below. Among others, we consider *success rate* [138], *accuracy* [82], *sales diversity*, and *novelty* the most suitable metrics for evaluating a recommender system in mining OSS repositories [121].

Success rate Given a set of P testing projects, this metric measures the rate at which a recommender system can return at least a match among *top-N* recommended items for every project $p \in P$ [138]. The metric is formally defined as follows:

$$success\ rate@N = \frac{count_{p \in P}(|GT(p) \cap (\cup_{r=1}^N REC_r(p))| > 0)}{|P|} \quad (13)$$

where the function *count()* counts the number of times that the boolean expression specified in its parameter is *true*.

Accuracy Given a list of *top-N* items, *precision@N*, *recall@N*, and *normalized discounted cumulative gain* are utilized to measure the *accuracy* of the recommendation results.

Precision@N is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

$$precision@N(p) = \frac{\sum_{r=1}^N |GT(p) \cap REC_r(p)|}{N} \quad (14)$$

Recall@N is the ratio of the ground-truth items appearing in the N items [35, 38, 99]:

$$recall@N(p) = \frac{\sum_{r=1}^N |GT(p) \cap REC_r(p)|}{|GT(p)|} \quad (15)$$

Normalized Discounted Cumulative Gain: Precision and recall reflect well the accuracy, however they neglect ranking sensitivity [11]. nDCG is an effective way to measure if a system can present highly relevant items on the top of the list:

$$nDCG@N(p) = \frac{1}{iDCG} \cdot \sum_{i=1}^N \frac{2^{rel(p,i)}}{\log_2(i+1)} \quad (16)$$

where iDCG is used to normalize the metric to 1 when an ideal ranking is reached.

Sales Diversity In e-commerce systems, *sales diversity* is the ability to improve the coverage as well as the distribution of products across customers [99, 144]. In the context of mining software repositories, sales diversity means the ability of the system to suggest to projects as much items, e.g. libraries, code snippets, as possible, as well as to disperse the concentration among all the items, instead of focusing only on a specific set of them [121].

Catalog coverage measures the percentage of items recommended to projects:

$$coverage@N = \frac{|\cup_{p \in P} \cup_{r=1}^N REC_r(p)|}{|I|} \quad (17)$$

where I is the set of all items available for recommendation and P is the set of projects.

Entropy evaluates if the recommendations are concentrated on only a small set or spread across a wide range of items [114]:

$$entropy = - \sum_{i \in I} \left(\frac{\#rec(i)}{total} \right) \ln \left(\frac{\#rec(i)}{total} \right) \quad (18)$$

where I is the set of all items available for recommendation, $\#rec(i)$ is the number of projects getting i as a recommendation, $\#rec(i) = count_{p \in P}(|(\cup_{r=1}^N REC_r(p)) \ni i|)$, $i \in I$, $total$ denotes the total number of recommended items across all projects.

Novelty *Novelty* measures if a system is able to expose items to projects. *Expected popularity complement* (EPC) is utilized to measure *novelty* and is defined as follows [143, 144]:

$$EPC@N = \frac{\sum_{p \in P} \sum_{r=1}^N \frac{rel(p,r) * [1 - pop(REC_r(p))]}{\log_2(r+1)}}{\sum_{p \in P} \sum_{r=1}^N \frac{rel(p,r)}{\log_2(r+1)}} \quad (19)$$

where $rel(p, r) = |GT(p) \cap REC_r(p)|$ represents the relevance of the item at the r position of the *top-N* list to project p ; $pop(REC_r(p))$ is the popularity of the item at the position r in the *top-N* recommended list. It is computed as the ratio between the number of projects that receive $REC_r(p)$ as recommendation over the number of projects that are recommended items among the most often recommended ones. Equation 19 implies that the more unpopular items a system recommends, the higher the EPC value it obtains and vice versa.

6.6.3 Evaluation Methodology

To thoroughly study the performance of LibRec and CROSSREC, we applied ten-fold cross validation, which has been identified as among the best methods for model selection [65]. Essentially, by referring to Figure 17, the dataset is divided into 10 equal parts, named *folds* hereafter and numbered from 1 to 10, with each consisting of 120 projects. For each recommender system, the validation has been conducted in ten rounds (see the activity *Split ten-fold*). For each round i , $fold_i$ is used as *testing data* and the remaining nine folds $fold_j$ with $j \neq i, 1 \leq i, j \leq 10$ are used as *training data* [32] (see *Split libs*). The training data is used to compute similarity and serves as the background data for recommendation, while the testing data is used to validate the model [5].

When a fold is used as testing data, library recommendation is conducted for all of its 120 projects. For every testing project p , a half of its libraries are *randomly* taken out and saved as ground truth data, let's call them **gt**, which will be used to validate the recommendation outcomes. The other half are used as testing libraries, which are called **te**, and serve as input for *Similarity Computation and Recommendation*. This mimics a real development process where a developer has already included some libraries in the current project, i.e. **te** and she awaits some recommendations, that means additional libraries to be incorporated. A recommender system is expected to provide her with the other half, i.e. **gt**.

By each validation round, we use the same folds for testing and training by both systems. Also for a testing project, the same sets of testing libraries and ground-truth libraries are used for experiments on LibRec and CROSSREC. This is to make sure that exactly the same condition is applied in both systems, so as to precisely validate their performance. The list of the projects together with the corresponding folds are publicly available in GitHub [39].

The testing libraries are provided as input to compare the similarity between the testing project and all the other 1.080 projects in the training data. For CROSSREC, a graph is built and similarity is computed using Eq. 8, whereas LibRec computes similarities as the cosine similarity between feature vectors [138]. After this phase, for each testing project there is a ranked list of projects from which the *top-k* most similar ones are selected to feed as inputs for the Recommendation Engine, which in turn predicts the ratings by computing a real score for each library following Eq. 12. For each testing project, the recommendation outcome is a Ranked List of libraries which are provided together with the ground-truth data as inputs for the Comparison module to compute the quality metrics.

By performing the evaluation with the consideration of the quality metrics previously introduced, we aim at addressing the following research questions:

- **RQ₁:** *Does CrossRec obtain a better success rate compared to LibRec?* As *success rate* was used as the only evaluation metric for LibRec [138], we exploit it to compare directly the performance of CrossRec with that of LibRec.
- **RQ₂:** *How well can LibRec and CrossRec recommend third-party libraries in terms of accuracy, sales diversity, and novelty?* It is our firm belief that *success rate* cannot fully reveal the recommendation quality. Thus, we also employ other quality indicators to painstakingly evaluate the performance of both approaches. Apart from *success rate*, we investigate if the approaches obtain a good performance in terms of *accuracy*, *diversity* and *novelty* [100],[118].
- **RQ₃:** *What are the reasons for the performance difference between LibRec and CrossRec?* We are interested in understanding the rationale behind the performance differences between the two systems.

In the next section, the outcomes of the performed evaluation are analyzed and the answers for such research questions are also given.

6.7 Result Analysis

Before addressing our research questions, we illustrate the recommendations of LibRec and CROSSREC through a running example, i.e., the project *peakgames/libgdx-stagebuilder* hosted on GitHub. As shown in Table 5, such a project uses 16 libraries, which are listed on the left-hand side of the table. Among 16 included libraries, 8 items are extracted and used as the ground-truth libraries that are shown in gray. The remaining 8 items are used as inputs for similarity computation and recommendation, or query. The output obtained by each system is a ranked list in descending order of recommendations with real scores. We took the first 10 libraries, removed the scores and kept only the order of the list to present the results as in Table 5. The top-10 items are then matched against the ground-truth data. The column **Freq.** reports the frequency of occurrence of the recommended library, over the set of 1,200 projects. In this case, LibRec only matches **junit:junit** with the ground true (which is obviously used by many projects for testing purposes) but, as we can notice, CROSSREC matches 4 more projects with the ground truth.

Both LibRec and CROSSREC obtain a *success rate@10=1.0*. However, CROSSREC has a better *recall@10* compared to LibRec as it returns more relevant items (see Eq. (15)). Furthermore, among the matches by CROSSREC, 4 items appear in the top rows of the ranked list, indicating that CROSSREC recommends with high *precision@N* (see Eq. (14)). LibRec returns only 1 relevant item, which means that both *precision@N* and *recall@N* are considerably lower compared to those of CROSSREC. Furthermore, LibRec tends to suggest very popular libraries: 6 out of 10 items recommended by LibRec are used by more than 200 projects. For instance, besides **junit:junit**, the second highest frequency item is **org.slf4j:slf4j-api** (473/1,200). By performing an

	<i>peakgames/libgdx-stagebuilder</i>	Recommendations				
	Libraries	Rank	LibRec	Freq.	CROSSREC	Freq.
Query	org.jetbrains.kotlin:kotlin-stdlib	1	junit:junit	969	com.badlogicgames.gdx:gdx-platform	3
	com.google.android:android	2	commons-io:commons-io	298	com.badlogicgames.gdx:gdx-backend-lwjgl	3
	com.google.gwt:gwt-user	3	org.slf4j:slf4j-api	473	com.badlogicgames.gdx:gdx-backend-android	3
	net.sf.kxml:kxml2	4	org.json:json	65	junit:junit	969
	org.mockito:mockito-all	5	org.slf4j:slf4j-simple	103	org.slf4j:slf4j-api	473
	com.badlogicgames.gdx:gdx-backend-gwt	6	commons-lang:commons-lang	227	org.slf4j:slf4j-jdk14	42
	net.peakgames.libgdx:stagebuilder-core	7	xmllpull:xmllpull	4	com.google.guava:guava	306
	com.badlogicgames.gdx:gdx	8	org.slf4j:slf4j-log4j12	275	com.moribitotech:mtx-core	1
Ground-truth	net.peakgames.libgdx:stagebuilder-extensions	9	com.google.guava:guava	306	com.moribitotech:mtx	1
	com.google.gwt:gwt-servlet	10	org.slf4j:slf4j-jdk14	42	com.google.gwt:gwt-servlet	27
	com.badlogicgames.gdx:gdx-platform	—	—	—	—	—
	com.badlogiclogic.gdx:gdx-backend-ios	—	—	—	—	—
	com.binarytweed:quarantining-test-runner	—	—	—	—	—
	com.badlogicgames.gdx:gdx-backend-lwjgl	—	—	—	—	—
	com.badlogicgames.gdx:gdx-backend-android	—	—	—	—	—
	junit:junit	—	—	—	—	—

Table 5: Recommendation results (matching with ground truth in bold face) for *peakgames/libgdx-stagebuilder*.

investigation on the outcome of all queries, we realize that LibRec usually recommends very popular items. The reasons for such differences are explained later on in this section.

Four out of five items recommended by CROSSREC have a low frequency of occurrence. For instance, the first item in the ranked list is **com.badlogicgames.gdx:gdx-platform** and this library is included in only 3/1,200 projects. Referring to Figure 6.6.1, it is evident that the top 3 items belong to the long tail, i.e., they are extremely unpopular since each is used by only 3 projects. However, they turn out to be useful as all of them match those stored as ground-truth data. In contrast to some existing studies which choose to recommend only popular items to developers [87],[112], we see that popularity is a good indicator for selecting a library. This implies that the novelty of a ranked list is important: a system should be able to recommend libraries that are *novel* [27], i.e., those that have been rarely seen. In this sense, we expect that CROSSREC can produce good outcomes, not only in terms of success rate and accuracy, but also sales diversity and novelty.

In summary, for the explanatory example, CROSSREC obtains a comparable success rate, but better accuracy and novelty than LibRec. This also confirms that success rate is not sufficient for evaluating the recommendation outcomes. A good recommender system is the one that can maintain a trade-off by improving diversity, novelty but still retaining a good accuracy [114]. Consequently, it is necessary to investigate if this trade-off is guaranteed by LibRec and CROSSREC, and this is done in the next sub-sections by considering the whole dataset discussed in the previous section.

The answers for the research questions of the performed evaluation are presented in the following:

RQ₁: *Does CrossRec obtain a better success rate compared to LibRec?*

To answer this question, we performed a series of experiments using different combinations of number of recommended libraries i.e., N , and number of neighbour projects exploited in the recommendation phase i.e., k . Varying N means changing the length of the recommendation list, whereas increasing k means considering more neighbour projects for recommendation.

As *success rate* was used as the only evaluation metric for LibRec [138], we exploit it to compare directly the performance of CROSSREC with that of LibRec. To answer this question, we performed a series of experiments

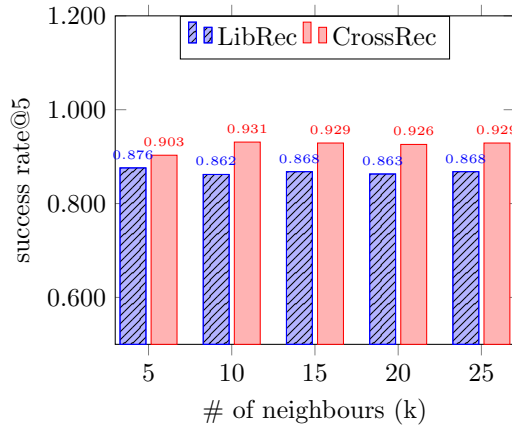
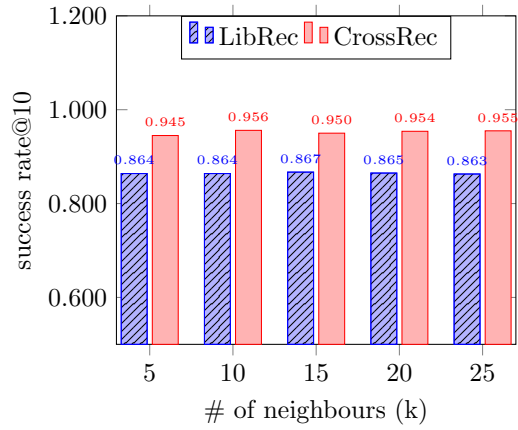
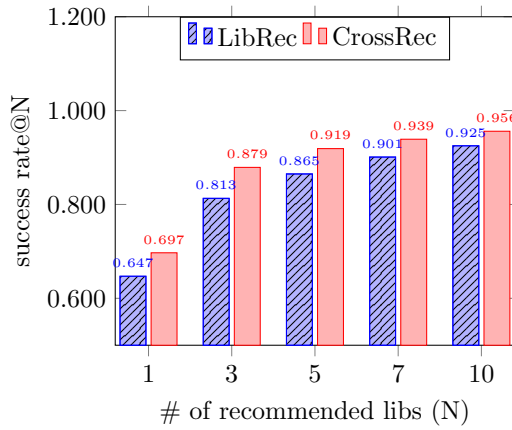
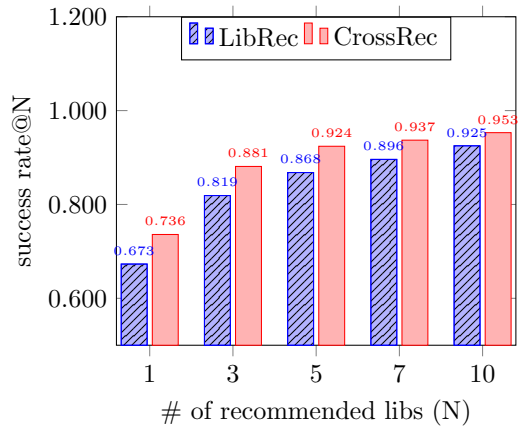
(a) Success rate@5, $k=\{5,10,15,20,25\}$ (b) Success rate@10, $k=\{5,10,15,20,25\}$ (c) Success rate@{1,3,5,7,10}, $k=10$ (d) Success rate@{1,3,5,7,10}, $k=20$

Figure 20: Success rate.

using different combinations of number of recommended libraries i.e., N , and number of neighbor projects exploited in the recommendation phase i.e., k . Varying N means changing the length of the recommendation list, whereas increasing k means considering more neighbour projects for recommendation.

Figure 20(a) shows the *success rate@5* for $k=\{5, 10, 15, 20, 25\}$. As can be seen there, the success rates obtained by CROSSREC are always superior to those of LibRec. The maximum *success rate@5* of LibRec is 0.876, whereas CROSSREC obtains success rates being greater than 0.903 for all configurations, with 0.931 being the maximum value. Figure 20(b) shows the *success rate@10*, for this setting, LibRec gains a comparable performance for $N = 5$. Meanwhile, CROSSREC gets a slight improvement in its performance compared to the case with $N = 5$. It is evident that CROSSREC outperforms LibRec in all test configurations. Figure 20(a) and 20(b) imply that changing the number of neighbour projects k does not make a big difference in their match rate as *success rate@N* is stable towards k for both systems.

Next, we investigate the success rate with regards to N . We consider a small number of recommended items, i.e., $N = \{1, 3, 5, 7, 10\}$. In practice, this means that the developer wants to see a short list of recommended libraries. In the first experiment, k is fixed to 10 and the outcomes are depicted in Figure 20(c). For $N = 1$, LibRec gets a success rate of 0.647 which is lower than the corresponding value 0.697 produced by CROSS-

	Success Rate	Accuracy		Sales Diversity		Novelty
N	Success Rate	Precision	Recall	Coverage	Entropy	EPC
3	0.02	1.58e-16	5.41e-08	0.0005	6.50e-05	4.33e-05
5	0.02	1.35e-24	9.10e-12	0.001	6.50e-05	4.33e-05
10	0.17	1.07e-21	9.12e-12	0.003	4.33e-05	1.52e-04
15	0.002	1.10e-14	–	0.003	6.50e-05	2.06e-04

Table 6: Wilcoxon rank sum test adjusted p -values for $N=\{3,5,10,15\}$, $k=10$.

	Success Rate	Accuracy		Sales Diversity		Novelty
N	Success Rate	Precision	Recall	Coverage	Entropy	EPC
3	0.70 (l)	0.18 (s)	0.12 (n)	0.92 (l)	0.96 (l)	1.00 (l)
5	0.74 (l)	0.23 (s)	0.16 (s)	0.86 (l)	0.98 (l)	1.00 (l)
10	0.93 (l)	0.22 (s)	0.16 (s)	0.80 (l)	1.00 (l)	0.94 (l)
15	0.58 (l)	0.17 (s)	–	0.80 (l)	0.98 (l)	0.90 (l)

Table 7: Cliff's d results for $N=\{3,5,10,15\}$, $k=10$. Labels in parenthesis indicate the magnitude (n:negligible, s:small, l:large).

REC. A success rate of 0.697 implies that CROSSREC can supply relevant recommendations to the developer at an encouraging match rate, even when she expects only an extremely brief list. Once k is changed from 10 to 20, both systems have a slight increase in *success rate@1* as depicted in Figure 20(d). However for other values of N , there are almost no changes in success rate. To further observe this behavior, we conducted more experiments with an increasing k , e.g., $k = \{50, 60, 100\}$. Nevertheless, the outcomes of these experiments are omitted from the paper due to space limitation. We noticed that considering more similar projects for recommendation does not improve success rate.

Last, but not least, the second column of Table 6 and of Table 7 report Wilcoxon rank sum test adjusted p -values and Cliff's d , respectively for the comparison of LibRec and CROSSREC in terms of *success rate*, using $k = 10$ and $N = \{3, 5, 10, 15\}$. As the tables indicate, the differences are always statistically significant and in favor of CROSSREC (effect size is positive), with a large effect size.

In summary, we see that CROSSREC significantly outperforms LibRec in all considered test configurations concerning *success rate*, with a large effect size. The recommendation time for a fold (120 projects) is relatively faster for CROSSREC (3s) than for LibRec (20s).

RQ₂: *How well can LibRec and CrossRec recommend third-party libraries in terms of accuracy, sales diversity, and novelty?*

Accuracy: to represent *accuracy*, we vary N (the cut-off value for the list of items to be recommended) from 1 to 30 to get *precision@N* and *recall@N* [99]. The rationale behind the selection of 30 as the maximum cut-off value is that LibRec normally produces a short list of recommendations, ranging from 35 to 50 items. The Precision-Recall curves (PRCs) for all 10 rounds of validation and different values of k are depicted in Figure 21(a) ÷ 22(b). Since a PRC closed to the upper right corner represents a better accuracy [38], we see that with LibRec, changing the number of neighbour k almost makes no difference in its accuracy. Meanwhile by CROSSREC, an increase of k brings a slightly better accuracy for some testing folds, however the gain is marginal. It is evident that for all pieces of testing data, CROSSREC always produces a superior accuracy compared to that of LibRec.

Sales Diversity: the catalog coverage scores for LibRec and CROSSREC are depicted in Table 8. The maximum coverage values are 4.594 and 5.897 for LibRec and CROSSREC, respectively. According to Eq. (17), a higher

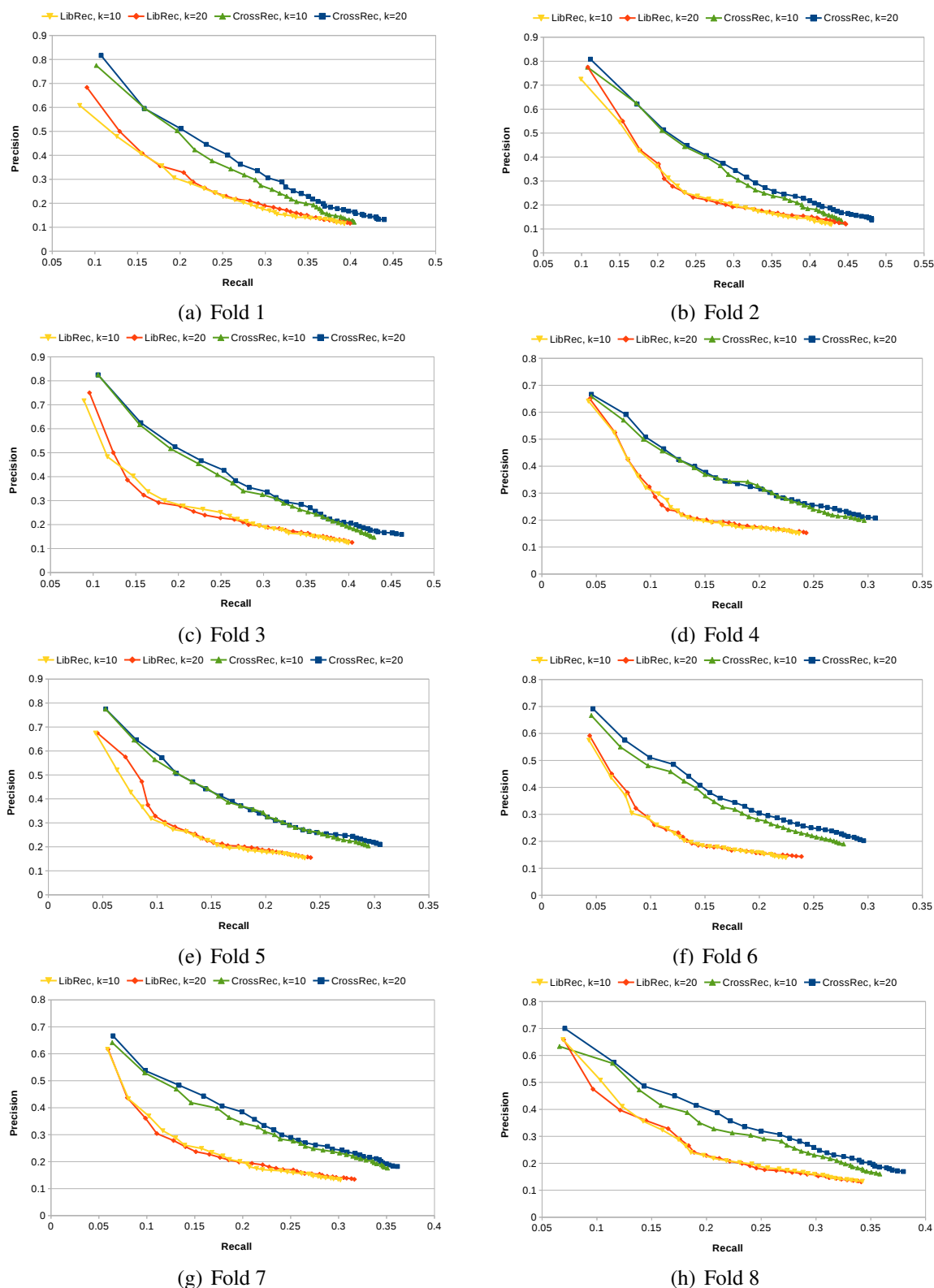
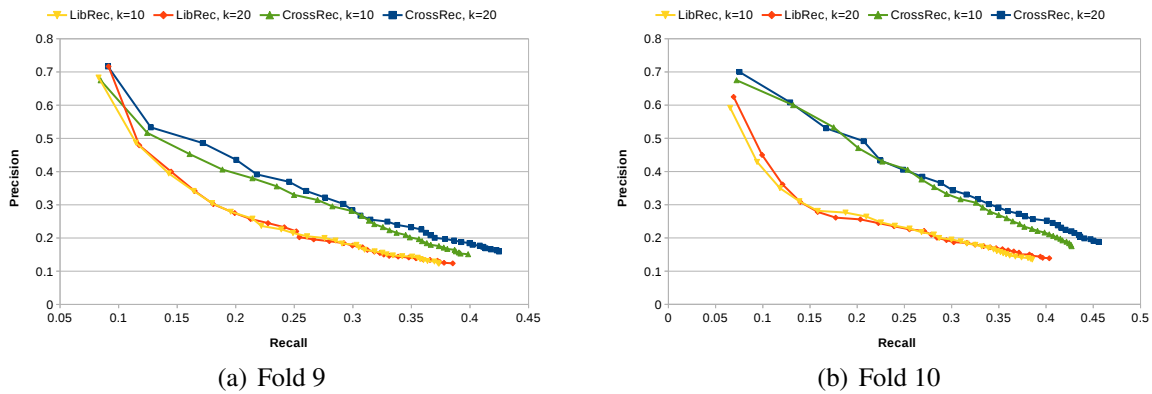


Figure 21: Accuracy: precision@N and recall@N, $k=\{10,20\}$.

