UNIVERSITÀ DEGLI STUDI DELL'AQUILA
Department of Information Engineering, Computer Science and Mathematics

Ph.D. Program in Information and Communication Technology
Curriculum Emerging computing models, software architectures,
and intelligent systems
XXXIV Cycle

Recommender systems to find and adopt reusable Open-Source Software components

SSD INF/01

Candidate
Riccardo Rubei

Doctoral Program Supervisor
Prof. Vittorio Cortellessa

Advisor
Prof. Davide Di Ruscio
Co-Advisor
Dr. Phuong T. Nguyen

a.y. 2020/2021

Abstract

To perform their daily tasks, developers intensively make use of existing resources by consulting open-source software (OSS) repositories. Such platforms contain rich data sources, e.g., code snippets, documentation, and user discussions, that can be useful for supporting development activities. Developing new software by leveraging existing open source components reduces their development effort considerably. The benefits resulting from the reuse of properly selected open-source projects are manifold, including that the system being implemented relies on open source code. In addition to source code, metadata available from different related sources, e.g., communication channels, and bug tracking systems, can be beneficial to the development life cycle if properly mined. Nevertheless, given many data sources and without being equipped with suitable machinery, developers would struggle to look for and approach the sources that meet their needs. In this respect, deploying systems that use existing data to improve developers' experience is of paramount importance.

Over the last decades, several techniques and tools have been promoted to provide developers with innovative features, aiming to improve development effort, cost savings, and productivity. This dissertation goes one step further to conceptualize a set of recommender systems to assist software programmers in different phases of the development process. By mining data from various sources, the systems provide developers with different types of recommendations, including similar projects to a project under development, upgrading plan for third-party libraries, or Stack Overflow posts relevant to the code being developed, to name a few. The recommender systems conceived in this dissertation have been empirically evaluated using real-world datasets. The experimental results show that they obtain high prediction performance compared to various state-of-the-art baselines, demonstrating their applicability in the field. Altogether, the ultimate aim is to help developers leverage the existing sources of information to improve their work efficiency and effectiveness.

Acknowledgements

I would like to express my deepest and sincerest gratitude to my supervisor, Prof. Dr. Davide Di Ruscio, for his expert guidance, constructive criticism, continuous support and encouragement throughout my study. It has been a privilege to work with him and I hope to have the chance to continue working under his supervision in the future.

I also gratefully thank my co-supervisor Dr. Phuong T. Nguyen for his supports, as well as for his patience. He has been always available for comments and suggestions.

A special thank to Dr. Juri Di Rocco for his help, in particular during the implementation phase of the software tools. His great expertise is always a good source of inspiration for me.

I am also grateful to my friend and colleague Claudio Di Sipio for the continuous support as we have worked side by side during this journey.

Also a big thank goes to my dear friends Roberta, Walter, Andrea, Francesco, Riccardo, Vincenzo, Maria Teresa, who sustained me during my study and who never let me go mad.

Finally I would like to thank my mother, Ariel and my family, without them I could not have been able to complete my study.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Open-source software (OSS) is computer program available in source code, in which the code and other rights are provided under a license that allows users to study, change, and improve the software for free. Open-source software forges, such as GitHub or Maven, offer many software projects that deliver stable and well-documented products. Most OSS forges typically sustain vibrant user and expert communities which in turn provide decent support, both for answering user questions and repairing reported software bugs. Moreover, OSS platforms are also an essential source of consultation for developers in their daily development tasks [4]. Code reusing is an intrinsic feature of OSS, and developing new software by leveraging existing open source components allows one to considerably reduce their development effort. 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"* [5]. 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 life cycle if being properly mined [6]. Nevertheless, given a plethora of data sources, developers would struggle to look for and approach the sources that meet their need without being equipped with suitable machinery. Such a process is time-consuming since the problem is not a lack, but in contrast, an overload of information coming from heterogeneous and rapidly evolving sources. In particular, when developers join a new project, they have to typically master a considerable number of information sources (often at a short time) [7]. In this respect, the deployment of systems that use existing data to improve developers' experience is of paramount importance.

Recommender systems are a crucial component of several online shopping platforms, allowing business owners to offer personalized products to customers [8]. The development of such systems has culminated in well-defined recommendation algorithms, which in turn

prove their usefulness in other fields, such as entertainment industry [9], or employment-oriented service [10]. Recommender systems in software engineering [11] (RSSE hereafter) have been conceptualized on a comparable basis, i.e., they assist developers in navigating large information spaces and getting instant recommendations that are helpful to solve a particular development task [6, 12]. Typical scenarios examples include, among others, code snippets [12–14], topics [15, 16], third-party components [17, 18], documentation [19, 20], to name a few.

1.1 Challenges and motivations

While research on recommender systems has gained momentum in recent years, there exist various challenges when it comes to designing and implementing systems to support software developers [21]. In particular, our work is motivated by the following challenges:

- *Measuring similarities among software systems:* Considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail. In this respect, it is necessary to conceptualize proper mechanisms to capture the relationship among the contributing artifacts, and eventually compute the similarities.

- *Incorporating user information to empower recommender systems:* Traditional support systems rely on information collected from the cloud to make their final decision. So far, feedback from users has not been considered as a part of the recommendation process. However, finding the right way to embed user information with the ultimate of improving the recommendation accuracy remains as an open research issue.