Figure 22: Accuracy: precision@N and recall@N, $k=\{10,20\}$.

score means a better coverage. In this sense, the recommendations generated by CROSSREC cover a wider spectrum of libraries than those by LibRec for both configurations, i.e., $k = 10$ and $k = 20$, using different cut-off values N . Table 9 shows the *entropy* for LibRec and CROSSREC. Equation (18) suggests that a low entropy value represents a better distribution of items, therefore the recommendations by CROSSREC have a much better distribution than those obtained by LibRec. For example, for the case $N = 25$ and $k = 20$, CROSSREC has an entropy of 0.635, which is much better than 2.751, the corresponding value by LibRec.

	k=10		k=20	
N	LibRec	CROSSREC	LibRec	CROSSREC
5	0.857	1.099	0.691	0.814
10	1.760	2.157	1.346	1.534
15	2.675	3.278	1.937	2.312
20	3.577	4.541	2.512	3.143
25	4.594	5.897	3.139	4.005

Table 8: Catalog coverage for $N=\{5,10,15,20,25\}$, $k=\{10,20\}$.

	k=10		k=20	
N	LibRec	CROSSREC	LibRec	CROSSREC
5	0.869	0.239	0.552	0.127
10	1.752	0.481	1.098	0.254
15	2.653	0.723	1.639	0.381
20	3.566	0.968	2.193	0.508
25	4.500	1.217	2.751	0.635

Table 9: Entropy for $N=\{5,10,15,20,25\}$, $k=\{10,20\}$.

Novelty: the $EPC@N$ scores for LibRec and CROSSREC are shown in Table 10. With LibRec, changing k from 10 to 20 decreases *novelty* for all cut-off values. For example, the *novelty* with $N = 25$ and $k = 10$ is 0.349, however once k is changed to 20, it drops to 0.261. For CROSSREC, changing the number of neighbours k from 10 to 20 brings a rise in *novelty*, except for $N=10$. As shown in Tab. 10 CROSSREC obtains always scores that are higher than those of LibRec. For example, when $N = 5$ and $k = 20$, CROSSREC gets 0.292 as its *novelty*, whereas LibRec gets 0.114. This implies that CROSSREC recommends libraries that are closer to the long tail than LibRec can do.

N	k=10		k=20	
	LibRec	CROSSREC	LibRec	CROSSREC
5	0.187	0.291	0.114	0.292
10	0.264	0.349	0.166	0.344
15	0.296	0.376	0.204	0.377
20	0.320	0.391	0.236	0.399
25	0.349	0.401	0.261	0.416

Table 10: EPC for $N=\{5,10,15,20,25\}$, $k=\{10,20\}$.

Also in this case (see columns 3-7 of Tables 6 and 7), our analyses are supported by statistical procedures. Differences are always statistically significant. The effect size is negligible/small for *accuracy* (Precision and Recall), whereas it is large for all other indicators *sales diversity* and *novelty*.

The results in Figure 6.7 and Tables 8, 9, 10 indicate that CROSSREC significantly outperforms LibRec concerning *accuracy*, *sales diversity*, and *novelty*, with a small/negligible effect size for *accuracy* and large elsewhere.

RQ₃: *What are the reasons for the performance difference between LibRec and CrossRec?*

We attempt to ascertain why CROSSREC outperforms LibRec. This task may necessitate further investigations, both qualitative and quantitative research. However, by carefully studying the internal design of LibRec, we found out that the improvement attributes to the following facts. In the first place, CROSSREC employs a completely different approach to represent projects and libraries: it encodes the relationships among them into a graph. Second, to compute the similarity between two projects, CROSSREC assigns a weight to every library node using tf-idf (see Eq. (5)). In this way, the level of importance of a node is disproportional to its popularity. This is similar to the context of document matching where popular terms are given a low weight [52]. For instance, in Figure 3, lib_1 is a popular node since it is referred by 4 projects and this makes it have a low weight. As a result, CROSSREC is able to better capture the similarity between two projects compared to LibRec, which equally treats all libraries. Third, LibRec employs a very simple collaborative-filtering technique, though it also considers a set of k-nearest neighbor similar projects for finding libraries, it neglects their similarity level by considering all projects in the same way. Also, the technique assigns more weight to popular libraries without considering the degree of similarity between projects, from where the libraries come. This explains why LibRec recommends very popular items. In contrast, CROSSREC improves by assigning a larger weight to libraries that come from highly similar projects (see Eq. (12)). In other words, given a project, CROSSREC is able to “mimic” the behaviour of highly similar projects, it attempts to suggest a comparable set of libraries. Lastly, LibRec exploits association rule mining which indeed mines items that co-exist. This is why the coverage of the recommended items is low compared to that by CROSSREC.

Our qualitative analysis suggests that the improvements achieved by CROSSREC with respect to LibRec are due to the weighting scheme being applied, which also considers the projects’ similarity, i.e., it rewards recommendation of libraries from similar projects.

6.8 Threats to Validity

We identify the threats that may adversely affect the validity of the experiments, and the countermeasures taken to mitigate them. In particular, we focus on internal and external threats to validity as discussed below.

Threats to internal validity are related to any factors internal to our study that can influence our results. In the performed experiments, we did not consider the version number of project libraries. Even though this is a limitation of the current implementation of CROSSREC, also LibRec neglects library versions and consequently the performance comparison between the two systems has not been affected.

CROSSREC performance can depend on the values of N and k . We showed in the paper results for $k = 10, 20$, and for $N = 3, 5, 10, 5$ (see more in Section 6.9). Results for other values of N and k are consistent with what we already found.

Threats to external validity concern the generalizability of our findings. In the data collection phase, we tried to cover a wide range of possibilities by mitigating also the fact that many repositories in GitHub are of low quality, which is especially true when they do not have many stars. The set of 1,200 GitHub projects was randomly created by obtaining the following distribution of stars: 14 projects have 0 stars, 135 projects have [1-4] stars, 66 projects have [5-9] stars, 512 projects have [10-99] stars, 300 projects have [100-499] stars, 78 projects have [500-999] stars, and 95 projects have more than 1000 stars. Moreover, the number of libraries that a project in the considered dataset includes varies considerably from 10 to more than 500.

6.9 Discussions

By performing experiments with LibRec and CROSSREC on the same dataset, and by applying the same experiment settings, we were able to compare their performance in a thorough manner. We have seen that an increase in the number of neighbor projects considered for recommendation from $k = 10$ to $k = 20$ does not make a big distinction in accuracy for both systems. Furthermore, as there are no changes in success rates by increasing k , we can conclude that almost all relevant libraries are concentrated on the top most similar projects. This is further enforced by the fact *entropy* is improved for both systems when k is increased from 10 to 20. The inclusion of more projects brings various libraries, which helps increase the item distribution. With LibRec, the fact that *novelty* deteriorates when k increases shows that the additional projects bring only popular libraries. Meanwhile *novelty* does not change with CROSSREC using the same setting with k . This indicates that the recommended libraries brought by the additional projects do not help improve the overall *novelty*.

In this sense, we see that the ability to compute similarities among projects plays an important role in obtaining a good recommendation performance. In addition, since considering more neighbors means adding more rows to the user-item ratings matrix, which indeed increases the computational complexity, we anticipate that utilizing an appropriate value of k can help speed up the computation, thus increasing the overall efficiency, but still preserving a decent effectiveness.

In contrast to LibRec, CROSSREC is able to maintain a trade-off between *accuracy* and *sales diversity*, it gains better precisions and recalls for all testing folds. Furthermore, CROSSREC also gets an adequate catalog coverage and novelty by recommending more unpopular libraries to projects.

6.10 Conclusions and Future Work

Third-party libraries contain tailored and well-defined functionalities which in turn offer a useful resource to software projects being developed. Making use of such libraries allows developers to leverage an existing infrastructure, without reinventing the wheel. In this way, recommending third-party libraries to developers help them save time as well as increase productivity. We implemented CrossRec, a novel approach to library recommendation that relies on a collaborative-filtering recommender system. The approach has been evaluated by

considering different quality metrics and a dataset consisting of 1,200 Java projects. The evaluation demonstrated that our approach outperforms LibRec, a well-known system for library recommendation with regards to various quality indicators. Furthermore, CrossRec is more efficient as it produces recommendations in a rather short time. To the best of our knowledge, our work is the first one that employs graphs to represent the relationships among software projects so as to effectively compute similarity and eventually to recommend libraries. Among other characteristics, we found out that the novelty of the outcomes is important in the context of library recommendation, as very unpopular items, i.e., those belong to the long tail, are also useful.

Concerning the experimental settings presented in the paper, we suppose that feeding CrossRec with more data as query should certainly improve the overall performance. However, this is a mere assumption and necessitates additional investigations, which can be considered as an open research issue.

The deployment of various quality metrics to study the systems' performance has shown to be meaningful. Though these metrics have been widely used to evaluate recommender systems [122],[59],[128], to the best of our knowledge, the current work is the first one that exploits them to examine the performance of a system for recommending third-party libraries. Apart from *success rate*, we also incorporate other metrics to investigate if the approaches obtain a good performance, i.e., *accuracy*, *sales diversity*, and *novelty*. Each of these metrics reflects a different view on the recommendations, which helps thoroughly study the final outcomes. For future work, among others we will investigate in detail the importance of Novelty and Sales Diversity in the recommendation outcomes.

The CrossRec tool has been successfully integrated into Eclipse and provided to developers as an IDE prompting instant recommendations. We are working to equip our tool with the ability to recommend different artifacts, such as API function calls and code snippets, as well as to extract background data from various sources, such as Eclipse. We plan also to apply the proposed approach to other ecosystems based on different languages such as C++, and C#.

Last but not least, we plan to compare CrossRec with other systems such as LibCUP [127] and LibFinder [105]. These are approaches that exploit clustering techniques to identify and recommend library co-usage patterns or incorporate semantics into the recommendation process. Such a comparison should allow for understanding the strengths and weaknesses of each approach, thereby helping developers select the one that best fits their need.

7 Recommendation of StackOverflow Posts

During the development of complex software systems, programmers look for external resources to understand better how to use specific APIs and to get advice related to their current tasks. StackOverflow provides developers with a broader insight of API usage and with useful code examples. However, finding StackOverflow posts that are relevant to the current context is a strenuous task [119]. In this section, we introduce SOrec, an approach that allows developers to retrieve messages from StackOverflow being relevant to the API function calls that they have already defined, as well as to the external libraries included in the project being developed.

The approach has been validated by means of a user study involving 11 developers who evaluated 500 posts with respect to 50 contexts. Experimental results indicate the suitability of SOrec to recommend relevant StackOverflow posts and concurrently show that the tool outperforms a well-established baseline.

7.1 Overview

Developing complex software systems requires mastering several languages and technologies [121]. Consequently, software developers need to continuously devote effort to understand how to use new third-party libraries even by consulting existing source code or heterogeneous sources of information. The time spent on discovering useful resources can have a dramatic impact on productivity [40].

A recent work shows that StackOverflow (SO) [2] is the most popular question-and-answer website [8], which is a good source of support for developers who seek for probable solutions from the Web [3, 72]. SO discussion posts provide developers with a broader insight of API usage, and in some cases, with sound code examples. Nevertheless, as the information space is huge, it is necessary to have tools that help narrow down the search scope as well as find the most relevant documentations [121]. However, how to construct a query that best describes the developer's context and how to properly prepare SO data to be queried are still challenging tasks [113]. In particular, there is a need to enhance the quality of retrieved posts as well as to refine the input context to generate decent queries, which then facilitate the search process.

We propose SOrec, a comprehensive approach imposing various measures on both the data collection and query phases. To improve efficiency, we exploit Apache Lucene [1], an information retrieval library to index the textual content and code coming from StackOverflow. Posts are retrieved and augmented with additional data to make them more exposed to queries. On the other side, we boost the context code with different factors to construct a query that contains information needed for matching against the stored indexes. In a nutshell, we make use of *multi facets* of the data available at hand to optimize the search process, with the ultimate aim of recommending highly relevant SO posts. The approach's performance has been assessed by considering various experimental settings. We also compare our tool against a well-established baseline, i.e., FaCoY [62]. Through a series of user studies, we demonstrate that our proposed approach considerably improves the recommendation performance, and thus outperforming the considered baseline. In this sense, we have the following contributions:

- Identification of augmentation measures to automatically refine the considered input SO dump by considering various pieces of information;
- Exploiting a well-founded IR tool to index the augmented data;
- Characterizing the context code by automatically boosting the constituent terms to improve their exposure to the indexed data and eventually build a proper query in a transparent manner for the developer;
- Two empirical evaluations of the proposed approach to evaluate the performance of SOrec and to compare it with FaCoY.

This chapter is structured into the following sections. Section 7.2 provides background and describes the motivations for our work. In Section 7.3, we introduce SOrec, our proposed approach to recommend StackOverflow posts. The evaluation is presented in Section 7.4. Section 7.5 analyzes the experimental results. Finally, Section 7.6 discusses the threats to validity.

7.2 Background and Motivations

Over the last decade several approaches have been conceived to leverage the use of crowdsourcing in software engineering [81]. Those exploiting StackOverflow as main source of information (e.g., [36, 62, 111, 113, 152]) can be distinguished in two main categories:

- C1. approaches that pay attention only to the automated creation of queries to be executed by search engines, and to the visualization of the retrieved posts according to some ranking model (e.g., [36, 111, 113]);
- C2. approaches that focus both on query creation and on advanced indexing mechanisms specifically conceived for storing and retrieving SO posts (e.g., [62, 152]).

Prompter [113] is among the most recent approaches falling in the first category above. It is an automatic tool, which is used to recommend SO posts given an input context built from source code. Prompter performs various processing steps to produce a query. First, it splits identifiers and removes stop words. Then it ranks the terms according to their frequency by considering also the entropy of the term in the entire SO dump. Once the query is built, the tool exploits a web service to perform the query via the Google and Bing search engines. Finally, a ranking model is employed to sort the results according to different metrics such as API similarity, tags analysis, and SO answers and questions.

FaCoY [62] is a recent code-to-code search engine that relies on Apache Lucene and provides developers with relevant GitHub code snippets. Two main phases are conducted to produce recommendations as follows. The first one is performed on the context code to get related SO posts from a local indexed dump. To this end, the system parses the context code and builds an initial query q_c to look for posts from StackOverflow. From the set of retrieved posts, it parses natural language descriptive terms from questions to match against the question index of Q&A that has been built ex-ante in order to get more posts that contain relevant source code. Afterwards, a new query q'_c is formed from the newly obtained source code. The second phase is done on q'_c to search from GitHub for more snippets, which are finally introduced to developers. By focusing on the first phase, i.e., searching for SO posts by exploiting the input context code, FaCoY can be considered in category C2 above. This module works like a bridge between the initial query q_c and the final results. In this sense, it has an important role to play since its performance considerably affects the final recommendation outcomes.

The experimental results [62] demonstrate that FaCoY obtains a superior performance with regards to several baselines. In this section we pay our attention only to the FaCoY's module for searching SO posts. By a careful observation on the system, we found out that it suffers a setback with respect to incomplete data as well as brief input query. As we can see later on in this chapter, for many queries the system is unable to retrieve any SO posts, or for some contexts it suggests irrelevant ones, i.e., false positives. To this end, we believe that there is a need to overcome the limitations, so as to enhance the overall performance of FaCoY.

As an example, we consider the explanatory code snippet shown in Listing 1. The code declares a `CamelContext` variable, and invokes functions `addRoutes()` and `configure()`. The input code is pretty simple, and the developer would benefit from being suggested with SO posts discussing aspects related to the input source code.

```
1  package camelinaction;
2
3  import org.apache.camel.CamelContext;
4  import org.apache.camel.builder.RouteBuilder;
5  import org.apache.camel.impl.DefaultCamelContext;
6
7
8  public class FilePrinter {
9
10     public static void main(String args[]) throws Exception {
11         // create CamelContext
12         CamelContext context = new DefaultCamelContext();
13
14         // add our route to the CamelContext
15         context.addRoutes(new RouteBuilder() {
16             public void configure() {
17
18             }
19         });
20     }
21 }
22
```

Listing 1: Explanatory input context code.

In other words, it is expected that a search engine can recommend discussions that are relevant to the developer context, for instance the post¹⁶ shown in Figure 23. In the figure, we only capture the key information from the post, i.e., title, question, answer, code and display it in the figure. The post contains two answers and one of them is useful for the context. For instance, the depicted snippet contains class `CurrencyRoute` where `addRoutes()` is informative and `configure()` is completely defined. More importantly, this code shows that some additional packages are required, e.g., `ActiveMQConnectionFactory` or `JmsComponent`. In this sense, the accompanying snippet is handy for supporting the development of the context code in Listing 1 since it provides a better insight into how to use the related API.

When the context code shown in Listing 1 is fed to FaCoY, the system fails to return any results. We anticipate that this is due to the lack of input data, i.e., the query code is very brief, and to the indexing process of FaCoY, which ignores some important components in source code when preparing the indexed data. Thus, we believe that there is still room for improvement. In this respect, Lucene offers a well-defined platform for managing and indexing data. However, it is incumbent upon the Lucene user to decide which data to index and which data to use as query. To this end, we propose an approach to improve the module for searching SO posts of FaCoY. We also attempt to perform various refinement steps on the input SO dump as well as to polish the query code. The details of the proposed approach are given in the next section.

7.3 Proposed Approach

Given a user context with code being developed, we aim at searching for posts that contain highly relevant answers from StackOverflow. We attempt to overcome the limitations of the existing approaches by properly indexing SO data and by processing the query by developers' side, exploiting various refinement techniques. In particular, we come up with a comprehensive approach named SOrec, which takes into consideration three consecutive phases, i.e., Index Creation, Query Creation, and Query Execution. By Index Creation, we parse and organize an SO dump into a queryable format to facilitate future search operations. Query Creation is done at the developer's side to transform the current context into an informative query that can be used to search

¹⁶<https://tinyurl.com/yydp8lwd>

XML to object in Apache Camel

I'm trying to fetch www.dnb.no/portalfont/datafiles/miscellaneous/csv/kursliste_ws.xml and convert it to a Java object using xstrem.

0 I get this error:

Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/camel/spi/DataFormatName Caused by: java.lang.ClassNotFoundException: org.apache.camel.spi.DataFormatName

2 Answers

active oldest votes

This is the working route:

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.jms.JmsComponent;
import org.apache.camel.impl.DefaultCamelContext;
import org.apache.log4j.BasicConfigurator;

import javax.jms.ConnectionFactory;

public class CurrencyRoute {

    public static void main(String args[]) throws Exception {
        // Log 4j
        BasicConfigurator.configure();

        // Create camel context
        CamelContext context = new DefaultCamelContext();

        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://
context.addComponent("test-jms", JmsComponent.jmsComponentAutoAcknowledge(c
// New route
context.addRoutes(new RouteBuilder() {
    public void configure() {

        from("quartz://myTimer?trigger.repeatCount=0")
        .log("### Quartz trigger ###")
        .to("direct:readFile");
    }
});
    }
```

Figure 23: A StackOverflow post that is relevant to Listing 1.

against the indexed data. Concerning Query Execution, the actual searching is performed by means of Apache Lucene.

Index and query creations rely on the JDT parser, which allows one to define different types of constructs to be parsed from the source as a parameter. In particular, there are the following options:

- `Compilation_Unit`: the source is parsed as a compilation unit.
- `Class_Body_Declarations`: the source is parsed as a sequence of class body declarations.
- `Expression`: the source is parsed as a single expression.
- `Statements`: a constant is used to specify that the source be parsed as a sequence of statements.

This list is ordered according to the information that they compute and the severity of the parser option, e.g., the `Compilation_Unit` option allows one to get more information than the others but it is less resilient to malformed statements.

An overview of the SOrec building components is depicted in Figure 24. The three constituting phases, i.e., Index Creation, Query Creation, and Query Execution are described in detail by the following subsections.

7.3.1 Index Creation

Starting from an SO dump, the original data is loaded into MongoDB for further processing. Then, the data is parsed and transformed into a format that can be queried later on. At this point, it is necessary to use indexed

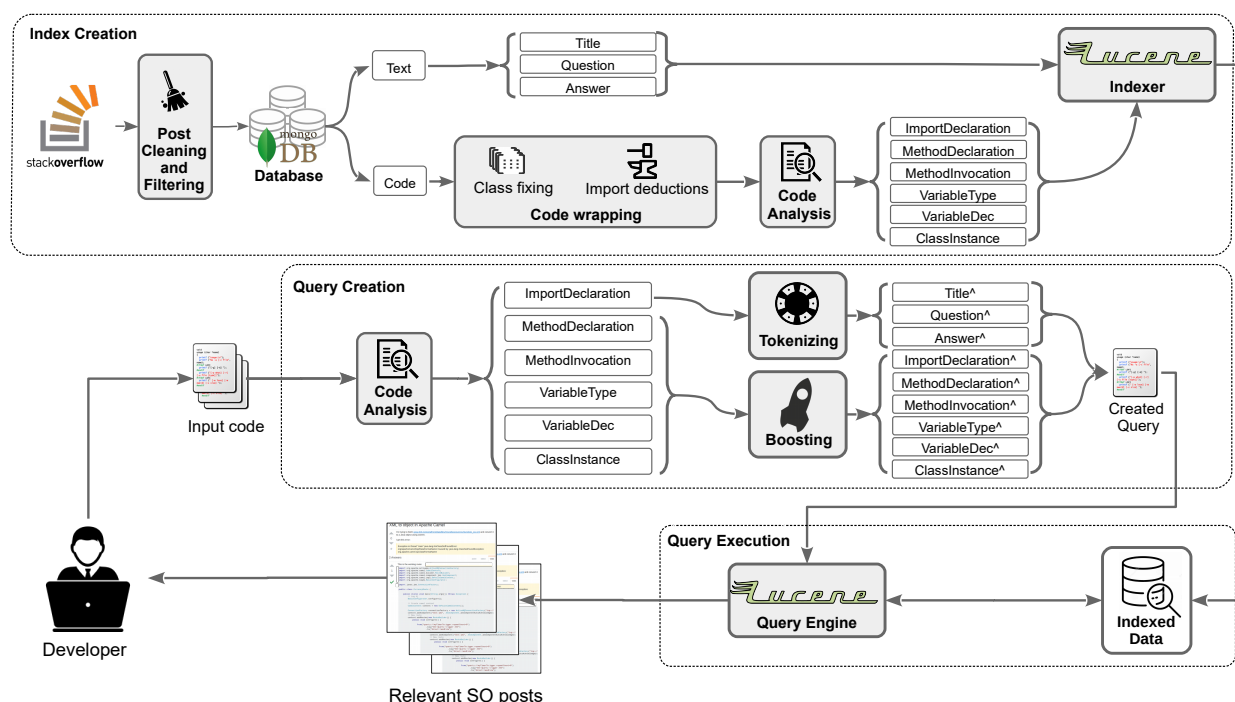


Figure 24: The SOrec architecture.

data for future look up. In this work, we opt for Apache Lucene as it is a powerful IR tool being widely used to manage and query vector data. For each SO post, the following components are excerpted: Title, Body and Code. Concerning the textual part, we extract questions, answers, and titles and index them by means of the Indexer. Meanwhile, code contents are parsed to extract useful artifacts before being fed to Lucene. In particular, the JDT parser is used to obtain six tokens as shown in Table 11. However, code snippets in StackOverflow posts may neither be complete nor compilable [133], and therefore they might not be found if being queried in their original format. Thus, to make them compilable, we propose two refinement steps, namely *Post cleaning and filtering* and *Code wrapping* as explained below.

Post cleaning and filtering In SO messages, a question is typically followed by answers and comments. However, many posts do not have any answers at all. Such these posts are considered not useful for recommendation tasks. Thus, we filter out irrelevant posts as well as remove the low-quality ones first by considering only those that have accepted answers. Then, only posts that contain the code tag¹⁷ to include in their bodies source code written in Java are accepted for further processing.

Code wrapping To deal with incomplete and uncompileable snippets, we use the parsing option that yields more tokens. Furthermore, a default class wrapping is added to those snippets that cannot be parsed. For code snippets without any imports, we wrap up with relevant classes to make them more informative. To this end, we exploit an archive provided by *Benellallam et al.* [13]. The dataset contains more than 2.8M artifacts together with their dependencies as well as other relationships, e.g., versions. We count the frequency that each

¹⁷We are aware that there are SO posts that do not always use the `code` tag to include inline source code. Thus, relying only on such a tag might discard messages that instead should be kept. Natural Language Processing techniques can be exploited to make the employed cleaning and filtering phase less strict even though we defer this as future work.

artifact is invoked, and sort into a ranked list in descending order. Then only top N class canonical names, i.e., the ones appear in an import statement, are selected. By parsing all API calls within a method declaration, we trace back to their original packages from the top N names. Nevertheless, given that a class instance is invoked without any declaration, more than one canonical name could be found there. In this case, we compute the Levenshtein distance from each name to the title and body text of the post and use it as a heuristic to extract the best-matched one. Finally, the corresponding import directives are placed at the beginning of the code.

```

1  ASTParser parser = ASTParser.newParser(AST.JLS9);
2  parser.setResolveBindings(true);
3  parser.setKind(ASTParser.K_COMPILATION_UNIT);
4  parser.setSource(snippet.toCharArray());
5  Hashtable<String,String> options = JavaCore.getOptions();
6  options.put(JavaCore.COMPILER_DOC_COMMENT_SUPPORT, JavaCore.ENABLED);
7  parser.setCompilerOptions(options);

```

Listing 2: Original code snippet.

We consider an example as follows. Listing 2 depicts a code snippet extracted from SO. The code contains just function calls, and it is incomplete since there is no class declaration and import. If we use this code without any refinement to index Lucene, it might not be unearthed by the search engine due to the lack of data. Thus, it is necessary to wrap up and augment it with additional information. By adding class fix and import directives, we obtain a new code snippet as shown in Listing 3. The resulted snippet resembles a real hand-written code, which probably facilitates the matching process later on.

```

1  import java.util.HashMap;
2  import java.util.Hashtable;
3  import org.eclipse.jdt.core.JavaCore;
4  import org.eclipse.jdt.core.dom.AST;
5  import org.eclipse.jdt.core.dom.ASTParser;
6  import org.eclipse.jdt.core.dom.ASTVisitor;
7
8  public class fix(){
9      ASTParser parser = ASTParser.newParser(AST.JLS9);
10     parser.setResolveBindings(true);
11     parser.setKind(ASTParser.K_COMPILATION_UNIT);
12     parser.setSource(snippet.toCharArray());
13     Hashtable<String, String> options = JavaCore.getOptions();
14     options.put(JavaCore.COMPILER_DOC_COMMENT_SUPPORT, JavaCore.ENABLED);
15     parser.setCompilerOptions(options);
16 }

```

Listing 3: Augmented code snippet.

Once the refinement steps have been done, we index all terms corresponding to the tokens specified in the Code part of Table 11 and store them into Lucene for further look up.

7.3.2 Query Creation

This phase is conducted on the client side, and the method declaration being developed is used as input context. A query can be formed by considering all terms extracted from the context code. It is evident that each term in posts has a different level of importance. Thus, the second phase is to equip a query with more information that better describes the current context, taking into account the terms' importance level. Fortunately, Lucene supports *boosting*, a scoring mechanism to assign a weight to each indexed token. Based on scoring, we perform two augmentation steps, i.e., *Boosting*, and *Tokenizing* as follows.

	Token	Description
Text	Title	The title of the post
	Answer	All answers contained in the post
	Question	The question
Code	ImportDeclaration	The directives used to invoke libraries
	MethodDeclaration	Method declarations with parameters
	MethodInvocation	API function calls
	VariableType	Types of all declared variables
	VariableDec	All declared variables
	ClassInstance	Class declarations

Table 11: The used facets.

Boosting The original code is parsed to obtain the six tokens listed in the last half of Table 11. Each term in the code is assigned a concrete weight to boost the level of importance. Entropy [132] is exploited to compute the quantity of information of a document using the following formula:

$$H = - \sum p(x) \log p(x) \quad (20)$$

where $p(x)$ is the probability of term x . An entropy value ranges from 0 to $\log(n)$, where n is the number of terms within the document. We compute entropy for all terms in the original source code and rank them in a list of descending order. Then the first quarter of the list is assigned a boost value of 4. Similarly, the next 2nd, 3rd, and 4th quarters get the boost value of 3, 2, and 1, respectively. Finally, all the code terms are attached to their corresponding tokens to form the query.

Tokenizing By the Index Creation phase in Section 7.3.1, nine different tokens have been populated (see Table 11). Among them, there are 3 textual tokens, i.e., Title, Answer, and Question. However, by the developer's side, the input context contains just source code and there are no textual parts that can be used to match against the three tokens. Thus, given the input code, we attempt to generate textual tokens by exploiting the import directives embedded at the beginning of each source file. Starting from an import directive, we break it into smaller pieces and attach them to all the textual tokens. A previous work [19] shows that in an SO post, the title is more important than the description. In particular, according to [19] the importance ratio between description and title of a given post is 1/3. Accordingly, we set a boost value of 4 to the title and 1.4 to both the answer and question.

By considering the code in Listing 1, the query that SOrec creates after the boosting and tokenizing phases is shown in Listing 4.

Listing 4: Sample query produced after the *Boosting* and *Tokenizing* phases.

```
VariableDeclarationType: CamelContext^1.0 OR
VariableDeclaration: context^1.0 OR
MethodInvocation: addRoutes^1.0 OR
ClassInstance: DefaultCamelContext^1.0 OR
ImportDeclaration: org.apache.camel.impl.DefaultCamelContext^1.0 OR
ImportDeclaration: org.apache.camel.CamelContext^1.0 OR
ClassInstance: RouteBuilder^1.0 OR
MethodDeclaration: main^1.0 OR
ImportDeclaration: org.apache.camel.builder.RouteBuilder^1.0 OR
MethodDeclaration: configure^1.0 OR
Answer: apache^1.4 OR
```

Answer: camel^1.4 **OR**
Question: apache^1.4 **OR**
Question: camel^1.4 **OR**
Title: apache^4 **OR**
Title: camel^4

7.3.3 Query Execution

Queries that are created, as described in the previous section, are executed by means of Apache Lucene. Moreover, we exploit the Lucene built-in BM25 to rank indexed posts. In particular, BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of the inter-relationship between the query terms within a document, e.g., their relative proximity. The index is computed as given below.

$$R(q, d) = \sum_{t \in q} \frac{f_t^d}{k_1((1 - b) + b \frac{l_d}{avgl_d}) + f_t^d} \quad (21)$$

where f_t^d is the frequency of term t in document d ; l_d is the document d length; $avgl_d$ is the document average length along the collection; k is a free parameter usually set to 2 and $b \in [0, 1]$. When $b = 0$, the normalization process is not considered and thus the document length does not affect the final score. In contrast, when $b = 1$, the full-length normalization is performed. In practice, b is normally set to 0.75. It has been shown that for ranking documents, BM25 works better than the standard TF-IDF one [109].

By considering the query in Listing 4 as input, the post shown in Figure 23 is among the resulting ones.

In the following section we introduce two evaluations to examine if our proposed solution is beneficial to the matching of relevant SO posts.

7.4 Evaluation

To study SOrec's performance, we performed two main evaluations by means of user studies. The first one is done to evaluate the role of each augmentation proposed in Section 7.3. To this end, we consider 6 experimental configurations, i.e., **A**, **B**, **C**, **D**, **E**, **F** with 10 queries for each (see Table 12). The second evaluation compares SOrec against a baseline, namely the module for retrieving SO posts of FaCoY [62], and this corresponds to the last configuration **G**. For the sake of representation, from now on the baseline is addressed as FaCoY (unless otherwise stated), despite the fact that it is only a module of the whole FaCoY system presented in [62]. To thoroughly examine the difference in their performance, we consider 50 queries in **G**. The queries contain code snippets that invoke ten of the most popular Java libraries, i.e., Jackson, SWT, MongoDB driver, Javax Servlet, JDBC API, JDT core, Apache Camel, Apache Wicket, Twitter4j, Apache POI.

The test configurations are explained in Table 12, whose the 2nd to 5th columns specify the presence of the techniques mentioned in Section 7.3, with the corresponding section being shown in parentheses. For example, the 2nd column **Wrapping** is a combination of *class fixing* and *library import deduction* introduced in Section 7.3.1. To facilitate future research, we made available the SOrec tool together with related data in GitHub.¹⁸

¹⁸<https://github.com/ase2019-sorec/SOrec>

Conf.	Wrapping (Sec. 7.3.1)	Boosting (Sec. 7.3.2)	Tokenizing (Sec. 7.3.2)	Ranking (Sec. 7.3.3)	# of queries	Description
A	—	—	—	—	10	Flat queries, without considering any proposed augmentations
B	✓	—	—	—	10	Wrapping is introduced to queries in Conf. A
C	✓	—	—	✓	10	BM25 is used to rank the retrieved posts
D	✓	✓	—	—	10	Entropy is used to boost the queries in Conf. B
E	✓	—	✓	—	10	Transforming import directives to textual tokens for queries in Conf. B
F	✓	✓	✓	✓	10	Imposing all proposed augmentations
G	✓	✓	✓	✓	50	Imposing all proposed augmentations to compare SOrec and FaCoY

Table 12: Experimental configurations.

Name	Value	Name	Value
Size	70GB	# of answers	1, 122, 789
# of posts	18, 300, 672	# of acc. answers	552, 458
# of Java posts	757, 439	# of posts fixed	32, 578

Table 13: A summary of the SO dump used in the evaluation.

Score	Description
0	No results at all are returned
1	The post is totally irrelevant
2	The post contains some hints but it is still out of context
3	There are relevant suggestions but the key features are missing
4	The post provides proper recommendations, together with its code snippets it can help developers solve their current task

Table 14: Confidence Scores.

7.4.1 Dataset

To provide input for the evaluation, we exploited a recent StackOverflow dump¹⁹ which is an XML file of 70GB in size and contains more than 18 millions posts. By filtering with tags, we obtained 757,439 posts containing Java source code. The resulting set has more than 1,2 millions of answers with 49.20% of them being already accepted i.e., 552,458 answers. In such posts there were 32,578 snippets that did not have any imports. We fixed them as presented in Section 7.3.1. Eventually, we indexed and parsed all the posts following the paradigm described in Section 7.3.1. More details of the dataset used in our evaluation are shown in Table 13.

7.4.2 User studies

We resort to user study as this is the only way to investigate whether the recommendation outcomes are really helpful to solve a specific task [77, 82, 113]. A group of 11 developers was asked to participate in the user study. Six participants are master students attending a Software Engineering course. Three of them are PhD students. The other two are postdoc researchers. Through a survey sent to each participant, we found out that more than a half of them have at least 7 years of programming experience. Among these people, three participants have worked with programming for 15 years. All of them are capable of Java and at least another programming language, e.g., Python or C++. The evaluators use code search engines like GitHub, StackOverflow, or Maven on a daily basis. Furthermore, they frequently re-use code fragments collected from these external sources.

The user studies aim at evaluating the relevance of a recommended post and a code snippet. Given a query, each system, i.e., FaCoY or SOrec produces as outcome in the form a ranked list of relevant posts. To aim for a fair evaluation, we mixed the top-5 results generated by each system in a single Google form and present them to the evaluators together with the corresponding context code. This simulates a *taste test* [44, 110] where users are asked to give feedback for a product, e.g., food or drink, without having a priori knowledge about what is being addressed. This aims to eliminate any bias or prejudice against a specific system. Each pair of code and post, i.e., $\langle query, retrieved\ post \rangle$ is examined and evaluated by at least two participants using the scores listed in Table 14. Apart from 11 developers mentioned before, we also involved one more senior researcher to validate the evaluation outcome of every query. In case there is a disagreement between any two participants, e.g., the first person assigned the score of 2 and the second one gave 4 to a same pair, the senior researcher examines the pair again to eventually reach a consensus.

7.4.3 Evaluation metrics

As typically done in related work, the following metrics have been considered to evaluate the recommendation outcomes [77, 82]:

- *Confidence*: it is the score given to a pair of $\langle query, retrieved\ post \rangle$ following Table 14;
- *Success rate*: if at least one of the top-5 retrieved posts receives 3 or 4 as score, the query is considered to be relevant. *Success rate* is the ratio of relevant queries to the total number of queries;
- *Precision*: it is computed as the ratio of pairs in the top-5 list that have a score of 3 or 4 to the total number of pairs, i.e., 5.

¹⁹<https://archive.org/details/stackexchange>

7.4.4 Research questions

The evaluations are conducted to answer the following research questions:

- **RQ₄**: *Which experimental configuration brings the best SOrec performance?* We compare the flat configuration with the augmented ones to see which setting fosters the best recommendation outcome for SOrec.
- **RQ₅**: *How does SOrec compare with FaCoY?* Compared to FaCoY, SOrec is equipped with various refinement techniques. By answering this question, we ascertain whether our proposed augmentations are useful for searching posts in comparison to the original approach FaCoY.
- **RQ₆**: *What are the reasons for the performance difference?* We are interested in understanding the factors that add up to the performance difference between the two systems.

The following section analyzes the systems' performance by addressing these research questions.

7.5 Experimental Results

This section presents the results obtained from the experiments as well as related discussions. First, we analyze the outcomes obtained by performing SOrec with six configurations, i.e., **A** ÷ **F** (see Table 12), to answer **RQ₄**. Afterwards, we compare FaCoY with SOrec by answering **RQ₅**. Finally, we attempt to reason what constitutes the performance differences between the two systems by means of **RQ₆**.

RQ₄: *Which experimental configuration brings the best SOrec performance?*

Each configuration is evaluated using 10 queries and each of them corresponds to 5 posts, resulting in 50 pairs of *<query, retrieved post>*. We gather the confidence scores of each configuration and represent them in a violin boxplot as shown in Figure 25. According to *Hintze et al.* [50], a violin boxplot is a combination of boxplot and density traces which gives a more informative indication of the distribution's shape, or the magnitude of the density. Thus, it is evident that performing SOrec with flat queries, i.e., configuration **A**, yields the worst performance since the corresponding boxplot is slim and distributed along the vertical axis. This suggests that feeding queries without incorporating any proposed augmentations brings less relevant posts. Meanwhile, the system obtains a better performance for configurations **B** (flat query plus wrapping) and **C** (flat query plus wrapping and ranking) with respect to **A**. Moreover, the two configurations **B**, **C** contribute to a comparable performance as their corresponding violins have a similar shape. Among others, the best confidence is seen when running SOrec with **F**, i.e., all proposed augmentations are incorporated. In particular, no query pair gets 1 as the confidence value and most of them are assigned a value of 3 or 4. This necessarily means that augmenting queries with all the proposed measures helps retrieve highly relevant posts.

Metric	Configuration					
	A	B	C	D	E	F
Success rate	0.90	0.90	0.90	0.90	1.00	1.00
Precision	0.60	0.66	0.68	0.74	0.78	0.82

Table 15: Success rate and Precision.

We consider the success rate and precision scores for the configurations in Table 15. Running SOrec on the dataset always gets a minimum success rate of 0.90, regardless of the configuration. SOrec gains the maximum

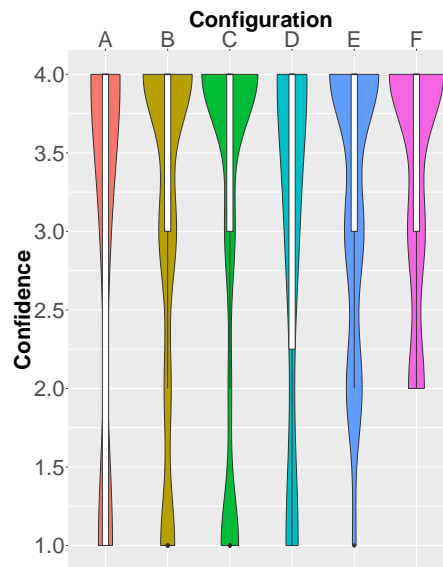


Figure 25: Confidence for configurations **A** ÷ **F**.

score, i.e., *success rate* = 1 when **E** or **F** is imposed. This demonstrates that Tokenizing (see Section 7.3.2) contributes much to the matching of relevant posts. Concerning *Precision*, we see that using flat query obtains the lowest precision, i.e., *Precision* = 0.6 and this is consistent with *Confidence* in Figure 25. Again, the best *Precision*, i.e., 0.82 is obtained when all augmentations are imposed on the queries.

In summary, running SOrec by deploying all proposed augmentations provides the best performance with respect to confidence, success rate, and precision.

RQ₅: How does SOrec compare with FaCoY?

To aim for a reliable comparison, we executed both FaCoY and SOrec on the dataset mentioned in Section 7.4.1. Considering the set of 50 queries, SOrec returns 250 pairs of query-post. Each pair gets a score ranging from 1 to 4. However, FaCoY does not find any results for 10 among the queries, i.e., the corresponding scores are 0 (see Table 14). We depict the confidence scores of both systems using violin boxplots in Figure 26(a). The boxplots demonstrate that SOrec gains considerably better confidence than that of FaCoY. In particular, SOrec has more scores of 3 and 4, whereas FaCoY has more scores of 1 and 2. By inspecting the ten queries that yield no results, we found out that their input context code is considerably short. This supports our hypothesis in Section 7.2 that FaCoY is less effective given that input data is incomplete or missing.

To aim for a more reliable comparison, we remove the ten queries from the results of both systems and sketch the confidence scores in Figure 26(b). For this set of queries, the FaCoY's violin fluctuates starting from 3 down to 1. In contrast, the majority of the violin representing SOrec lies on the upper part of the figure, starting from 3 in the vertical axis. We conclude that SOrec obtains a better confidence compared to FaCoY.

We further investigate the systems by considering Figure 26(c) where the precision scores for 40 queries are depicted. By this metric, the performance difference between the two systems becomes more noticeable. To be more concrete, a larger part of the FaCoY's boxplot resides under the median horizontal line, implying that most of the queries get a precision lower than 0.5. In the opposite side, SOrec gains better precisions that are larger than 0.5, and agglomerate to the upper bound, i.e., 1.0. The metric shows that, given the same query, SOrec returns more relevant posts than the baseline does.

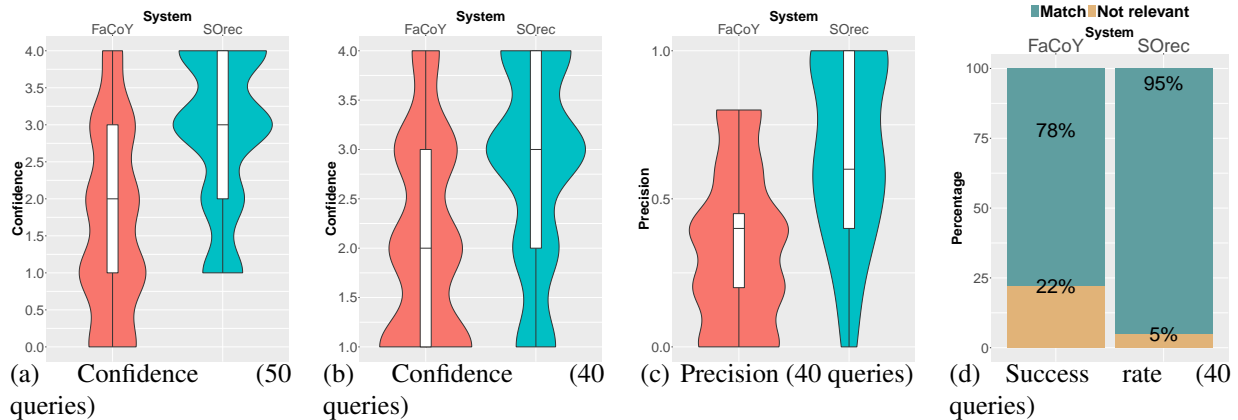


Figure 26: Confidence, precision, and success rate for Configuration G.

The obtained success rates for both systems are shown in Figure 26(d). Among 40 queries fed to FaCoY, 78% of them are successful, i.e., at least one pair of a query gets a value of 3 or 4. Meanwhile, SOrec achieves a better percentage of success, 38 among 40 queries are successful, yielding a success rate of 95%.

Finally, it is important to understand if the performance difference is statistically significant. We compute Wilcoxon rank sum test [148] on the scores obtained by the systems and get the following results: p-value for Confidence is $1.08e-10$; p-value for Precision is $8.90e-06$; p-value for Success rate is $2.00e-02$. The null hypothesis is that there are no differences between the performance of FaCoY and that of SOrec. Using 95% as the significance level, or p-value < 0.05 we see that by all quality indicators the p-values are always lower than $5e-02$. Thus, we reject the null hypothesis and conclude that the performance improvement obtained by SOrec is statistically significant.

SOrec outperforms FaCoY in terms of confidence, success rate, and precision. Furthermore, the performance difference between the two systems is statistically significant.

RQ₆: What are the reasons for the performance difference?

We refer back to the example introduced in Section 7.2. Actually, the post in Figure 23 is recommended by SOrec when the code in Listing 1 is used as query. In this example, compared to the baseline, SOrec works better since it is capable of recommending a very relevant and helpful discussion, while it is not the case with FaCoY. By carefully investigating the query generated by SOrec, we see that the transformation of import directives to produce textual tokens as shown in Listing 4 is beneficial to the search process: it equips the query with important terms which then match with the post's title. Table 16 distinguishes between the two systems by listing the facets exploited by each of them.

FaCoY exploits the Porter stemming algorithm and the English analyzing utilities provided by Lucene to perform a query. It parses the developer's source code as well as comments and uses extracted indexes described in Section 7.2 to search. As shown in RQ₄ and RQ₅, a flat query containing only full-text is not sufficient to retrieve useful results. Though FaCoY employs Lucene as its indexer, it does not exhaustively exploit boosting which is considered to be the heart of Lucene. To this end, SOrec attempts to improve the baseline by imposing various boosting measures. By the Index Creation phase, SOrec enriches incomplete code snippet with class and import directives and then tokenizes them. By the Query Creation and Execution phases, SOrec exploits import directives from source code to build indexes to match against indexed textual data. Since FaCoY

Technique	FaCoY	SOrec
Code analysis	✓	✓
Class fixing	✓	✓
Import mining	—	✓
Entropy	—	✓
BM25	—	✓
Tokenizing	—	✓

Table 16: Comparison between FaCoY and SOrec.

does not perform these phases, it is unable to match source code with textual context in post. Furthermore, FaCoY cannot match input code with snippets stored in database but without import directives.

Altogether, the query boosting scheme and the considered facets for the creation of indexes are attributed to the performance difference between the two systems.

7.6 Threats to validity

We investigate the threats that may affect the validity of the experiments as well as the efforts made to minimize them.

Internal validity. It concerns any confounding factors that may have an influence on our results. We attempted to avoid any bias in the user studies by: (i) involving 11 developers with different levels of programming experience; (ii) simulating a taste test where users are not aware of what they are evaluating. Furthermore, the labeling results by two evaluators were then double-checked by another senior researcher to aim for soundness of the outcomes.

External validity. This refers to the generalizability of the obtained results and findings. To contrast and mitigate this threat, we enforced the following measures. The sets of code snippets that have been selected as queries invoke various Java libraries. Furthermore, the number of code lines of the queries ranges from 22 to 608, attempting to cover a wide range of possibilities in practice. Our approach is also applicable to other programming languages, however, in the scope of this work we restricted ourselves to perform evaluations on posts containing Java source code.

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

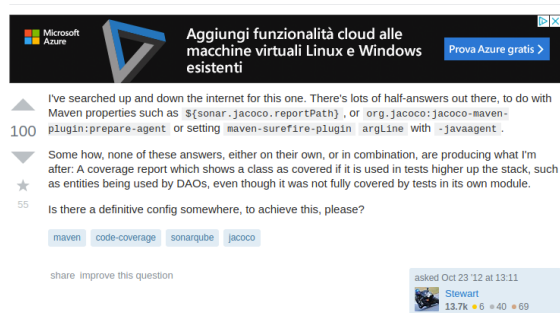
Conclusion validity. This is whether the exploited experiment methodology is intrinsically related to the obtained outcome, or there are also other factors that have an impact on it. The evaluation metrics, i.e., Success rate, Confidence, Precision might cause a threat to conclusion validity. To confront the issue, we employed the same metrics to evaluate both FaCoY and SOrec.

8 Categorization of Relevant API Discussions

When incorporating a new API, developers need to look for related resources to find out how to properly use the API's constituent functions. However, official documentation often provides a generic usage of the API, without having tailored examples that fit the current context [96]. It is necessary to find additional discussions that are useful to solve the development task. To this end, there are different forums and communication channels that contain discussions about how to use the given API. Nevertheless, the large and ever-growing volume of such forums and channels is hard to manage without proper tools. In this sense, the classification of posts to independent categories by topic provides developers with a quick-to-perceive summary of the topics that are being discussed in a communication channel [54]. This helps developers swiftly approach the most relevant posts to their need by narrowing down the search scope.

Figure 27(a) and 27(b) illustrate an example where there are two StackOverflow posts that discuss a similar issue. In Post 1²⁰, the user asks for how to use configure Maven, SonarQube and JaCoCo to produce a coverage report. Meanwhile, in Post 2²¹ the user is interested in why there is a discrepancy between the coverage that reported by JaCoCo and SonarQube. Developers who look for discussions related to the usage of SonarQube²² together with JaCoCo²³ would gain a benefit by being suggested with the two posts together as they complement each other.

How to configure multi-module Maven + Sonar + JaCoCo to give coverage report?



(a) Post 1

SonarQube plugin not finding JaCoCo results



(b) Post 2

Figure 27: An example of two related StackOverflow posts.

In supervised classification, posts can be assigned to specific categories to facilitate the search process. The labeling is performed manually, e.g., when developers create or upload a post, they specify one or more categories to the contained post. Later on, these categories serve as a means to help other developers narrow down the search scope and efficiently approach the post. Conventional wisdom suggests that the prescribed information related to posts and their corresponding categories is meaningful: it reflects the perception of humans towards the relationship between posts and categories. We hypothesize that posts' features such as questions, answers, snippets of code as well as their labels can be exploited to automatically group posts into categories. In other words, it is reasonable to categorize post by simulating humans' cognition towards the posts-categories relationship, using the available data.

²⁰<https://stackoverflow.com/questions/13031219/how-to-configure-multi-module-maven-sonar-jacoco-to-give-merged-coverage-rep>

²¹<https://stackoverflow.com/questions/38646016/sonarqube-plugin-not-finding-jacoco-results>

²²<https://www.sonarqube.org/>

²³<https://www.eclemma.org/jacoco/>

There are some notable approaches that deal with the classification of StackOverflow posts. *Hou and Mo* [54] implement a Naïve Bayes classifier to classify API discussions into API specific topics (NBM). The approach was evaluated using three datasets. CASE has been proposed to improved the overall classification performance [158]. *Beyer et al.* [15] present an automated classifier using Random Forest (RF) and Support Vector Machines (SVM).

The proliferation of Machine Learning techniques in recent years has fostered a plethora of applications in various domains, contributing superior performance compared to conventional approaches. The ability to learn from labeled data underpins the main strength of neural networks, making them a well-founded technique in Machine Learning. To name just a few, pattern recognitions [17], forecasting [154], and classification [6, 116] are their main application domains. We came up with the adoption of a neural network to build a supervised classifier to classify StackOverflow posts. We present SCORE, a tool for **S**upervised **C**lassification of **S**ta**C**k**O**ver**f**low discussions using a **neuR**al **nE**twork [97]. An evaluation using various datasets demonstrates that the tool is able to learn from manually classified data and effectively categorize incoming unlabeled data, thereby obtaining a high prediction performance. As a base for further presentations, Section 8.1 recalls the key concepts and notations related to feed-forward neural networks, which mainly come from [101] and [135]. Afterwards, Section 8.2 introduces the proposed architecture. The experimental settings are explained in Section 8.3. Finally, Section 8.5 lists the probable threats to the validity of our findings.

8.1 Feed-forward Neural Networks

The atomic element of a neural network is called *perceptron*. Each perceptron receives a set of inputs and produces an output. For each input x_i , there is a weight ω_i associated with it. A bias b is attached to allow for more flexibility in adjusting the output. An activation function f is used to compute the output, given an input. Figure 28 depicts a simplified perceptron with three inputs, and the corresponding output is: $f(\sum_{j=1}^3 \omega_j x_j + b)$.

A feed-forward neural network is made of several connected layers of neurons and the output of one layer is fed as input for the next layer's neurons, with an exception of the output layer. The edges of the network convey information in a unique direction [123], e.g., from left to right. Depending on the purpose, the number of neurons in a hidden layer as well as the number of hidden layers may vary. We consider a concrete example as depicted in Figure 29(b). For a clear presentation, the constituent weights, biases as well as the activation functions are omitted. The neural network is made of three layers: the first layer is called input and it consists of L neurons, corresponding to the number of input features, i.e., $X = (x_1, x_2, \dots, x_L)$ [154]. The second one is the hidden layer with M perceptrons, i.e., $H = (h_1, h_2, \dots, h_M)$. The output layer consists of N perceptrons, corresponding to N output categories, i.e., $C = (C_1, C_2, \dots, C_N)$. In this work, the *sigmoid function* is used as the activation function as it has been widely exploited in different studies [101]. Figure 29(a) sketches the shape of the sigmoid function.

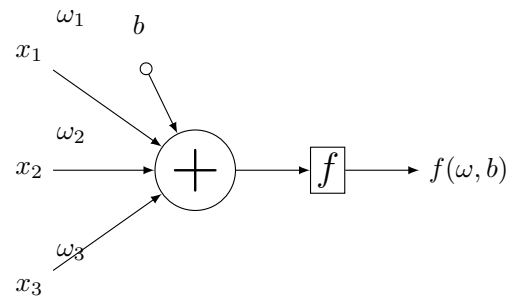


Figure 28: A perceptron.

The learning process is illustrated in the pseudo code in Algorithm 1. In this listing, *epoch* is one round of learning performed on the training data, i.e., introducing all the input vectors [14]. At the beginning of each epoch, the training set is shuffled (Line 5) with the aim of randomizing the input data, thus avoiding the problem of *being stuck in local minima* [101]. For each epoch, only some mini-batches of the training inputs

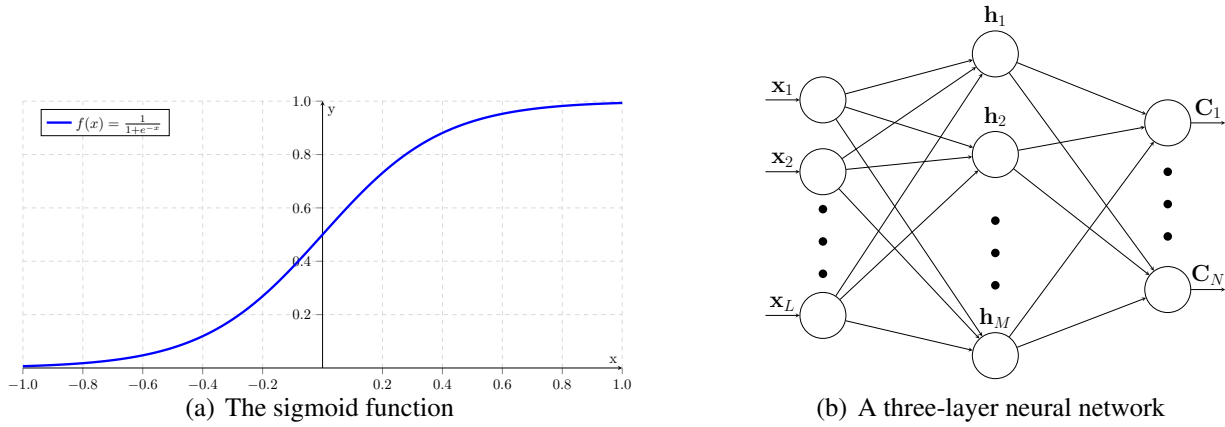


Figure 29: Sigmoid function and neural network.

Algorithm 1 The learning process

```

1: procedure LEARNING(training_data, epochs)
2:   e = 0;
3:   model = initialization();
4:   while e < epochs do
5:     training_data = shuffle(training_data);
6:     X, y = split(training_data);
7:      $\hat{y}$  = predict(X, model);
8:     error = calculate_error(y,  $\hat{y}$ );
9:     model = refine_model(model, error);
10:    e ++;
11:  end while
12:  return model
13: end procedure

```

are selected, and the training is done only on these samples. The network is fed with input data X and labels y obtained after being split (Line 6). The predicted values \hat{y} (Line 7) are the results of running on the training data and they are computed using the following formula.

$$\hat{y} = f(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + \exp(-\sum_j \omega_j x_j - b)} \quad (22)$$

The difference between the real category and the predicted value is called error (Line 8) and computed as given below.

$$E(w, b) = \frac{1}{2L} \sum_x \|y(x) - (\omega \cdot x + b)\|^2 \quad (23)$$

The error converges to zero when $\hat{y} \approx y$, i.e., the predictions match with the real labels. The final aim of the learning process is to find a function that best maps the input data with the output data. In other words, we minimize the error function $E(w, b)$ by choosing a suitable set of weights and biases [17], and this is done by applying Stochastic Gradient Descent (SGD) as follows [21, 37, 101]. The model performs prediction on the training data, then the error between the actual outcome and the predicted results is used to adjust the model to minimize errors (Line 9). The outcome of the training phase is *model* with weights and biases (Line 12) that can be used to approximately produce the outputs from the input data.

Learning is essentially the process of refining the constituent weights and biases so as to produce the corresponding output y , given a specific input X .

Algorithm 2 depicts the testing process. In contrast to learning, this phase is much simpler where only the testing data, i.e., the posts that need to be classified, is fed to `model` that has been obtained from the training phase. The final outcome is the predicted labels for the input data.

Algorithm 2 The testing process

```

1: procedure TESTING(model, testing_data)
2:    $X = \text{testing\_data}$ ;
3:    $\text{results} = \text{predict}(X, \text{model})$ ;
4: end procedure
  
```

8.2 System Architecture

The architecture of SCORE is illustrated in Figure 30 and consists of the building and deployment phases. The former processes the labeled posts ① as follows: each post is serialized into a feature vector, a format that SCORE can process through the Data Extractor ②; subsequently, the feature vectors and the corresponding labels are used to train SCORE using the Weights Calculator ③. The result consists of the weights and biases that are going to be used by the neural network to classify any incoming post in the deployment phase. Whenever an unlabelled post ④ is fed to SCORE, it is parsed employing the Data Extractor ②; the generated feature vector is then fed to the neural network ⑤ that, in turn, performs the needed classification and assigns a label to the vector. Finally, the outcome is a label that classifies the post given as input ⑥.

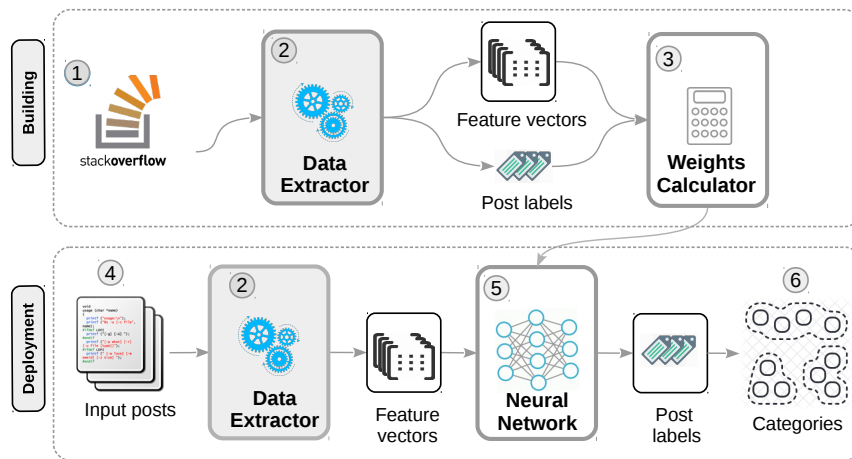


Figure 30: The SCORE architecture.

Pre-processing tasks are performed to transform a post into a feature vector [70], i.e., $X = (x_1, x_2, \dots, x_L)$, L is the number of input neurons (see Figure 29(b)).

8.3 Evaluation

This section presents the evaluation we have conducted to study the system’s performance. In particular, Section 8.3.1 introduces the datasets exploited in the evaluation. Afterwards, the metrics used to evaluate the classification outcomes are presented in Section 8.3.2.

8.3.1 Datasets

We perform evaluation using different datasets. First, we exploit the data curated and manually classified by Hou and Mo [54]. These datasets consist of posts collected from the Swing forum. Furthermore, we consider also the dataset by Beyer *et al.* [15]. This dataset consists of 500 posts which have been categorized into seven groups as follows: API CHANGE, API USAGE, CONCEPTUAL, DISCREPANCY, DOCUMENTATION, ERRORS, REVIEW. Table 17 gives a summary of all the datasets considered in our evaluation. By performing evaluation on these datasets, we are able to compare SCORE, albeit indirectly, with the above mentioned tools.

Study	Dataset	# of Posts	# of Cat.
Hou <i>et al.</i> [54]	DS-1.0	45	10
Zhou <i>et al.</i> [158]	DS-2.0	158	17
	DS-3.0	835	8
Beyer <i>et al.</i> [15]	SO	500	7

Table 17: Datasets used in the evaluation.

8.3.2 Evaluation Metrics

We apply ten-fold cross validation to evaluate the system’s performance. Given a test set, from the manually labeled data we know exactly which category each post belongs to. Thus from the testing data, we create N independent groups of posts with labels, i.e., $G = (G_1, G_2, \dots, G_N)$, which are called ground-truth data. After running SCORE on the test set, we obtain N classes i.e., $C = (C_1, C_2, \dots, C_N)$, and each contains a set of posts. We are interested in how well the produced categories match with the ground-truth data. Thus, to measure the performance of SCORE, *success rate*, *precision*, *recall*, and F_1 score are utilized [96]. If $match_i = |G_i \cap C_i|$ is the number of items that appear both in the results and ground-truth data of class i , then the metrics are explained as follows.

Success rate: It is defined as the ratio of correctly classified posts to the total number of posts in the test set.

$$success\ rate = \frac{\sum_i^N match_i}{\sum_i^N |G_i|} \times 100\% \quad (24)$$

Precision and Recall: These metrics are used to measure how accurate the results are with respect to the ground-truth data. *Precision* is the ratio of the classified items belonging to the ground-truth data:

$$precision_i = \frac{match_i}{|C_i|} \quad (25)$$

and *recall* is the ratio of the ground-truth items being found in the classified items:

$$recall_i = \frac{match_i}{|G_i|} \quad (26)$$

F₁ score (F-Measure): The metric is computed as the harmonic average of precision and recall by means of the following formula:

$$F_1 = \frac{2 \cdot precision_i \cdot recall_i}{precision_i + recall_i} \quad (27)$$

8.4 Results

Table 18 compares the accuracy obtained by SCORE and those from NBM and CASE which are extracted directly from the original papers [54],[158]. As claimed by *Zhou et al.*[158], CASE obtains a better success rate compared to NBM. It is evident that SCORE gains a better success rate for all the three datasets. For DS-1.0 where there are only 45 posts, SCORE gets 0.750 as accuracy, whereas the corresponding values by NBM and CASE are 0.622 and 0.688, respectively. This implies that given a limited amount of training data, SCORE is still able to classify posts with a considerably high success rate. For DS-3.0 where there are 835 posts, .

Dataset	NBM	CASE	SCORE
DS-1.0	0.622	0.688	0.750
DS-2.0	0.696	0.766	0.896
DS-3.0	0.931	0.950	0.969

Table 18: Success Rate.

Table 19 reports the evaluation results obtained by . We compared SCORE with the best configuration obtained by *Beyer et al.* [15] which is using Random Forrest (RF) for classification. Although in some cases, RF yields a better classification outcomes, generally, SCORE outperforms RF with respect to different metrics. For example, by category API CHANGE, RF gets 0.97, 0.97, and 0.96 as precision, recall, and F1; meanwhile, the corresponding metrics by SCORE are 0.94, 0.92, and 0.93. However, overall the accuracy achieved by SCORE is superior to that of RF in terms of precision, recall and F-Measure scores. For example, the best performance by SCORE is seen with category REVIEW, where precision, recall, and F1 scores are 0.95, 0.96, and 0.95, respectively. Finally, the average accuracy by SCORE is also better than that of RF. For instance, the average precision is 0.93 by SCORE compared to 0.89 by RF. In this sense, we conclude that our proposed approach obtains a better prediction performance on the considered datasets in comparison to the baselines.

8.5 Threats to validity

We distinguish between internal, construct, and external validity as follows.

Internal validity. Such threats are the internal factors that could have influenced the final outcomes. One possible threat could be seen through the results obtained for the dataset with a considerably low number of items, e.g., DS-1.0. Such a threat is eased by denser datasets, i.e., DS-2.0, DS-3.0, and SO.

Construct validity. They are related to the experimental settings presented in the paper, concerning the simulated setting used to evaluate the tool. The threat has been mitigated by applying ten-fold cross-validation, attempting to simulate a real scenario of classification.

Category	Precision		Recall		F-Measure	
	RF	SCORE	RF	SCORE	RF	SCORE
API CHANGE	0.97	0.94	0.95	0.92	0.96	0.93
API USAGE	0.86	0.90	0.85	0.90	0.72	0.90
CONCEPTUAL	0.82	0.91	0.82	0.92	0.69	0.91
DISCREPANCY	0.79	0.88	0.79	0.89	0.72	0.88
DOCUMENTATION	0.96	0.98	0.95	0.92	0.93	0.94
ERRORS	0.90	0.93	0.90	0.95	0.84	0.93
REVIEW	0.91	0.95	0.90	0.96	0.82	0.95
AVERAGE	0.89	0.93	0.88	0.92	0.81	0.92

Table 19: Precision, Recall, and F-Measure: Trained with RF (best configuration) and SCORE.

External validity. The main threat to *external validity* concerns the generalizability of our findings, i.e., whether they would still be valid outside the scope of this study. We attempt to moderate the threat by considering various sets of StackOverflow posts that are of different sizes and cover various categories. Nevertheless, such datasets might not necessarily reflect the entire domain. In this sense, it is essential to evaluate SCORE by incorporating a bigger dataset with more categories, as well as more items for each category. Also considering different classification categories may give more insight about encodings and whether they are (partly or totally) independent from the classification criteria. We consider this task as a future work.

9 Mining API Migration Patterns

Making use of existing third-party libraries allows developers to exploit a well-founded infrastructure, without re-implementing everything from scratch. In this way, they can save time and increase productivity. However, third-party libraries evolve over the course of time, many API functions are added, and many others are removed. A source code project that contains API calls coming from an old version of a library cannot work when it is integrated with a new version of the library. In this sense, depending software clients of a library need to be migrated in order to make use of a new library version, and this is understood as *API migration*. However, the manual migration process between different library version is time-consuming and prone to error [66]. In order to migrate a client from one version to another version of a library, a developer has to understand well the documentation of both versions as well as to choose the right matching between corresponding methods. Given the circumstances, the lack of knowledge on how to migrate the API impedes the development process. In this sense, the problem of recommending API migration is a strenuous task and it is essential to have proper machinery to assist developers in choosing the most suitable API migration patterns.

In this chapter, we introduce **amAdvisor**, a recommender system for providing recommendations related to API migrations. The system advises developers to migrate a project to use a new library version. Given two versions of a library, i.e., lib_{v1} and lib_{v2} , we collect a set of projects that use each library, which are called P_1 and P_2 , respectively. The final aim is to migrate a project in P_1 to make it be compatible with library lib_{v2} . In order to do this, all projects in P_2 are exploited as training data to be mined for patterns. Given a depending client, we migrate every breaking change declaration by means of the training data. The chapter is organized as follows. In Section 9.1 we present an introduction to the problem of API migration. The amAdvisor approach is presented in Section 9.2.

9.1 Use Case

Figure 31 depicts an example where there are two versions of a same library, i.e., lib_{v1} and lib_{v2} . For each version, there is a set of depending clients associated with it. In particular, there are three clients that use lib_{v1} , i.e., $c_{v1.1}$, $c_{v1.2}$, $c_{v1.3}$ and four clients depending on lib_{v2} , i.e., $c_{v2.1}$, $c_{v2.2}$, $c_{v2.3}$, $c_{v2.4}$. Among them, $c_{v1.1}$ now needs to be migrated in order to use the new library's version, i.e., lib_{v2} . By lib_{v2} , $c_{v2.1}$ should be the new version of $c_{v1.1}$.

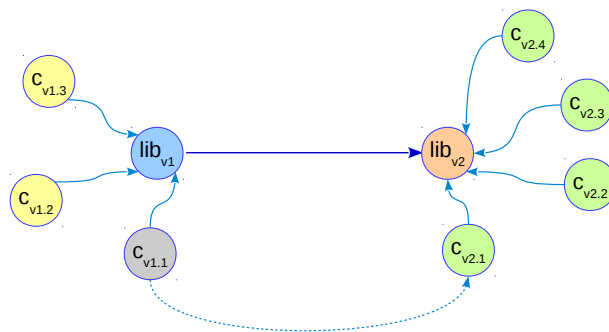


Figure 31: An example of API migration.

We take a concrete example as follows. We consider a client that invokes SonarQube (SQ) and it needs to be changed in order to use a new version of the library. Listing 5 shows a method declaration that contains API calls of the SQ version 5.6. The API function `saveMeasure()` was deprecated in SQ 5.6 and removed in SQ 7.3 and the solution is to replace `saveMeasure()` with `newMeasure()`.

```

/** Old */
public void uploadMetrics(Attribute metricAttribute, Resource resource) {
    try {
        Measure measure = createMeasureFromAttribute(metricAttribute, resource);
        if (measure != null) {
            this.sensorContext.saveMeasure(resource, measure);
        }
    } catch (SonarException e) {
        LOG.warn(e.getMessage());
        if (resource != null) {
            LOG.warn("Resource: " + resource.getName());
        }
    }
}

```

Listing 5: A declaration that needs migration.

In this case, when a new version of the library is released, many API calls of the old library's version are no longer valid, and thus causing the source code to break. In this sense, it is necessary to migrate the old source code, in order to adapt to the new library. Listing 6 depicts the new source code after the migration.

```

/** New */
public void uploadMetrics(Attribute metricAttribute, InputComponent inputComponent) {
    try {
        aType metricType = metricAttribute.getType();
        Metric metric = this.metricFinder.findByKey(metricAttribute.getName());
        if (metric != null) {
            if (metricType == aType.atInt) {
                int value = ((AttributeInt) metricAttribute).getValue();
                this.sensorContext.newMeasure().forMetric(metric).withValue(value).on(
                    inputComponent).save();
            } else if (metricType == aType.atFloat) {
                Double value = (double) ((AttributeFloat) metricAttribute).getValue();
                if (!value.isNaN() && !value.isInfinite()) {
                    if (metric.valueType().equals(ValueType.PERCENT)) {
                        value = value * 100;
                    }
                    this.sensorContext.newMeasure().forMetric(metric).withValue(value).on(
                        inputComponent).save();
                }
            }
        }
    } catch (IllegalArgumentException e) {
        LOG.warn(e.getMessage());
        if (inputComponent != null) {
            LOG.warn("Resource: " + inputComponent.key());
        }
    }
}

```

Listing 6: The declaration after migration.

Such changes are difficult to deal with, since a migration pattern depends very much on the development context. A pattern may be suitable for a specific context but not for others. In order to find a probable

migration, a developer needs to carefully read the documentation as well as to look for projects that tackle the same migration issue. In this section, we reformulate the FOCUS approach [96] to solve the API migration problem. Given a project that needs to be migrated, we search for source code projects that are similar to the given project. Afterwards, we directly mine migration patterns from the set of top most similar projects.

9.2 Proposed Approach

We propose amAdvisor, a recommender system for guiding API Migration. First, we utilize a tool developed by the CWI team, i.e., Maracas to analyze the changes between two specific versions of the same library, so-called *delta*. Then, we use the extracted delta to replace all the occurrences of the old library in the client. Finally, the FOCUS framework that we developed in the previous phases of the CROSSMINER project is used to recommend relevant API function calls and usage patterns to the “migrated” client. Moreover, to better support developers, the SOrec framework (see Section 7) is exploited to recommend also StackOverflow posts that eventually provide relevant discussions. In this sense, we propose a *triathlon approach* to API migration as we provide developers with API function calls, API usage pattern as well as StackOverflow posts to solve the problem.

Given two versions of a library, i.e., lib_{v1} and lib_{v2} , we attempt to migrate a project running lib_{v1} to make it be compatible with library lib_{v2} . In order to do this, we collected a set of projects that use each library and exploited all projects that use lib_{v2} as training data to be mined for patterns. By a depending client, we migrate every breaking change declaration by means of the training data. In particular, we deal with the following types of migrations.

- Method breaking changes.
- Class breaking changes.
- Removed methods.
- Renamed methods.
- Changed parameter list.

9.2.1 Architecture

The amAdvisor architecture is depicted in Figure 32. We exploit the FOCUS tool [96] previously developed to search for invocations from closely relevant projects. Data collected from OSS repositories ① is fed as input for the Code Parser ②, which then extracts source code projects to obtain relevant metadata. The similarities among projects, declarations are computed by the Similarity Calculator ③. The input metadata is then encoded in a computable format by means of the Data Encoder ④. Afterwards, the Recommendation Engine ⑤ exploits the similarity scores and computes using a collaborative-filtering technique to generate recommendations. In particular, the API Generator returns a ranked list of invocations, whereas the Code Builder accepts such a list to query a Knowledge Base to get real source code snippets. Afterwards, these code snippets are fed as input for SOrec ⑥ (see Section 7) which then searches for relevant posts. Finally, a combo of three different recommendations is supplied to the developer, i.e., ranked list of invocations, code snippets, and StackOverflow posts containing relevant discussions. In this way, amAdvisor is deemed to be a *triathlon approach* to API migration as it provides developers with three types of recommendations to deal with API migration. In the following subsections, we introduce in greater detail the constituent components of amAdvisor.

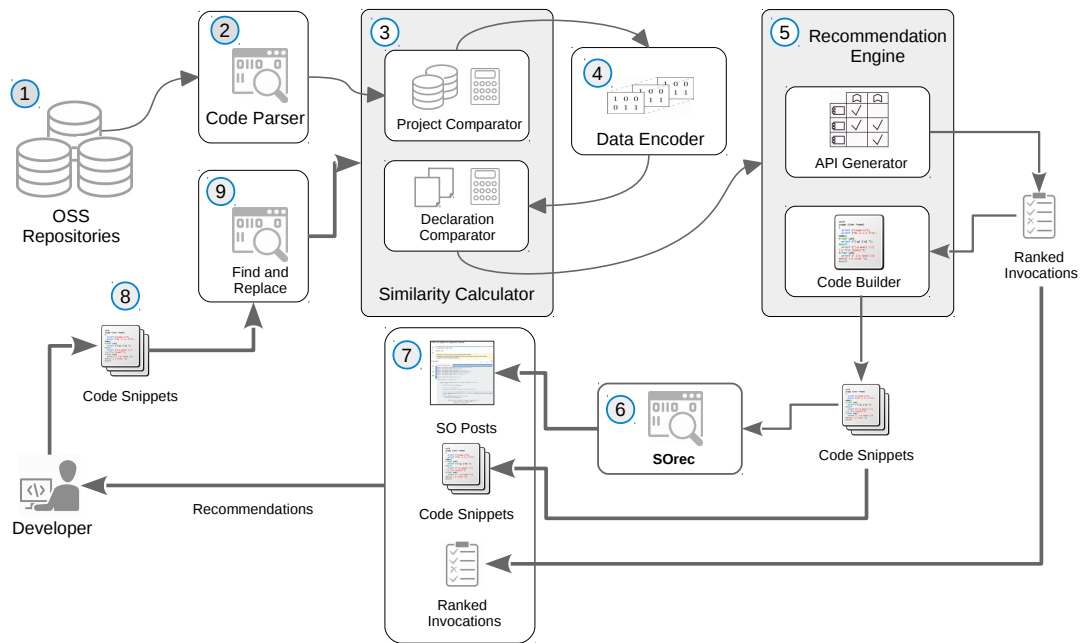


Figure 32: The amAdvisor architecture.

9.2.2 Recommending relevant function calls and code snippets

We consider declaration d_a of a client c that uses lib_{v1} . At the time of consideration, c invokes lib_{v2} , and as a result, d_a needs to be changed. For each pair of library versions, there is a list of changes that are made from the former version to the latter one and this is computed by means of Maracas, resulting in the delta. From the delta, we parse c to replace all the old API calls with the new ones. After this phase, we obtain a client which contains only API function calls of lib_{v1} , i.e., c' . We then feed c' as input to FOCUS find a set of similar projects from the set of projects that use lib_{v2} .

By FOCUS we consider a developer who is writing declaration d_a . The developer has already finished δ declarations (methods), and in d , π invocations (API calls) have been written. Furthermore, there is a set of OSS projects Q available as background data, e.g., crawled from GitHub. The final aim is to recommend to the developer real source code that helps her finish d_a , by mining the projects in Q . First, all declarations and invocations in all source files in c' and Q are extracted. Afterwards, the relationship among *projects*, *declarations*, and *invocations* is represented in a 3-D matrix. Figure 33 depicts an example of such a matrix for the set of 5 projects, i.e., (p_1, \dots, p_5) , 4 declarations (d_1, \dots, d_4) and 7 invocations (i_1, \dots, p_7) . Each slice corresponds to a project, each row is a declaration and each column is an API call. A cell is set to 1 if the project invokes an API call in the given declaration otherwise it is 0.

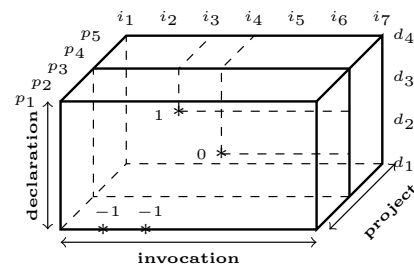


Figure 33: Tensor representation of projects, declarations and API calls.

Given a project p has a set of invocations (i_1, i_2, \dots, i_l) , its feature set is vector $\vec{\phi} = (\phi_1, \phi_2, \dots, \phi_l)$, where ϕ_k is the weight of node i_k computed as the *tf-idf* value, i.e., $\phi_k = f_{i_k} * \log(\frac{|P|}{a_{i_k}})$, f_{i_k} is the number of invocations i_k in p ; $|P|$ is the number of projects; and a_{i_k} is the number of projects that invoke i_k [38], the similarity between projects $sim_\alpha(p, q)$, $q \in Q$ is computed as shown below:

$$sim_\alpha(p, q) = \frac{\sum_{t=1}^n \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^n (\phi_t)^2} \times \sqrt{\sum_{t=1}^n (\omega_t)^2}} \quad (28)$$

FOCUS exploits a collaborative-filtering technique to predict additional invocations for d by computing the missing ratings [28] by using Equation (29):

$$r_{d,i,p} = \bar{r}_d + \frac{\sum_{e \in topsim(d)} (R_{e,i,p} - \bar{r}_e) \cdot sim_\beta(d, e)}{\sum_{e \in topsim(d)} sim_\beta(d, e)} \quad (29)$$

where $R_{e,i,p}$ is the combined rating of declaration d for i in all similar projects, computed as follows:

$$R_{e,i,p} = \frac{\sum_{q \in topsim(p)} r_{e,i,q} \cdot sim_\alpha(p, q)}{\sum_{q \in topsim(p)} sim_\alpha(p, q)} \quad (30)$$

where $sim_\alpha(d, e)$ is the similarity between projects p and q , computed using Eq. (28); $topsim(d)$ is the set of top similar declarations of d ; \bar{r}_d and \bar{r}_e are mean ratings of d and e , respectively; $sim_\beta(d, e)$ is the similarity between declarations d and e , if we call $\mathbb{F}(d)$ and $\mathbb{F}(e)$ the sets of invocations of d and e , respectively then $sim_\beta(d, e)$ is computed as follows:

$$sim_\beta(d, e) = \frac{|\mathbb{F}(d) \cap \mathbb{F}(e)|}{|\mathbb{F}(d) \cup \mathbb{F}(e)|} \quad (31)$$

The final score $r_{d,i,p}$ is a prediction for the cell representing invocation i , in declaration d of project p . Then, we get a ranked list of invocations that are considered to be relevant to d . Top- N items are selected to combine with π invocations to search for similar declarations stored in Q . Finally, a list of real code snippets is suggested to the developer.

9.3 Mining cross-project dependencies to discover API migration samples

In Section 9.3.1, we present AETHEREAL, our prototype tool for cross-projects migration dependencies graph definition. Afterwards, we introduce some interesting use cases of migration graphs in Section 9.3.2.

9.3.1 AETHEREAL

Figure 34 depicts a snapshot of a cross-projects migration dependencies graph (CPMDG). We distinguish between clients and libraries as follows. Third-party libraries are blank nodes and they identify the versions that should be migrated, whereas clients (grey nodes) are dependencies that use the libraries. The edges between can be (i) usage dependencies which are depicted as grey solid edges between a library and a client, or (ii) version dependencies, which are dotted edges among libraries. AETHEREAL is a tool written in Java that aims at supporting the automatic generation of CPMDG. The current version of AETHEREAL is both available online²⁴ and fully integrated into the KB. Maven-miner²⁵ tool extracts the Maven Dependency Graph, which

²⁴<https://github.com/crossminer/aethereal>

²⁵<https://github.com/diverse-project/maven-miner>

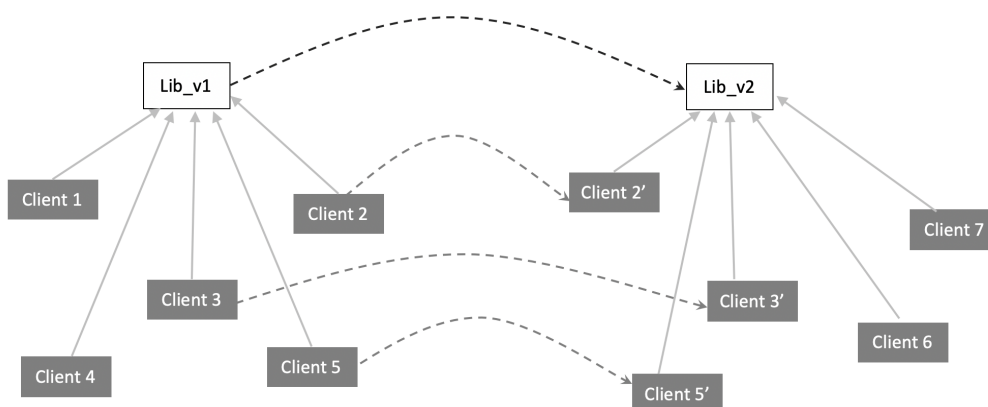


Figure 34: Cross-project migration dependencies.

is a graph-based representation of the Maven Central Repository²⁶ [13]. A graph snapshot has been showcased in [12]. AETHEREAL uses the Maven-miner dataset to compute the library CPMDGs and provides three different mining modes as follows:

- Client usage identification: given a specific library version, it provides the list of clients that use the library version.
- Migrated client pairs: given two versions of a library, it computes all the clients that have been already migrated from the initial version to the evolved one. This information is useful to understand how other clients deal with the migration of library breaking changes. Table 21 reports a simple client pairs result.
- Dependency version matrix: given a library without any version, AETHEREAL computes a dependency matrix where each column corresponds to a library version, each row is a non-version client; each cell contains the client version that uses the specific library version. A sample of the dependency version matrix is reported in Table 20. Moreover, the migrated client pairs are computed for each library version pairs.

	v1	v2	v3	v4
client1	1.1		1.2	
client2		1.0		1.1
...
clientN	1.02	1.12	1.2	2

Table 20: Dependency version matrix of a non-version library.

AETHEREAL includes various facilities that enable one to download and analyze static Maven artifacts. For instance, artifacts are manipulated as M3 models [9] which are automatically extracted from Java jar file with Rascal [64] to perform static analysis on the source or binary code. AETHEREAL supports the identification of clients that deal with the migration of library breaking changes.

MARACAS is a framework developed by the CWI team to support automatic migration of client code according to changes in APIs. The delta, i.e., changes that occur between two specific versions of the same library, and

²⁶<https://search.maven.org/>

	v2	v4
client2	1.0	1.1
client3	2.12	3.01
...		
clientN	1.12	2

Table 21: Migrated client pairs of two specific library versions, i.e., v2 and v4.

client detection model which identifies the parts of client code affected by the API evolution and must thus be migrated, are the key ingredients to tackle the API migration challenges. In this context, the MARACAS framework gets through those steps and it is used in conjunction with AETHEREAL to identify deltas and detections over mined CPDMGs. Interested readers are kindly referred to Deliverable D2.8 “API Analysis Components” for more details on MARACAS.

9.3.2 Analysis Results

In this section, we report the result of our mining studies. It is our firm belief that amAdvisor works more effectively given that more migration samples are available for recommendation.

CPMDGs have been computed over 10 popular Maven dependencies. Table 22 gives a summary of the analyzed CPMDG: The columns `#clients` and `#versions` count the total number of clients that use any version of the library and the number of library version, respectively. As reported in Table 20 AETHEREAL provides a dependency version matrix for each input library, then the `matrix density` column reports the number of non-empty cells over the total number of dependency version matrix cells. This value gives an intuition about how the clients have a proclivity for updating the library among the versions. The other columns report statical data of clients, i.e., average, min, max and the most used version. `org.sonarsource.sonarqube:sonar-plugin-api` library was included in the study to effectively support the FrondEndArt use case.

In our study, migration samples are mined between two specific versions of the library. Clients update library in spontaneous ways, e.g., they change the library every update, they change the library version when a major version is released, or they never update the library. Migration client pairs analysis are presented in Table 23. Tables 24 and 25 report the delta changes computed by MARACAS. In particular, Table 24 summarizes the delta that occurs between two versions of `gson`, i.e., 2.3.1 and 2.8.0, whereas the changes between `guava` 18 and `guava` 19 are shown in Table 25. Each delta change is related to a different element, i.e., Classes, methods and files, and different type of change, i.e., Method parameters, Static modifiers, Class/Interface implementation, Access modifiers, Abstract modifiers, etc. It is worth noting that the number of migration samples strongly depends on the distance between the targeting library version. In our study, detection models have been computed between two particular versions of a library and all initial client of clients pairs to detect which parts of client code are affected by the API evolution and thus must be migrated. An excerpt of clients detection is reported in Table 26.

id	Library	# clients	# versions	matrix density	CPL (avg)*	CPL (min)*	CPL (max)*	MUV**
1	com.google.code.gson:gson	43,727	35	0.029	1249.51	0	9979	2.3.1
2	org.springframework.data:spring-data-jpa	1,298	83	0.012	14.0	0	94	1.7.1.RELEASE
3	org.springframework.data:spring-data-mongodb	5,840	84	0.011	69,53	1	2388	1.10.7.RELEASE
4	org.apache.common:commons-collections4	11,114	3	0.33	3704.66	74	9392	4.1
5	com.google.guava:guava	1,063,016	90	0.011	1182.7	0	21753	18.0
6	org.springframework:spring-core	29,779	153	0.007	195.55	0	1384	3.0.5.RELEASE
7	joda-time:joda-time	52,003	38	0.026	1369.34	0	9595	2.9.9
8	commons-cli:commons-cli	13,152	10	0.100	1315.2	1	5376	1.2
9	commons-io:commons-io	84,675	25	0.040	3392.08	0	5376	1.2
10	com.fasterxml.jackson.core:jackson-databind	73,870	121	0.008	611.22	0	4542	2.7.4
11	org.sonarsource.sonarqube:sonar-plugin-api	365	60	0.017	6.08	0	38	5.6

* Clients per library

** Most used version

Table 22: Summary on CPMDG.

	Library	v1	v2	# client pairs	# clients V1	# clients V2	% migrated client	Major
1	com.google.guava:guava	18.0	19.0	584	21753	19335	2	Yes
2	com.google.code.gson:gson	2.3.1	2.8.0	58	9979	2992	0.5	Yes
3	com.google.code.gson:gson	2.8.0	2.8.2	129	2992	1972	4.31	No
4	org.springframework:spring-core	4.3.17.RELEASE	4.3.18.RELEASE	301	819	327	36.75	No
5	org.springframework:spring-core	3.0.5.RELEASE	4.3.8.RELEASE	18	1384	1130	0.57	Yes
6	org.sonarsource.sonarqube:sonar-plugin-api:jar	5.6	6.3	6	38	9	15.78	Yes

Table 23: Migration report.

Change type	# changes
Method parameters changed	1
Static modifiers changed	3
Class/Interface implementation changed	3
Access modifiers changed	14
Abstract modifiers changed	0
Removed elements	35
Added elements	97
Renamed elements	1
Final modifiers changed	1
Deprecated elements	0
Moved elements	109
Field and method types changed	1
Class extension changed	2
Total	267

Table 24: Delta changes from gson:2.3.1 to gson:2.8.0.

Change type	# changes
Method parameters changed	0
Static modifiers changed	0
Class/Interface implementation changed	1
Access modifiers changed	1
Abstract modifiers changed	3
Removed elements	3
Added elements	5
Renamed elements	0
Final modifiers changed	1
Deprecated elements	12
Moved elements	2
Field and method types changed	0
Class extension changed	3
Total	31

Table 25: Delta changes from guava:18.0 to guava:19.0.

Client	Client detection types				
	ABSTRACT MODIFIER	ACCESS MODIFIER	DEPRECATED	EXTENDS	FINAL MODIFIER
cassandra-driver-core-2.1.6	0	0	11	48	0
async-datastore-client-2.1.0	0	0	22	34	0
futures-extra-2.6.1	0	0	3	44	0
helios-client-0.9.25	0	0	16	29	0
herdcache-1.0.31	0	0	0	35	0
calcite-core-1.11.0	18	0	0	0	17
gfc-guava_2.10-0.1.4	0	0	8	20	0
gfc-guava_2.11-0.1.3	0	0	8	20	0
folsom-0.7.1	0	0	3	22	0
bigtable-client-core-0.2.1	0	0	2	20	0
gax-0.12.0	0	0	0	18	0

Table 26: An excerpt of client detections over 584 clients migrating from guava 18.0 to guava 19.0.

10 The Knowledge Base

There are many different ways to give real-time suggestions to developers within their accustomed IDE. CROSSMINER brings a whole new dimension to the advanced IDEs because it collects, processes and stores huge amount of data about open source components in a complex and cross-project data model. By extracting data from OSS repositories, we are able to populate a rich knowledge base that facilitates various information retrieval and recommendation techniques. This section explains in greater detail the CROSSMINER Knowledge Base (KB) and its constituent components.

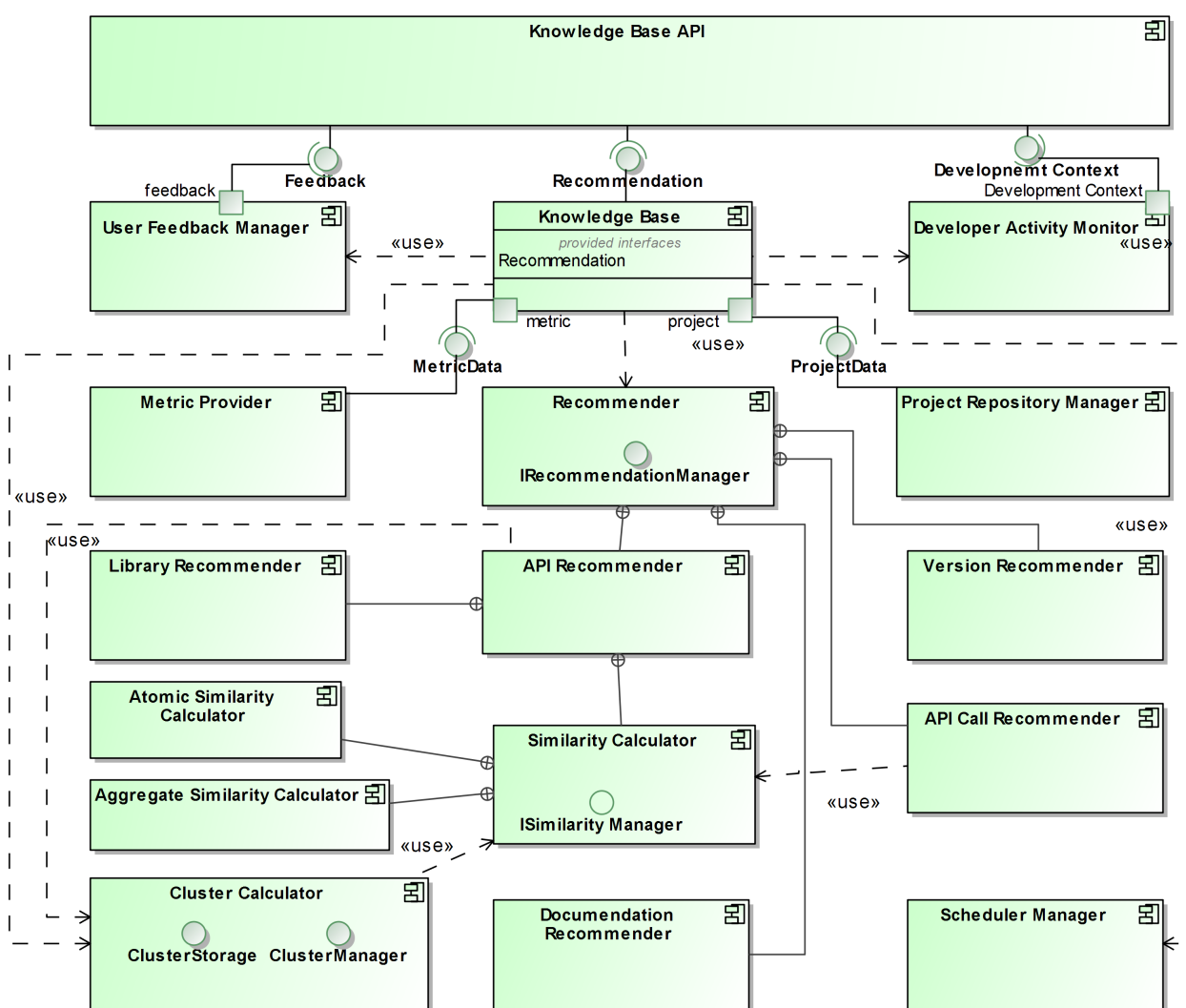


Figure 35: The Knowledge Base component diagram.

10.1 Overview

The conceived architecture together with all implemented components are depicted in Figure 35, and they are described as follows.

- The *KnowledgeBase* component is responsible for setting up the other components, as well as for executing them;
- *Recommender* creates recommendations in response to user requests. This component is extensible in order to add the management of specific types of artifacts. There are three recommender components as follows: *ApiCallRecommender* (i) provides developers with code examples that can be examined to solve the particular problem at hand (e.g., the correct usage of an API) and (ii) suggests next API function calls. The *VersionRecommender* component is used to suggest the correct version of a used third party component to developers. Finally, *APIRecommender* implements the mechanisms for retrieving from the KB information about APIs that should be used in the project being implemented;
- The *ClusterCalculator* component calculates clusters of analyzed artifacts. To this end, similarity functions are used as implemented by the *SimilarityCalculator* component, which is in charge of managing the execution of both atomic and composed similarity calculations (see the *AtomicSimilarityComposer* and *SimilarityComposer* components, respectively);
- *KnowledgeBaseScheduler* triggers the calculation of similarities and clusters thus the execution of the available similarity functions. *UserFeedbackManager* manages the feedback expressed by users on the received recommendations. *DeveloperActivityMonitor* stores and manages the data about the activity of the developer while using the CROSSMINER IDE;
- The *ClusterCalculator* component builds clusters by relying on various similarity functions and clustering algorithms as presented in Deliverable D6.4.

10.2 Use Cases

Figure 36 shows the use cases and the implementation coverage. In particular, the use cases are marked with the corresponding implemented tools which have been properly integrated into the Knowledge Base.

- **GetProjectAlternatives:** We implement novel clustering and similarity mechanisms being able to suggest alternative OSS components for already implemented OSS projects [26]. Based on designated similarity functions, we are able to detect projects that are similar because of: Provided APIs (*GetProjectAlternativesWithSimilarAPIs*) [82]; Size (*GetProjectAlternativesWithSimilarSize*); Application domain (*GetProjectAlternativesWithSimilarTopics*); and Comparable quality (*GetProjectAlternativesWithSimilarQuality*). All use cases related to similarity computation are covered by CROSSSIM [90],[98] (see Section 4 and Deliverable D6.2);
- **GetProjectsByUsedComponents:** Depending on the used components, the KB is able to identify and suggest further components that, according to what other developers have done in the past, should be also included in the system being implemented. Two prominent examples include recommendation of third-party libraries [138] and code snippets [83]. We implemented FOCUS [96] to provide the corresponding functionalities (see Section 5);
- **GetAPIUsageSupport:** The Knowledge Base provides developers with recommendations on how to use a given API and to manage the migration of the system in case of deprecated methods. This use case consists of:
 - *GetAPIUsageDiscussions:* Given an API the developer has already included, it is possible to retrieve messages from communication channels (like forums, bug reports, and Stack Overflow

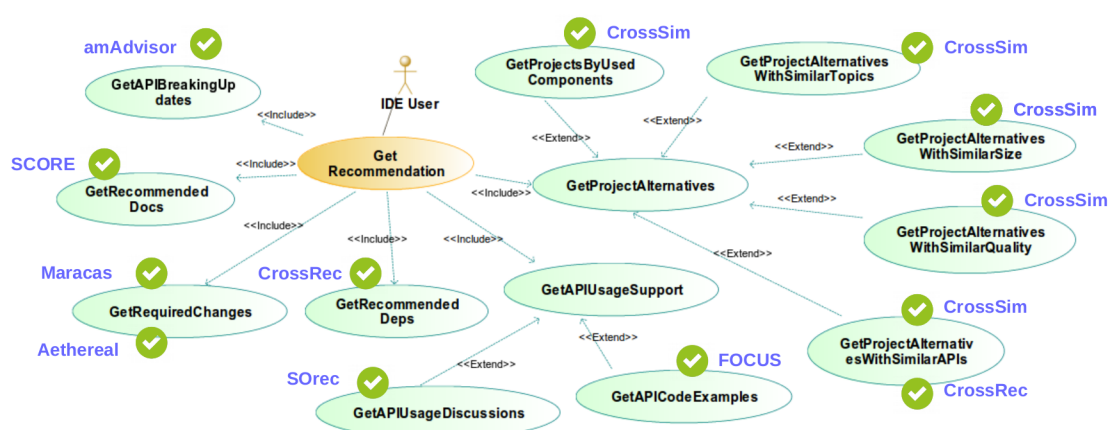


Figure 36: Implementation coverage of the use cases.

posts) that are useful for understanding how to properly use it [112]. This is covered by SCORE, a supervised classifier for categorizing related StackOverflow posts (see Section 8). Furthermore, we also provide developers with StackOverflow posts relevant to the code being developed with SOrec [124] (see Section 7);

- **GetAPIUsagePatterns:** In case of deprecated API methods, the Knowledge Base recommends code examples that can be considered as a reference for migrating the system and to make it work with the new version of the used API [102, 157]. In our implementation amAdvisor is used to suggest API migration to developers (see Section 9).
- **GetRecommendedDeps:** Starting from a given configuration and by considering similar projects developed by other developers, the Knowledge Base recommends additional third-party libraries that should be further included [138]. To this end, CROSSREC [92],[95] has been fully implemented and integrated;
- **GetRecommendedDocs:** By considering the documentation examined by other developers that used similar APIs and frameworks, the Knowledge Base suggests additional sources of information, e.g., technical documents, tutorials, etc., that are useful for solving the development problem at hand [139]. We satisfy this requirement by means of SCORE (see Section 8);
- **GetAPIBreakingUpdates:** The Knowledge Base implements the notion of API evolution with the aim of identifying backward compatibility problems affecting source code that uses evolving APIs. In our implementation, we have amAdvisor to meet this requirement (see Section 9);
- **GetRequiredChanges:** Given a changed API and a project using it, the Knowledge Base provides users with an overview of the impact that the API changes have on the depending project (Maracas). Communication channel items discussing about such API changes are also shown. Furthermore, Moreover, the Knowledge Base provides a list of all the clients that migrate from an old version of the library to a new one by means of Aethereal (see Section 9).

It is worth noting that the recommendations previously summarized have been identified during the first 6 months of the CROSSMINER project to satisfy the requirements of the industrial partners that work in the domains of IoT, multi-sector IT services, API co-evolution, software analytics, quality assurance, and OSS forges [7].

10.3 Datasets

The KB's performance largely relies on the availability of the background data. It is our firm belief that the system works effectively given that more data is available for recommendation. For this reason, we provided an initial dataset that contains data mined from various data sources as follows:

- 600 GitHub projects;
- $\approx 4,000$ jar libraries;
- $\approx 2,500,000$ Maven artifacts²⁷;
- $\approx 9,500,000$ Maven dependencies²⁷;

The current dump which is available online²⁸ allows one to replicate all the evaluation of our implemented recommender systems.

10.4 Technology Dependencies

In this section, we present the technology involved in the Knowledge Base and the rationale behind our choices. We used MongoDB²⁹ as the data store for the KB platform and Maven³⁰ to define how the KB is built and to describe its dependencies. The Knowledge Base is mainly written in Java, by means of the Spring framework³¹. Spring Boot³² makes it easy to create stand-alone, production-grade Spring-based applications.

In order to facilitate different data access technologies, to non-relational databases, Map Reduce frameworks, and cloud-based data services, we opted for Spring data³³ which is fully integrated into the Spring framework. Spring data is an umbrella project which contains many subprojects that are specific to a given database.

According to the requirements defined by CROSSMINER, all integrated components need to be tested. To this end, Junit³⁴ and spring-boot-starter-test³⁵ libraries have been chosen since they support many utilities and annotations when testing the Knowledge Base.

The equipped mining tools rely on the following frameworks and libraries:

- Apache Lucene³⁶ is a high-performance, full-featured text search engine library written entirely in Java and it has been used for tasks related to text indexing, comparison and search. The technology is suitable for nearly any application that requires full-text search, especially cross-platform.
- Simian³⁷ is used to identify duplication in Java, C#, C, etc. source code and even plain text files. Simian can also be applied on any human readable files such as ini files, deployment descriptors. The technique is suitable especially for large enterprise projects, where it can be difficult for any one developer to keep track of all the features (classes, methods, etc.) of the system.

²⁷ These datasets are directly imported from Maven miner [13].

²⁸ http://ci3.castalia.camp/dl/M30/KB_CROSSMINER.gz

²⁹ <https://www.mongodb.com>

³⁰ <https://maven.apache.org/>

³¹ <https://spring.io/>

³² <https://spring.io/projects/spring-boot>

³³ <http://projects.spring.io/spring-data/>

³⁴ <https://junit.org/junit5/>

³⁵ <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html>

³⁶ <https://lucene.apache.org/core/>

³⁷ <https://www.harukizaemon.com/simian/>

- CLAMS is a tool for summarizing API usage patterns [60]. We apply the tool to pre-group source code into independent clusters and match the developer's code against those clusters to find relevant StackOverflow posts.
- Maven-Miner³⁸ aims at resolving all Maven dependencies hosted in the Maven central repository, then, storing them into a graph database.

10.5 REST API

In this section we discuss the adoption of the final version of the Knowledge Base, which is able to receive queries from the user and answer with requested recommendations as shown in Figure 37. In particular, we present the REST APIs that have been developed in order to enable the adoption of such components from the other CROSSMINER components, and especially from the Eclipse-based IDE and the Web dashboards under development in Work Package 7. As shown in the upper side of Figure 35, the KB can be used by means of a dedicated REST API.

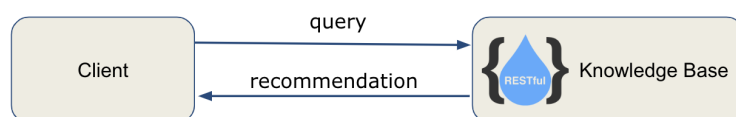


Figure 37: The interaction between a client and the Knowledge Base.

The proposed REST API has been developed using the *Spring framework*³⁹. REST has quickly become the *de facto* standard for building web services on the web since they're easy to realize and use. *Spring Web MVC* is the original web framework built on the Servlet API and included in the Spring Framework from the very beginning. It provides support to easy develop REST API.

The OpenAPI Specification⁴⁰ is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. Swagger⁴¹ takes the manual work out of API documentation, with a range of solutions for generating, visualizing, and maintaining API docs. We have integrated Swagger and OpenApi specification into the Knowledge Base, which is accessible at `/swagger-ui.html` (see Figure 38) and `/v2/api-docs` (Figure 39 shows an excerpt of the implemented APIs). The Knowledge Base architecture automatically generates OpenAPI specification and Swagger interface starting from code annotations. All the API interfaces consume and produce both `application/json` and `application/xml` content type. The currently supported operations of the Knowledge Base API are presented in the following sections.

10.5.1 Get analyzed projects

GET `/api/artifacts/artifacts?page={page}&size={size}&sort={sort}`

Description: This resource is used to retrieve a paginated list of imported projects. The path parameters are listed in Table 27.

³⁸<https://github.com/diverse-project/maven-miner>

³⁹<http://spring.io/>

⁴⁰<https://www.openapis.org/>

⁴¹<https://swagger.io/>

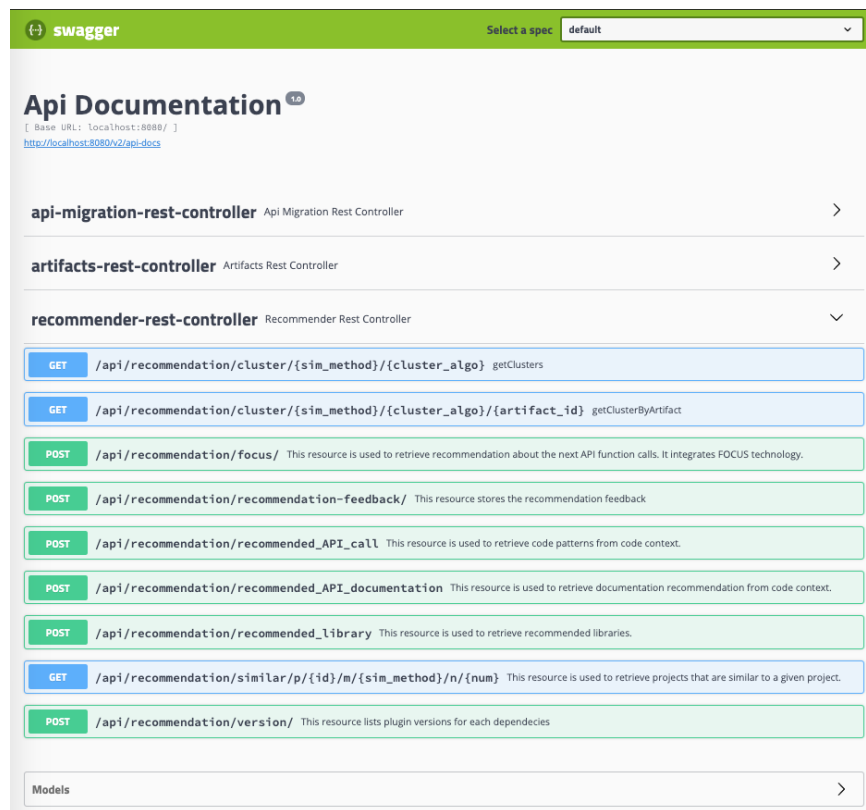


Figure 38: REST API - Swagger documentation.

Output data model: Figure 40(a) shows the artifact paged resource. Paged<Artifact> contains the pagination data (e.g.,totalElements, totalPages) and the content list of paged artifacts.

Name	Description
{page}	The number of result pages that are returned (0 .. N)
{size}	The number of records per page
{sort}	Sorting criteria in the format: property(asc desc). The default sort order is ascending. Multiple sort criteria are supported

Table 27: artifacts path parameters.

Example: Listing 7 is a call example of this API method. In particular, lines 1 and 2 produce application/json and application/xml content type, respectively. In the rest of this section we show a set of curl commands that use application/json as consumed or produced content types. The accept 'application/xml' header allows one to produce/consume XML output/input. The data model of the result is depicted in Figure 40(a)

```
1 curl -X GET "http://localhost:8080/api/artifacts/artifacts?page=0&size=10&sort=asc" -H "accept: application/json"
2 curl -X GET "http://localhost:8080/api/artifacts/artifacts?page=0&size=10&sort=asc" -H "accept: application/xml"
```

Listing 7: The curl command that gets KB analyzed project by the KB's id.

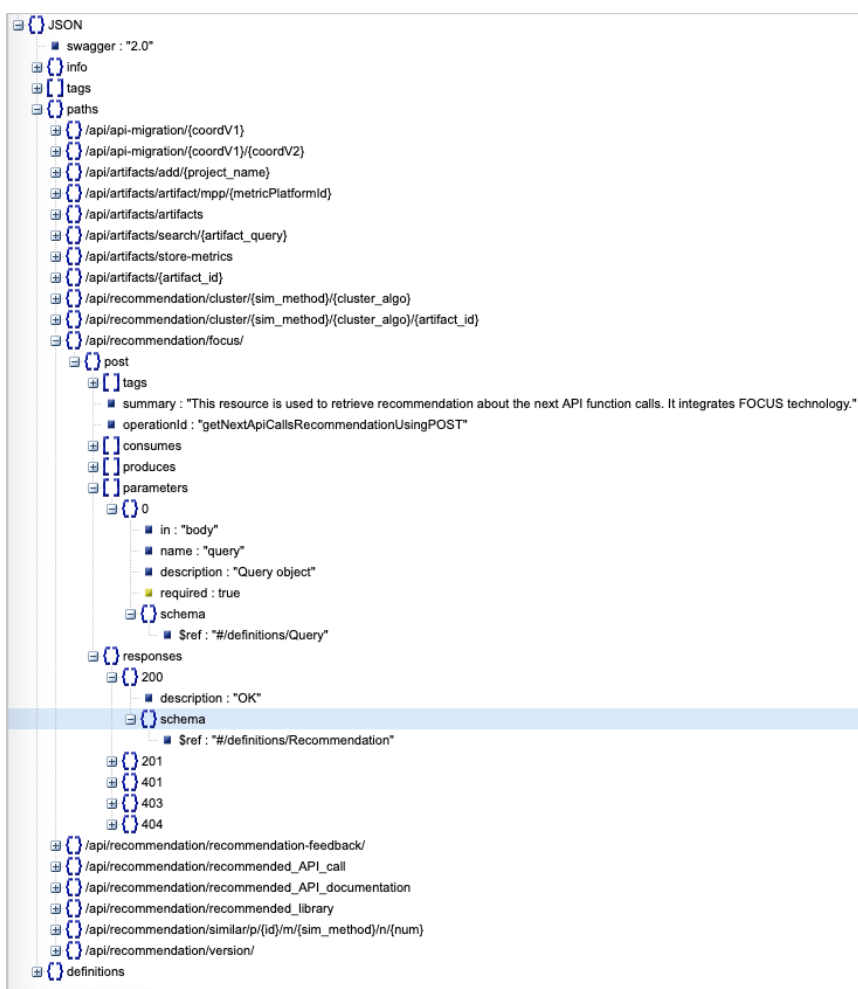


Figure 39: Open API description.

10.5.2 Get analyzed project by id

GET /api/artifacts/{artifact_id}

Description: This resource is used to retrieve an artifact that is analyzed by the Knowledge Base. This resource requires {artifact_id} as path parameter.

Output model: Figure 40(b) shows the Artifact data model that contains attributes (e.g., fullName, metricPlatformId) and inner objects (i.e., dependencies, methodDeclarations, starred, type).

Example: Listing 8 is the curl command example that calls this API method. The data model of the result is depicted in Figure 40(b)

```
curl -X GET "http://localhost:8080/api/artifacts/5b155b04065f2d726d6db241" -H "accept: application/json"
```

Listing 8: The curl command that gets KB analyzed project by the KB's id.

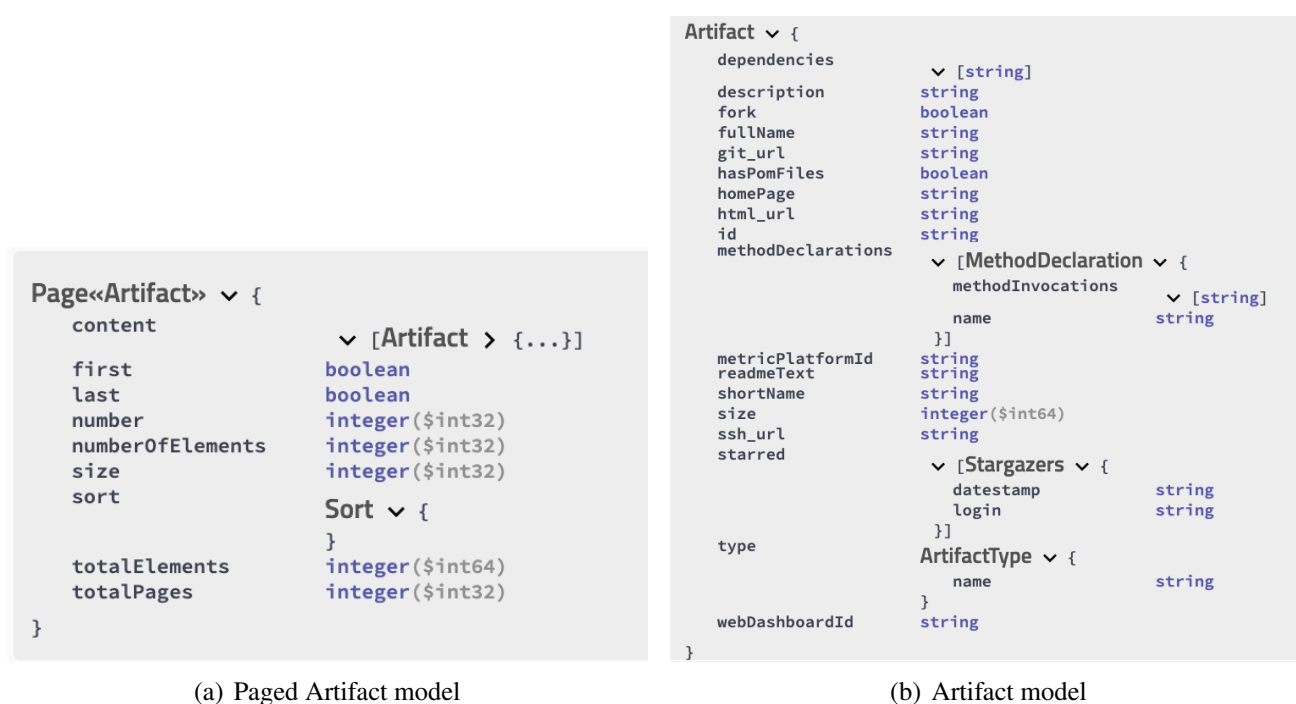


Figure 40: Artifact and paged list data objects.

10.5.3 Get projects by metric provider platform id

GET /api/artifacts/artifact/mpp/{metricPlatformId}

Description: This resource is used to retrieve an artifact that is analyzed by both the Knowledge Based and the metric provider platform. This resource takes the id of metric provider platform as path parameter and returns the artifact metadata described in the KB.

Output data model: See Section 10.5.2 for more detail on the data model.

{metricPlatformId}	The id of the metric provider platform that has been analyzed.
--------------------	--

Example: Listing 9 is a call example of this API method by curl command.

```
curl -X GET "http://localhost:8080/api/artifacts/artifact/mpp/jsonsimple" -H "accept: application/json"
```

Listing 9: The curl command that gets KB analyzed project by metric provider platform id.

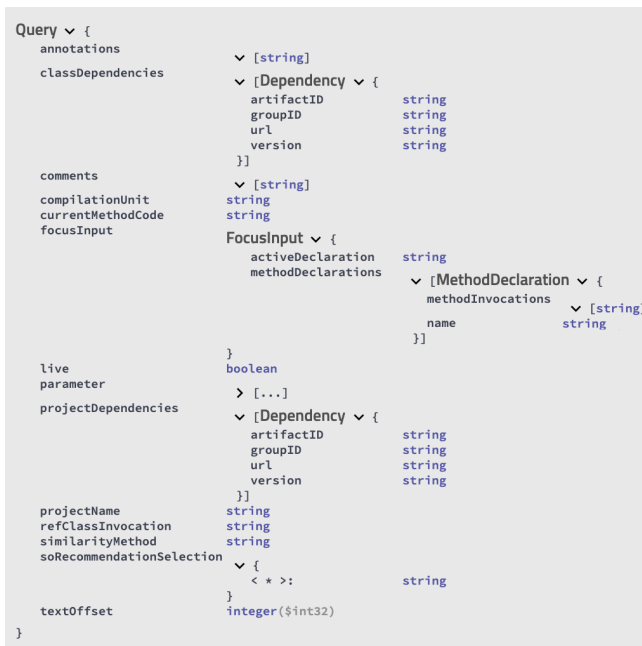
10.5.4 Search analyzed projects

GET /api/artifacts/search/{search_string}?page={page}&size={size}&sort={sort}

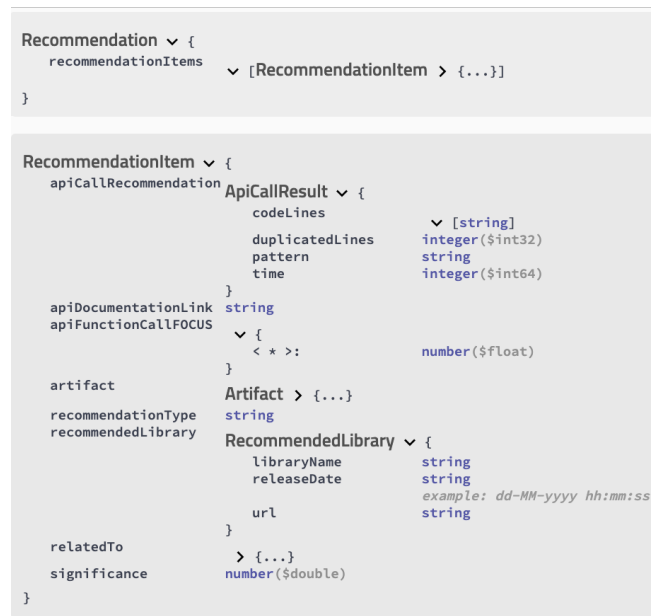
Description: This resource is used to retrieve a paginated list of projects that are analyzed by the Knowledge Based and match the search string. Table 28 lists the required path parameters.

Output data model: See output model paragraph in Section 10.5.1 .

Example: Listing 10 is a call example of this API method.



(a) Query model



(b) Recommendation model

Figure 41: Query and recommendation models.

Name	Description
{search_string}	Search string
{page}	Results page you want to retrieve (0..N)
{size}	Number of records per page
{sort}	Sorting criteria in the format: property(asc desc). The default order is ascending. Multiple sort criteria are supported

Table 28: search path parameters

```
curl -X GET "http://localhost:8080/api/artifacts/search/json-simple?page=0&size=10&sort=asc" -H "accept: application/json"
```

Listing 10: The curl command that adds a new GitHub project to the KB analyzed project.

10.5.5 Add a new GitHub project to the analyzed projects

POST /api/artifacts/add/{github_fullname}

Description: This resource is used to add a new GitHub project to the KB analyzed projects.

Path parameters:

Name	Description
{github_fullname}	The GitHub full name includes the owner (it can be a user or an organization) of repository and the name of the repository. The pattern of the GitHub fullname is <owner-name>--<repository-name>

Example: Listing 11 is a call example of this API method.

```
curl -X POST "http://localhost:8080/api/artifacts/add/crossminer--scava" -H "accept: application/json"
```

Listing 11: The curl command that adds a new GitHub project to the KB analyzed projects.

Output: the server returns 200 OK http message if the project is properly imported, 405 otherwise.

10.5.6 Store developer activity metrics

POST	/api/artifacts/store-metrics
------	------------------------------

Description: This resource is used to store the user activity metrics. It returns true if it properly stores the developers activity metrics, false otherwise.

Body request: Figure 42 shows the request's object model. All the classes that map this request model are available online⁴².

Output: The server returns 200 OK http message if the metrics are properly imported, 500 Internal error otherwise.



Figure 42: Developer activity metrics model.

Example: Listing 14 is a curl command example of this API method.

```
curl -X POST "http://localhost:8080/api/artifacts/store-metrics" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"id\": \"string\", \"metricMilestoneSlice\": [ { \"boundary\": [ { \"beginDate\": \"2019-06-21T16:33:57.342Z\", \"endDate\": \"2019-06-21T16:33:57.342Z\", \"metricValues\": [ { \"itemValuePairs\": { \"additionalProp1\": 0, \"additionalProp2\": 0, \"additionalProp3\": 0 }, \"metricName\": \"string\" } ] } ], \"boundary\": \"string\" } ], \"projectId\": \"string\", \"userId\": \"string\"}"
```

⁴²<https://tinyurl.com/y5gbrobr>

Listing 12: Store developer activity metrics curl command.

```
{
  "id": "string",
  "metricMilestoneSlice": [{
    "boundary": [
      {
        "beginDate": "2019-06-21T17:04:29.834Z",
        "endDate": "2019-06-21T17:04:29.834Z",
        "metricValues": [
          {
            "itemValuePairs": {
              "additionalProp1": 0,
              "additionalProp2": 0,
              "additionalProp3": 0
            },
            "metricName": "string"
          }
        ]
      }
    ],
    "boulder": "string"
  }],
  "projectId": "string",
  "userId": "string"
}
```

Listing 13: MetricsForProject developer activity JSON request.

10.5.7 Get alternatives projects

```
GET /api/recommendation/similar/p/{id}/m/{sim_method}/n/{num}
```

Description This resource is used to retrieve projects that are similar to a given one. All path parameters are listed in Table 29.

Output Model: This resource returns the list of most similar artifacts. Section 10.5.2 describes the *Artifact* data model.

Name	Description
{id}	id of project that is input of the query.
{sim_method}	<p>Results are computed by using the similarity function specified as parameter, which can be:</p> <ul style="list-style-type: none"> • Readme: exploiting readme files to compute the similarity between two projects; • Dependency: using the Jaccard distance on project dependencies to calculate the similarity between two projects; • CrossSim: computing similarities among all imported projects by using the star events and project dependencies; • CrossRec: a lightweight version of CROSSSIM. It uses project dependencies to provide similarities among all imported projects; • Focus: it takes the API method calls used by the projects to compute the similarity between them. • RepoPalCompound: being inspired by the approach presented in [155]; • RepoPalCompoundV2: an evolved version of RepoPalCompound similarity; • SizeBased: aggregating projects with similar size in term of lines of codes; • QualityBased: exploiting euclidean distance between the quality models computed on metric provider platform.
{num}	number of expected projects in the result.

Table 29: Path parameters of the similar artifact resource.

Example:

```
curl -X GET "http://localhost:8080/api/recommendation/similar/p/5b155b04065f2d726d6db241/m/CrossSim/n/5" -H
"accept: application/json"
```

Listing 14: The curl command that stores developer activity metrics.

```
[
  {
    id: "5a228cd62e429420384481ab",
    description: "Sample application using Spring Boot, Axon, Elasticsearch, AngularJS and Websockets",
    active: true,
    fullName: "avthart/spring-boot-axon-sample",
    html_url: "https://github.com/avthart/spring-boot-axon-sample",
    clone_url: "https://github.com/avthart/spring-boot-axon-sample.git",
    git_url: "git://github.com/avthart/spring-boot-axon-sample.git",
    master_branch: "master",
    target: {
      id: "5a228cd62e459420384351a2",
      ...
    }
  },
  ...
]
```

Listing 15: An instance of JSON result.

10.5.8 Get relevant StackOverflow posts

POST /api/recommendation/recommended_API_documentation

Description: Given a source code context as input this resource returns relevant Stack Overflow posts as output.

Body request: Figure 41(a) shows the object model of the request. In particular, this resource requires the value of compilationUnit as input.

Output Model: Figure 41(b) shows the Recommendation resource definition. Listing 18 is an excerpt of a SOrec result. apiDocumentationLink field contains the id of StackOverflow post in each recommendation item.

Example: Listing 16 is a curl command example. An instance of JSON query object is depicted in Listing 17.

```
curl -X POST "http://localhost:8080/api/recommendation/recommended_API_documentation" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"compilationUnit\": \"package camelinaction;import org.apache.camel.CamelContext;import org.apache.camel.builder.RouteBuilder;import org.apache.camel.impl.DefaultCamelContext;public class FilePrinter{\\tpublic static void main (String[] args) throws Exception{\\t\\tCamelContext context = new DefaultCamelContext();\\t\\tcontext.addRoutes(new RouteBuilder(){\\t\\t\\tpublic void configure(){\\t\\t\\t}};\\t}\\t}\""
```

Listing 16: SOrec curl command example.

```
{
  "compilationUnit": "package camelinaction;import
    org.apache.camel.CamelContext;import org.apache.camel.builder.RouteBuilder; import org.apache.camel.
    impl.DefaultCamelContext; public class FilePrinter {public static void main (String[] args) throws
    Exception{CamelContext context = new DefaultCamelContext();context.addRoutes(new RouteBuilder(){public
    void configure(){}});}"
}
```

Listing 17: SOrec request body example.

```
{
  "recommendationItems": [
    {
      "apiDocumentationLink": "45700257",
      "significance": 141.320068359375,
    },
    {
      "apiDocumentationLink": "46766311",
      "significance": 96.2380599975586,
    },
    ...
  ]
}
```

Listing 18: SOrec JSON response excerpt.

10.5.9 Get third-party libraries

POST /api/recommendation/recommended_library

Description: Given an input project, this resource returns relevant third-party libraries as output.

Body request: Figure 41(a) shows the object model of the request. In particular, this resource requires the list of used dependencies (i.e., compilationUnit inner objects) as input.

Output Model: Figure 41(b) shows the Recommendation resource definition. Listing 21 is an excerpt of a CROSSREC result. projectDependencies objects are the recommend libraries.

Example: Listing 19 is a curl command example. An instance of JSON query object is depicted in Listing 20.

```
curl -X POST "http://localhost:8080/api/recommendation/recommended_library" -H "accept: application/json" -H "Content-Type: application/json" -d '{"projectDependencies": [ { "artifactID": "junit", "groupID": "junit", "version": "4.0" }, { "artifactID": "commons-io", "groupID": "commons-io", "version": "2.0" } ]}'
```

Listing 19: CROSSREC curl command example.

```
{
  "recommendationItems": [{
    "recommendedLibrary": {
      "libraryName": "log4j:log4j:latest.release",
      "url": "https://mvnrepository.com/artifact/log4j/log4j/latest.release"
    },
    "significance": 1.3099382141578972,
    "recommendationType": "RecommendedLibrary"
  },
  {
    "recommendedLibrary": {
      "libraryName": "org.slf4j:slf4j-api:1.8.0-beta2",
      "url": "https://mvnrepository.com/artifact/org.slf4j:slf4j-api/1.8.0-beta2"
    },
    "significance": 1.2574478115112746,
    "recommendationType": "RecommendedLibrary"
  },
  ...
  ]
}
```

Listing 20: CROSSREC response body example.

```
{
  "projectDependencies": [
    {
      "artifactID": "junit",
      "groupID": "junit",
      "version": "4.0"
    },
    {
      "artifactID": "commons-io",
      "groupID": "commons-io",
      "version": "2.0"
    }
  ]
}
```

Listing 21: CROSSREC recommendation body response example.

10.5.10 Get API function calls

POST	/api/recommendation/focus/
------	----------------------------

Description: Given a project source code as input this resource returns a recommended list of API function calls.

Body request: Figure 41(a) shows the object model of the request. In particular, this resource requires the list of method declaration and method invocation pairs (i.e., methodDeclarations).

Output Model: Figure 41(b) shows the Recommendation resource definition. Listing 23 is an excerpt of a FOCUS result. `apiFunctionCallFOCUS` objects contains the recommend API function calls.

Example: The listing of a curl command is too long to be shown. Thus, an excerpt of a JSON query object is depicted in Listing 22.

```
{
  "focusInput": {
    "activeDeclaration": "com/sun/activation/viewers/TextEditor/performSaveOperation()",
    "methodDeclarations": [
      {
        "name": "com/sun/activation/registries/MimeTypeFile/parseEntry(java.lang.String)",
        "methodInvocations": [
          "java/lang/String/length()",
          "java/util/StringTokenizer/nextToken()",
          "java/lang/String/charAt(int)",
          "java/util/StringTokenizer/StringTokenizer(java.lang.String,java.lang.String)",
          "java/lang/String/trim()",
          "java/lang/String/equals(java.lang.Object)",
          "java/lang/StringBuffer/toString()",
          "java/lang/StringBuffer/append(java.lang.String)",
          "java/util/StringTokenizer/StringTokenizer(java.lang.String)",
          "java/util/StringTokenizer/countTokens()",
          "java/util/StringTokenizer/hasMoreTokens()",
          "java/util/Hashtable/put(java.lang.Object,java.lang.Object)",
          "java/lang/String/indexOf(int)",
          "java/lang/StringBuffer/StringBuffer()"
        ]
      },
      {
        "name": "com/sun/activation/registries/MailcapFile/parse(java.io.Reader)",
        "methodInvocations": [
          "java/lang/String/length()",
          "java/lang/StringBuffer/toString()",
          "java/lang/String/substring(int,int)",
          "java/lang/StringBuffer/append(java.lang.String)",
          "java/lang/StringBuffer/StringBuffer()",
          "java/lang/String/charAt(int)",
          "java/lang/String/trim()",
          "java/io/BufferedReader/BufferedReader(java.io.Reader)",
          "java/io/BufferedReader/readLine()"
        ]
      }
    ],
    ...
  ]
}
```

Listing 22: FOCUS request body example.

```
{
  "recommendationItems": [
    {
      "apiDocumentationLink": null,
      "apiCallRecommendation": null,
      "significance": 0,
      "recommendationType": "FOCUS",
      "apiFunctionCallFOCUS": {
        "com/sun/activation/viewers/ImageViewer/getToolkit()": 0.9430653,
        "java/awt/MediaTracker/MediaTracker(java.awt.Component)": 0.9430653,
        "java/awt/MediaTracker/addImage(java.awt.Image,int)": 0.9430653,
        "java/awt/MediaTracker/getErrorsID(int)": 0.9430653,
        "java/awt/MediaTracker/statusID(int,boolean)": 0.9430653,
        "java/awt/MediaTracker/waitForAll()": 0.9430653,
        "java/awt/MediaTracker/waitForID(int)": 0.9430653,
        "java/awt/Toolkit/createImage(byte[])": 0.9430653,
      }
    }
  ]
}
```

```

        "java/io/ByteArrayOutputStream/ByteArrayOutputStream()": 0.9430653,
        "java/io/ByteArrayOutputStream/toByteArray()": 0.9430653,
        "java/io/ByteArrayOutputStream/write(byte[],int,int)": 0.9430653,
        "java/io/IOException/IOException(java.lang.String)": 0.9430653,
        "java/io/InputStream/close()": 0.9430653,
        "java/io/InputStream/read(byte[])": 0.9430653,
        ...
    }
}
1
}

```

Listing 23: CROSSREC recommendation body response example.

10.5.11 Get API usage patterns

POST	POST /api/recommendation/recommended_API_call
------	---

Description: Given a project code as input this resource returns a list of patterns that matches with the current code.

Body request: Figure 41(a) shows the object model of the request. In particular, this resource requires the value of `compilationUnit` as input.

Output Model: Figure 41(b) shows the Recommendation resource definition. Listing 26 is an excerpt of a pattern recommender result. `apiCallRecommendation` contains the suggested patterns.

Example: Listing 24 is a `curl` command example. However, an excerpt of a JSON query object is depicted in Listing 25.

```

curl -X POST "http://localhost:8080/api/recommendation/recommended_API_call" -H "accept:
application/json" -H "Content-Type: application/json" -d "{ \"currentMethodCode\": \"JavaType myType =
oldType.getContentType(); \\JsonDeserializer<Object> myDeserializer = myType.getValueHandler();\\
TypeDeserializer myTypeDeserializer = myType.getTypeHandler();\\}"

```

Listing 24: Pattern recommender curl command example.

```

{
  "currentMethodCode": "JavaType myType = oldType.getContentType(); \\nJsonDeserializer<Object>
myDeserializer = myType.getValueHandler();\\nTypeDeserializer myTypeDeserializer = myType.getTypeHandler
();"
}

```

Listing 25: Pattern recommender request body example.

```

{
  "recommendationItems": [
    {
      "apiCallRecommendation": {
        "codeLines": [
          "{",
          "    DeserializationContext ctxt;",
          "    final BeanDescription beanDesc;",
          "    final DeserializerFactoryConfig _factoryConfig;",
          "    ArrayType type;",
          "    final DeserializationConfig config = ctxt.getConfig();",
          "    JavaType elemType = type.getContentType();",
          "...",
          "}"
        ],
        ...
      }
    }
  ]
}

```



Figure 43: Cluster data model.

```

    "duplicatedLines": 6,
    "time": 2449,
    "pattern": "jackson-databind_69.java"
  }, ...
]
}

```

Listing 26: Pattern recommender recommendation body response example.

10.5.12 Get clustered projects

GET cluster/sim_method/cluster_algo

Description This resource is used to retrieve clusters of projects. All path parameters are listed in Table 30.

Output Model: This resource returns the list of artifact clusters. Figure 43 depicts the Cluster data object. In Section 10.5.2, we described the Artifact data model.

Example: An excerpt of JSON response object is depicted in Listing 27.

Name	Description
{sim_method}	Results are computed by using the similarity function specified as parameter, which is the same with those in Table 29.
{cluster_algo}	results are computed by using the clustering algorithm specified as parameter, which can be <i>CLARA</i> , <i>K-Medoids</i> or <i>HCLibrary</i> .

Table 30: Resource path parameters.

```

[
  {
    "clusterization": {
      "clusterizationDate": 1544376657311,

```

```

    "similarityMethod": "CrossSim",
    "clusterAlgorithm": "Clara",
    "id": "5c0d5151fe03927a05d2055a"
  },
  "mostRepresentative": {
    "id": "5c0d5150fe03927a05d203ce",
    "name": "AlipayOrdersSupervisor-GUI",
    "description": "GUI of AlipayOrdersSupervisor, implemented in Java and Swing",
    ...
  },
  "artifacts": [
    {
      "id": "5c0d5150fe03927a05d203d2",
      "name": "RssToMobiService",
      "description": "A rss to mobi service",
      ...
    }, ...
  ],
  {
    "clusterization": {
      ...
    }
  }
]

```

Listing 27: An excerpt of JSON result.

10.5.13 Get cluster containing a particular project

GET cluster/{sim_method}/{cluster_algo}/{artifact_id}

Description This resource is used to retrieve the cluster that contains a given project. It checks among the clusters that are computed by the given similarity method and cluster algorithm. All path parameters are listed in Table 31.

Output Model: This resource returns the list of cluster of artifacts. Figure 43 depicts the `Cluster` data object. In section 10.5.2, we describe the `Artifact` data model.

Example: An excerpt of JSON response object is depicted in Listing 28.

Name	Description
{sim_method}	Results are computed by using the similarity function specified as parameter, which is the same with those in Table 29.
{cluster_algo}	Results are computed by using the clustering algorithm specified as parameter, which can be <i>CLARA</i> , <i>K-Medoids</i> or <i>HCLibrary</i> .
{artifact_id}	The id of the artifact.

Table 31: Resource path parameters.

```

{
  "clusterization": {
    "clusterizationDate": 1544376657311,
    "similarityMethod": "CrossSim",
    "clusterAlgorithm": "Clara",

```



```

MigrationInfo {
  clientsV1      > [...]
  clientsV2      > [...]
  count          integer($int32)
  libv1          Artifact > {...}
  libv2          Artifact > {...}
}

```

Figure 44: Migration model.

```

    "id": "5c0d5151fe03927a05d2055a"
  },
  "mostRepresentative": {
    "id": "5c0d5150fe03927a05d203ce",
    "name": "AlipayOrdersSupervisor-GUI",
    "description": "GUI of AlipayOrdersSupervisor, implemented in Java and Swing",
    ...
  },
  "artifacts": [
    {
      "id": "5c0d5150fe03927a05d203d2",
      "name": "RssToMobiService",
      "description": "A rss to mobi service",
      ...
    }, ...
  ],
  ...
}

```

Listing 28: JSON result excerpt.

10.5.14 Get migration client pairs examples

```
GET /api/api-migration/{coordV1}/{coordV2}
```

Description This resource provides the list of client pairs that already migrate from the initial version of the API to the evolved one. All path parameters are listed in Table 32.

Output Model: This resource returns the client migration information. This recommendation uses the Artifact object model defined by aether Eclipse projects. The Java MigrationInfo mapping class is shown in Figure 44.

```

{
  "libv1": {
    "groupId": "junit",
    "artifactId": "junit",
    "version": "4.10",
    "extension": "jar"
  },
  "libv2": {
    "groupId": "junit",
    "artifactId": "junit",
    "version": "4.12",
    "extension": "jar"
  }
}

```

Name	Description
{coordV1}	The Maven central coordinate of the initial version of the library. The format is <code><group_id>:<artifact_id>:<version></code> .
{coordV2}	The Maven central coordinate of the evolved version of the library. The format is <code><group_id>:<artifact_id>:<version></code> .

Table 32: Resource path parameters.

```

}
"count": 143,
"clientsV1": [
  {
    "groupId": "org.apache.jmeter",
    "artifactId": "ApacheJMeter_tcp",
    "version": "2.8",
    "extension": "jar",
  }, ...
]
"clientsV2": [...]
}

```

Listing 29: JSON result excerpt.

10.5.15 Get clients using a particular library version

Description This resource provides the list of client pairs that already migrate from the initial version of the API to an evolved one. {coord} path parameter is the Maven central coordinate of a specific library version. The coordinate format is `<group_id>:<artifact_id>:<version>`.

Output Model: This resource returns the list of clients that use a specific Maven library version. This recommendation uses the `Artifact` object model defined by aether Eclipse projects. Listing 30 shows an excerpt of the resource response.

```

[
  {
    "groupId": "org.apache.jmeter",
    "artifactId": "ApacheJMeter_tcp",
    "version": "2.8",
    "extension": "jar",
  }, ...
]

```

Listing 30: JSON result excerpt.

10.5.16 Get StackOverflow posts related to discussions about API migration

GET /api/api-migration/documentation/{coordV1}/{coordV2}

Description This resource provides a list of StackOverflow posts. The list of path parameters is shown in Table 33.

Name	Description
{coordV1}	The Maven central coordinate of the initial version of the library. The format is <code><group_id>:<artifact_id>:<version></code>
{coordV2}	The Maven central coordinate of the evolved version of the library. The format is <code><group_id>:<artifact_id>:<version></code>

Table 33: Resource path parameters.

Output Model: Figure 41(b) shows the Recommendation resource definition. Listing 31 is an excerpt of a SOrec result. `apiDocumentationLink` field contains the id of StackOvderflow post in each recommendation item.

```
{
  "recommendationItems": [
    {
      "apiDocumentationLink": "45700257",
      "significance": 141.320068359375,
    },
    {
      "apiDocumentationLink": "46766311",
      "significance": 96.2380599975586,
    },
    ...
  ]
}
```

Listing 31: SOrec JSON response excerpt.

10.5.17 Get impact on library evolution

POST	POST /api/api-migration/detection/{clientV1}/{clientV2}
------	---

Description This resource provides a list of client locations that are impacted by the evolution on the specific library. Table 34 shows the list of path parameters.

Output Model: This resource returns the list of detections. Figure 45 depicts the Detection data object.

```
Detection {
  clientLocation string
  newLibraryLocation string
  oldLibraryLocation string
  type string
  Enum:
    [ ACCESS_MODIFIER, FINAL_MODIFIER, STATIC_MODIFIER, ABSTRACT_MODIFIER, DEPRECATED, RENAMED, MOVED, REMOVED, PARAMS_LIST, RETURN_TYPE, TYPE, EXTENDS, IMPLEMENTS ]
}
```

Figure 45: Detection data object

Example: Listing 32 is an instance of curl command that calls this resource. An excerpt of JSON response object is depicted in Listing 33.

```
curl -X POST "http://localhost:8080/api/api-migration/detection/com.google.guava:guava:18/com.google.guava:guava:19" -H "accept: */*" -H "Content-Type: multipart/form-data" -F "file=@my.m3;type=text/plain"
```

Listing 32: Detection curl command.

Name	Description
{coordV1}	The Maven central coordinate of the initial version of the library. The format is <code><group_id>:<artifact_id>:<version></code>
{coordV2}	The Maven central coordinate of the evolved version of the library. The format is <code><group_id>:<artifact_id>:<version></code>
{clientM3}	The M3 model of the client. A multipart file is needed

Table 34: Resource path parameters.

```
[
  {
    "clientLocation": "autofixture/publicinterface/generators/implementationdetails/ConcreteInstanceType/
isAssignableFrom(java.lang.Class",
    "oldLibraryLocation": "java+method:///com/google/common/reflect/TypeToken/isAssignableFrom(java.lang.
reflect.Type)",
    "newLibraryLocation": "java+method:///com/google/common/reflect/TypeToken/isAssignableFrom(java.lang.
reflect.Type)",
    "type": "REMOVED"
  },
  ...
]
```

Listing 33: Detection JSON response.

10.5.18 Get useful code snippets to migrate towards a new library version

Description: This resource provides a list of code snippet that other clients use to support the breaking changes introduced by the adoption of a new version of a library. Table 34 shows the list of path parameters.

Output Model: This resource returns the list of code snippet. Figure 46 depicts the Detection data object.

Example: Listing 34 is an instance of curl command that calls this resource. An excerpt of the JSON response object is depicted in Listing 35.

```
Detection {
  clientLocation string
  newLibraryLocation string
  oldLibraryLocation string
  type string
  Enum:
    [ ACCESS_MODIFIER, FINAL_MODIFIER, STATIC_MODIFIER, ABSTRACT_MODIFIER, DEPRECATED, RENAMED, MOVED, REMOVED, PARAMS_LIST, RETURN_TYPE, TYPE, EXTENDS, IMPLEMENTS ]
}
```

Figure 46: Detection data object.

```
curl -X POST "http://localhost:8080/api/api-migration/recommend/com.google.guava:guava:18/com.google.guava:
guava:19" -H "accept: */*" -H "Content-Type: multipart/form-data" -F "file=@my.m3;type=text/plain"
```

Listing 34: Detection curl command.

```
{
  "recommendationItems": [
    {
      "apiCallRecommendation": {
        "codeLines": [
          "{",
```

```
"    DeserializationContext ctxt;",
"    final BeanDescription beanDesc;",
"    final DeserializerFactoryConfig _factoryConfig;",
"    ArrayType type;",
"    final DeserializationConfig config = ctxt.getConfig();",
"    JavaType elemType = type.getContentType();",
"",
"    ...",
"}"
],
"duplicatedLines": 6,
"time": 2449,
"pattern": "jackson-databind_69.java"
},
},...
]
```

Listing 35: Pattern recommender recommendation body response example.

11 Conclusions

In this deliverable, we presented the results related to the final version of the Knowledge Base. Together with the previous deliverables, i.e., D6.1, D6.2, D6.3, and D6.4, we fulfilled all the requirements defined in Deliverable D1.1 as shown in Table 35. This section summarizes the main contributions of Work Package 6.

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

Table 35: Implementation status of the required recommendations as presented in D6.1. Early stage: ○; Half done: ◐; Fully done: ●

We developed FOCUS, a context-aware collaborative-filtering recommender system for supporting API recommendations. The tool is twofold, as it concurrently supports two related use cases, namely API function calls and API usage pattern recommendations. Based on a dedicated graph representation, we are able to represent the relationships among projects, method declarations and method invocations and to compute the similarities among them. The similarity scores are then used as input for the recommendation engine that in turn predicts the inclusion of additional invocations and eventually generates recommendations. A preliminary evaluation on a dataset of 3,600 jar files curated from the Maven repository shows that FOCUS is able to provide highly relevant API function calls. Furthermore, we use a combination of CLAMS [60] and Simian to recommend API usage patterns. Though this functionality is already covered by FOCUS, we aim at enriching the Knowledge Base with various recommendation techniques, thus allowing developers to freely choose the most convenient one.

We presented CROSSREC, a novel approach to library recommendation that relies on a collaborative-filtering recommender system to assist software developers in mining OSS repositories. The approach has been evaluated by considering different quality metrics and a dataset consisting of 1,200 Java projects. The experimental results show that our system for recommending third-party libraries outperforms LibRec, a well-known baseline.

To solve the problem of API documentation, we developed SOrec, a tool that searches for StackOverflow posts containing related code and discussions that are useful for a programming task. SOrec addresses both the issue of properly indexing SO posts, and that of automatically creating queries in a transparent manner for the developer. In particular, SOrec performs different augmentations of SO posts for indexing them, and of input contexts for creating corresponding queries. To study the performance of SOrec we performed large-scale user studies. A first study has been done in order to understand which combination of the conceived augmentations is the best one in terms of SOrec performance. A second and larger user study has been done to compare SOrec with FaCoY. The experimental results show that SOrec outperforms the module of FaCoY, which is devoted to searching for SO posts that are relevant with input developer contexts. The implementation of SOrec is

twofold. First, we already started with the integration of SOrec into the Eclipse IDE and in this respect, our short-term plan is to improve the usage of the tool directly from the IDE to suitably support developers in real-world settings. Second, the tool can be used to replace the corresponding module by FaCoY, aiming to boost up the system's overall performance. Our future research agenda focuses on performing further evaluations, especially to compare SOrec with those approaches that rely on general purpose search engines and that focus only on the query creation phase (e.g., Prompter [113]). SOrec can be combined with the previous tools and approaches with the aim of providing developers with recommendations consisting of both source code and related discussions retrieved from StackOverflow. SOrec is highly related to existing code search engines as they can be used in combination to provide developers with not only API calls, sample source code but also related discussions. Though SOrec works well given the context, we still believe that its performance can be further improved, e.g., by better exploiting the boosting scheme. We consider the issue as our future research.

Aiming at supporting developers in working with new APIs, we introduced SCORE, a supervised classifier for categorizing StackOverflow posts. Given a specific API, SCORE is able to group the most relevant discussions that are useful for facilitating the integration task. To evaluate SCORE, we exploit various datasets coming from existing studies. The experimental results show that SCORE obtains a high classification performance and thus outperforming the considered baselines. To further improve the system's performance, we plan to deploy a deep neural network to classify posts, and this remains a future work in our academic calendar.

Finally, to assist developers in dealing with API breaking changes by suggesting relevant migration patterns, we proposed *amAdvisor*, a recommender system that works on top of FOCUS and a tool for detecting API changes developed by the CWI team. *amAdvisor* is able to assist developers in choosing the right migration patterns by mining from projects that invoke the new library version. Furthermore, *amAdvisor* exploits SOrec to supply relevant StackOverflow discussions to developers.

References

- [1] Apache Lucene Core. <https://lucene.apache.org/core/>. last access 26.04.2019.
- [2] Stackoverflow. <https://stackoverflow.com/>. last access 04.04.2019.
- [3] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow: An exploratory study on android apps. *Information and Software Technology*, 88:148 – 158, 2017.
- [4] Charu Aggarwal. *Neighborhood-Based Collaborative Filtering*, pages 29–70. Springer International Publishing, Cham, 2016.
- [5] Sylvain Arlot, Alain Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010.
- [6] M. F. Augusteijn and B. A. Folkert. Neural network classification and novelty detection. *International Journal of Remote Sensing*, 23(14):2891–2902, 2002.
- [7] Alessandra Bagnato et. al. Developer-centric knowledge mining from large open-source software repositories (crossminer). In *Software Technologies: Applications and Foundations*, pages 375–384. Springer International Publishing, 2018.
- [8] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 319–330, New York, NY, USA, 2018. ACM.
- [9] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A general model for code analytics in rascal. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pages 25–28, March 2015.
- [10] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 251–262, July 2017.
- [11] Alejandro Bellogín, Iván Cantador, and Pablo Castells. A comparative study of heterogeneous item recommendations in social systems. *Inf. Sci.*, 221:142–169, February 2013.
- [12] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: a temporal graph-based representation of maven central. *CoRR*, abs/1901.05392, 2019.
- [13] Amine Benelallam, Nicolas Harrand, César Soto Valero, Benoit Baudry, and Olivier Barais. Maven central dependency graph, November 2018. The Maven dependency graph is the fruit of a collaboration between the DiverSE team (Inria Rennes, France) and CASTOR project (KTH, Sweden). Instructions on how to use and reproduce the dataset can be found in the dataset’s repository on [Github](<https://github.com/diverse-project/maven-miner>). A complete description of the dataset and usages can be found in the accompanying [paper] (<https://arxiv.org/abs/1901.05392>).
- [14] Yoshua Bengio. *Practical Recommendations for Gradient-Based Training of Deep Architectures*, pages 437–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

- [15] Stefanie Beyer, Christian Macho, Martin Pinzger, and Massimiliano Di Penta. Automatically classifying posts into question categories on stack overflow. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 211–221, New York, NY, USA, 2018. ACM.
- [16] Upasna Bhandari, Kazunari Sugiyama, Anindya Datta, and Rajni Jindal. Serendipitous recommendation for mobile apps using item-item similarity graph. In Rafael E. Banchs, Fabrizio Silvestri, Tie-Yan Liu, Min Zhang, Sheng Gao, and Jun Lang, editors, *AIRS*, volume 8281 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2013.
- [17] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.
- [18] 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.
- [19] Markus Borg, Per Runeson, Jens Johansson, and Mika V. Mäntylä. A replicated study on duplicate detection: Using apache lucene to search among android defects. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 8:1–8:4, New York, NY, USA, 2014. ACM.
- [20] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, Oct 2016.
- [21] Léon Bottou. Stochastic gradient learning in neural networks. In *In Proceedings of Neuro-Nîmes. EC2*, 1991.
- [22] Cristian E. Briguez, Maximiliano C.D. Budán, Cristhian A.D. Deagustini, Ana G. Maguitman, Marcela Capobianco, and Guillermo R. Simari. Argument-based mixed recommenders and their application to movie suggestion. *Expert Systems with Applications*, 41(14):6467 – 6482, 2014.
- [23] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. On evaluating recommender systems for api usages. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 16–20, New York, NY, USA, 2008. ACM.
- [24] Fidel Cacheda, Víctor Carneiro, Diego Fernández, and Vreixo Formoso. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Trans. Web*, 5(1):2:1–2:33, February 2011.
- [25] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [26] Andrea Capiluppi, Davide Di Ruscio, Juri Di Rocco, Phuong T. Nguyen, and Nemitari Ajienka. The Effects of Clustering on the Characteristics of Java Software - manuscript under revision. *Journal of Systems and Software*, 2019.
- [27] Pablo Castells and Saúl Vargas. Novelty and diversity metrics for recommender systems: Choice, discovery and relevance. In *In Proceedings of International Workshop on Diversity in Document Retrieval (DDR)*, pages 29–37.

- [28] Annie Chen. Context-aware collaborative filtering system: Predicting the user's preference in the ubiquitous computing environment. In *Proceedings of the First International Conference on Location- and Context-Awareness*, LoCA'05, pages 244–253, Berlin, Heidelberg, 2005. Springer-Verlag.
- [29] 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.
- [30] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.
- [31] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 191–198, New York, NY, USA, 2016. ACM.
- [32] Paolo Cremonesi, Roberto Turrin, Eugenio Lentini, and Matteo Matteucci. An evaluation methodology for collaborative recommender systems. In *Proceedings of the 2008 International Conference on Automated Solutions for Cross Media Content and Multi-channel Distribution*, AXMEDIS '08, pages 224–231, Washington, DC, USA, 2008. IEEE Computer Society.
- [33] Jonathan Crussell, Clint Gibling, 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.
- [34] Barthélemy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. Moving into a new software project landscape. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 275–284, New York, NY, USA, 2010. ACM.
- [35] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM.
- [36] Lucas B. L. de Souza, Eduardo C. Campos, and Marcelo de A. Maia. Ranking crowd knowledge to assist software development. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 72–82, New York, NY, USA, 2014. ACM.
- [37] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th Int. Conf. on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc.
- [38] 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.
- [39] Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio. CrossRec tool and evaluation data, 2018. <https://doi.org/10.5281/zenodo.1252848>.

- [40] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 266–276, Piscataway, NJ, USA, 2012. IEEE Press.
- [41] 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.
- [42] 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.
- [43] Marko Gasparic and Andrea Janes. What recommendation systems for software engineering recommend. *J. Syst. Softw.*, 113(C):101–113, March 2016.
- [44] Sanjoy Ghose and Oded Lowengart. Taste tests: Impacts of consumer perceptions and preferences on brand positioning strategies. *Journal of Targeting, Measurement and Analysis for Marketing*, 10(1):26–41, Aug 2001.
- [45] Carlos A. Gomez-Urbe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4):13:1–13:19, December 2015.
- [46] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *40th International Conference on Software Engineering*, pages 933–944, New York, 2018. ACM.
- [47] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API Learning. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, New York, 2016. ACM.
- [48] Guibing Guo, Jie Zhang, and Neil Yorke-Smith. A novel bayesian similarity measure for recommender systems. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI '13*, pages 2619–2625. AAAI Press, 2013.
- [49] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.
- [50] Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.
- [51] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, October 1977.
- [52] Angelos Hliaoutakis, Giannis Varelakis, Epimenidis Voutsakis, Euripides G. M. Petrakis, and Evangelos E. Milios. Information retrieval by semantic similarity. *Int. J. Semantic Web Inf. Syst.*, 2(3):55–73, 2006.
- [53] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 237–240, 2005.

- [54] Daqing Hou and Lingfeng Mo. Content categorization of api discussions. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13*, pages 60–69, Washington, DC, USA, 2013. IEEE Computer Society.
- [55] F.O. Isinkaye, Y.O. Folajimi, and B.A. Ojokoh. Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal*, 16(3):261 – 273, 2015.
- [56] Anastasia Izmaylova, Paul Klint, Ashim Shahi, and Jurgen J. Vinju. M3: an open model for measuring code artifacts. *CoRR*, abs/1312.1188, 2013.
- [57] Paul Jaccard. The Distribution of the Flora in the Alpine Zone. *New Phytologist*, 11(2):37–50, 1912.
- [58] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. Why and how developers fork what from whom in github. *Empirical Softw. Engg.*, 22(1):547–578, February 2017.
- [59] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Procs. of the Tenth International Conf. on Information and Knowledge Management, CIKM '01*, pages 247–254, New York, NY, USA, 2001. ACM.
- [60] Nikolaos Katirtzis, Themistoklis Diamantopoulos, and Charles Sutton. Summarizing software api usage examples using clustering techniques. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 189–206, Cham, 2018. Springer International Publishing.
- [61] Houda Khrouf and Raphaël Troncy. Hybrid event recommendation using linked data and user diversity. In *Proceedings of the 7th ACM Conference on Recommender Systems, RecSys '13*, pages 185–192, New York, NY, USA, 2013. ACM.
- [62] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: a code-to-code search engine. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 946–957. ACM, 2018.
- [63] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conf. on Empirical Methods in NLP, EMNLP 2014, October 25-29, 2014, Doha, Qatar*, pages 1746–1751, 2014.
- [64] P. Klint, T. v. d. Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, Sep. 2009.
- [65] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'95*, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
- [66] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. A4: automatically assisting android API migrations using code examples. *CoRR*, abs/1812.04894, 2018.
- [67] Thomas K Landauer. *Latent semantic analysis*. Wiley Online Library, 2006.
- [68] T.K. Landauer, P.W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse processes*, 25:259–284, 1998.

- [69] Alexander LeClair, Zachary Eberhart, and Collin McMillan. Adapting neural text classification for improved software categorization. *CoRR*, abs/1806.01742, 2018.
- [70] Edda Leopold and Jörg Kindermann. Text categorization with support vector machines. How to represent texts in input space? *Machine Learning*, 46(1-3):423–444, 2002.
- [71] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346, May 2017.
- [72] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014*, pages 83–94, New York, NY, USA, 2014. ACM.
- [73] 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.
- [74] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.
- [75] Erik Linstead, Sushil Krishna Bajracharya, Trung Chi Ngo, Paul Rigor, Cristina Videira Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.
- [76] 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.
- [77] 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.
- [78] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on API understanding and extended boolean model (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 260–270, 2015.
- [79] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16*, pages 653–656, New York, NY, USA, 2016. ACM.
- [80] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, August 1991.
- [81] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57 – 84, 2017.
- [82] 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.

- [83] 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.
- [84] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.
- [85] Catarina Miranda and Alípio M. Jorge. Incremental collaborative filtering for binary ratings. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '08, pages 389–392, Washington, DC, USA, 2008. IEEE Computer Society.
- [86] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. Documenting apis with examples: Lessons learned with the apiminer platform. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 401–408, Oct 2013.
- [87] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method? In *37th International Conference on Software Engineering*, pages 880–890, Piscataway, 2015. IEEE.
- [88] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522, New York, NY, USA, 2016. ACM.
- [89] 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.
- [90] P. T. Nguyen, J. Di Rocco, R. Rubel, 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.
- [91] 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.
- [92] 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.
- [93] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. Building Information Systems Using Collaborative-Filtering Recommendation Techniques. In Henderik A. Proper and Janis Stirna, editors, *Advanced Information Systems Engineering Workshops*, pages 214–226, Cham, 2019. Springer International Publishing.
- [94] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. Enabling heterogeneous recommendations in oss development: What's done and what's next in crossminer. In *Proceedings of the Evaluation and Assessment on Software Engineering*, EASE '19, pages 326–331, New York, NY, USA, 2019. ACM.

- [95] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. CrossRec: Recommending highly relevant third-party libraries - manuscript under review. *Journal of Systems and Software*, 2019.
- [96] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, pages 1050–1060, Piscataway, NJ, USA, 2019. IEEE Press.
- [97] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. Automated Classification of Metamodel Repositories: A Machine Learning Approach. In *Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (to appear), MODELS 2019, Munich, Germany, September 14-19, 2019*, 2019.
- [98] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An Automated Approach to Assess the Similarity of GitHub Repositories - manuscript under revision. *Software Quality Journal*, 2019.
- [99] 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.
- [100] 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.
- [101] Michael A. Nielsen. Neural networks and deep learning, 2018.
- [102] Haoran Niu, Iman Keivanloo, and Ying Zou. Api usage pattern recommendation for software development. *J. Syst. Softw.*, 129(C):127–139, July 2017.
- [103] Haoran Niu, Iman Keivanloo, and Ying Zou. API Usage Pattern Recommendation for Software Development. *Journal of Systems and Software*, 129(C):127–139, 2017.
- [104] 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.
- [105] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. Search-based software library recommendation using multi-objective optimization. *Inf. Softw. Technol.*, 83(C):55–75, March 2017.
- [106] Manos Papagelis and Dimitris Plexousakis. Qualitative analysis of user-based and item-based prediction algorithms for recommendation agents. In Matthias Klusch, Sascha Ossowski, Vipul Kashyap, and Rainer Unland, editors, *Cooperative Information Agents VIII*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [107] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep*, 2012.

- [108] Michael J. Pazzani and Daniel Billsus. *Content-Based Recommendation Systems*, pages 325–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [109] Joaquín Pérez-Iglesias, José R. Pérez-Agüera, Víctor Fresno, and Yuval Z. Feinstein. Integrating the probabilistic models BM25/BM25F into lucene. *CoRR*, abs/0911.5046, 2009.
- [110] Simone Pettigrew and Stephen Charters. Tasting as a projective technique. *Qualitative Market Research: An International Journal*, 11(3):331–343, 2008.
- [111] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1295–1298, May 2013.
- [112] 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.
- [113] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Prompter - turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, 21(5):2190–2231, 2016.
- [114] Azzurra Ragone, Paolo Tomeo, Corrado Magarelli, Tommaso Di Noia, Matteo Palmonari, Andrea Maurino, and Eugenio Di Sciascio. Schema-summarization in linked-data-based feature selection for recommender systems. In *Proceedings of the Symposium on Applied Computing, SAC '17*, pages 330–335, New York, NY, USA, 2017. ACM.
- [115] R. Ranjan, S. Sankaranarayanan, C. D. Castillo, and R. Chellappa. An all-in-one convolutional neural network for face analysis. In *2017 12th IEEE Int. Conf. on Automatic Face Gesture Recognition (FG 2017)*, pages 17–24, May 2017.
- [116] David Reby, Sovan Lek, Ioannis Dimopoulos, Jean Joachim, Jacques Lauga, and Stéphane Aulagnier. Artificial neural networks as a classification method in the behavioural sciences. *Behavioural Processes*, 40(1):35 – 43, 1997.
- [117] Peter C. Rigby and Martin P. Robillard. Discovering essential code elements in informal documentation. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 832–841, 2013.
- [118] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, July 2010.
- [119] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, Nov 2009.
- [120] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Trans. Softw. Eng.*, 39(5):613–637, May 2013.
- [121] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. DOI: 10.1007/978-3-642-45135-5.
- [122] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

- [123] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, Berlin, Heidelberg, 1996.
- [124] Riccardo Rubei, Claudio Di Sipio, Phuong T. Nguyen, Juri Di Rocco, and Di Ruscio. Recommending highly relevant StackOverflow posts with boosted multi-facet queries - manuscript under review. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, California, USA, 2019*.
- [125] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining Multi-level API Usage Patterns. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 23–32, Piscataway, 2015. IEEE.
- [126] Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari Sahraoui. Could We Infer Unordered API Usage Patterns Only Using the Library Source Code? In *23rd International Conference on Program Comprehension*, pages 71–81, Piscataway, 2015. IEEE.
- [127] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145:164 – 179, 2018.
- [128] Tefko Saracevic. Evaluation of evaluation in information retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '95*, pages 138–146, New York, NY, USA, 1995. ACM.
- [129] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based Collaborative Filtering Recommendation Algorithms. In *10th International Conference on World Wide Web*, pages 285–295, New York, 2001. ACM.
- [130] 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.
- [131] 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.
- [132] Claude E. Shannon. A mathematical theory of communication. *Mobile Computing and Communications Review*, 5(1):3–55, 2001.
- [133] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, 23(5):2622–2654, Oct 2018.
- [134] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, Jan 2004.
- [135] Daniel Svozil, Vladimir Kvasnicka, and Jiří Pospíchal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39:43–62, 11 1997.
- [136] Cédric Teyton, Jean-Rémy Falleri, Floréal Morandat, and Xavier Blanc. Find your library experts. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 202–211, Oct 2013.
- [137] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 204–213, New York, NY, USA, 2007. ACM.

- [138] F. Thung, D. Lo, and J. Lawall. Automated library recommendation. In *2013 20th Working Conf. on Reverse Engineering (WCRE)*, pages 182–191, Oct 2013.
- [139] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. Automatic recommendation of api methods from feature requests. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13*, pages 290–300, Piscataway, NJ, USA, 2013. IEEE Press.
- [140] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Int. Res.*, 37(1):141–188, January 2010.
- [141] Amos Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.
- [142] Secil Ugurel, Robert Krovetz, and C. Lee Giles. What's the code?: Automatic classification of source code archives. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '02*, pages 632–638, New York, NY, USA, 2002. ACM.
- [143] Saúl Vargas and Pablo Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the Fifth ACM Conference on Recommender Systems, RecSys '11*, pages 109–116, New York, NY, USA, 2011. ACM.
- [144] Saúl Vargas and Pablo Castells. Improving sales diversity by recommending users to items. In *Eighth ACM Conference on Recommender Systems, RecSys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014*, pages 145–152, 2014.
- [145] 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.
- [146] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-coverage API Usage Patterns from Source Code. In *10th Working Conference on Mining Software Repositories*, pages 319–328, Piscataway, 2013. IEEE.
- [147] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering, ICDE '04*, pages 79–, Washington, DC, USA, 2004. IEEE Computer Society.
- [148] Frank Wilcoxon. *Individual Comparisons by Ranking Methods*, pages 196–202. Springer New York, New York, NY, 1992.
- [149] Lili Wu, Sam Shah, Sean Choi, Mitul Tiwari, and Christian Posse. The browsmaps: Collaborative filtering at linkedin. In *RSWeb@RecSys*, volume 1271 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.
- [150] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Tag recommendation in software information sites. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, pages 287–296, Piscataway, NJ, USA, 2013. IEEE Press.
- [151] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 513–523, New York, NY, USA, 2002. ACM.

- [152] A. Zagalsky, O. Barzilay, and A. Yehudai. Example overflow: Using social media for code recommendation. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 38–42, June 2012.
- [153] Wei Zeng, An Zeng, Hao Liu, Ming-Sheng Shang, and Tao Zhou. Uncovering the information core in recommender systems. *Scientific reports*, 4:6140, 2014.
- [154] Guoqiang Zhang, B. Eddy Patuwo, and Michael Y. Hu. Forecasting with artificial neural networks:: The state of the art. *International Journal of Forecasting*, 14(1):35–62, 1998.
- [155] 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.
- [156] Zhi-Dan Zhao and Ming-sheng Shang. User-based collaborative-filtering recommendation algorithms on hadoop. In *Proceedings of the 2010 Third International Conference on Knowledge Discovery and Data Mining, WKDD '10*, pages 478–481, Washington, DC, USA, 2010. IEEE Computer Society.
- [157] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *23rd European Conference on Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer.
- [158] Bo Zhou, Xin Xia, David Lo, Cong Tian, and Xinyu Wang. Towards more accurate content categorization of api discussions. In *Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014*, pages 95–105, New York, NY, USA, 2014. ACM.