- *Providing developers with more relevant pieces of information related to the task under development:* When working with a software project, developers have to consult various sources to search for relevant components that meet their needs. Such a process is time consuming and prone to errors, triggering the need for proper searching mechanisms that can assist developers in approaching the most relevant resources.

- *Mining sequential and time-series data:* In open source software repositories, there exist time-series artifacts which are the result of the interaction between developers and hosting platforms, e.g., the evolution of a software project in GitHub over the course of time. Such type of data, once being properly mined, can provide developers/modelers with useful recommendations, enabling them to fulfill their tasks.

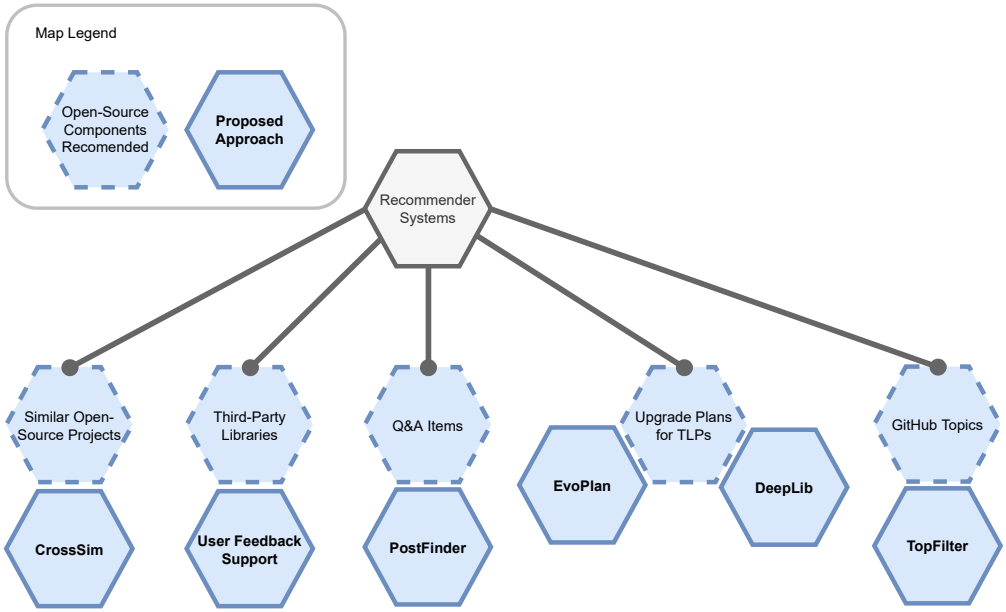


Figure 1.1: Dissertation conceptual map.

1.2 Research objectives

Figure 1.1 presents a conceptual map for the work presented in this dissertation. Each of the blocks presents a particular recommender system providing reusable open-source software components. The dotted hexagons mark the open-source artefacts related to a challenge, while the continuous ones represent our proposed tool-supported approach to tackle the corresponding challenge. We identify the following research objectives.

Modeling the OSS ecosystem to compute similarity

In OSS forges like GitHub, there are several connections and interactions, such as development commit to repositories, user star repositories, or projects contain source code files, to mention a few. We come up with the application of a graph-based representation to capture the semantic features among various actors, and consider their intrinsic connections.We modeled the community of developers together with OSS projects, libraries, source code, and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependencies and implications on the others. Graphs allow for flexible data integration and facilitates numerous similarity metrics [22]. We adopt a graph-based representation to deal with the project similarity issue because some of the co-authors already addressed a similar problem in the context of Linked Data. The analogy of the two problems inspired us to apply the similarity technique already developed [23] to calculate the similarity of representative software projects.

Incorporating user feedback

During their daily routine, developers often have to deal with a plethora of resources, attempting to search for relevant artefacts that can be added to the project underdevelopment. This kind of information overload may render developers overwhelmed, thus undermining their productivity and efficiency. Recommender systems are effective means of easing such a burden, providing developers with relevant items for the current programming contexts, e.g., third-party libraries (TPLs), API calls, or code snippets. By focusing on TPLs there has been no work to allow for the integration of tailored feedback mechanisms with which users can conveniently accept or discard the recommended libraries. This motivates us to develop a tool to handle explicit user feedback, including positive, negative, and additive. Thus, further than accepting or discarding the recommended TPLs, users can also endorse libraries that, in their opinion, are relevant for the current context, even though they are not included in the provided recommendations.

Finding relevant Stack Overflow 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. Stack Overflow provides developers with a broader insight into API usage as well as useful code examples. Given the circumstances, tools and techniques for mining Stack Overflow are highly desirable. We aim to propose an approach that analyses the project under development to extract suitable context, and allows developers to retrieve messages from Stack Overflow being relevant to the API function calls that have already been invoked. The system augments posts with additional data to make them more exposed to queries. On the client side, it boosts the context code with various factors to construct a query containing information needed for matching against the stored indexes. Multiple facets of the data available are used to optimize the search process, with the ultimate aim of recommending highly relevant SO posts.

Upgrading third-party libraries

To keep their code up-to-date with the newest functionalities as well as bug fixes offered by third-party libraries, developers often need to replace an old version of third-party libraries (TPLs) with a newer one. However, choosing a suitable version for a library to be upgraded is complex and susceptible to error since it is crucial to maintain a harmonious relationship with other libraries. So far, Dependabot is the only tool that supports library upgrades; yet, it targets only security fixes and singularly analyses libraries without considering the whole set of related libraries. We deal with the recommendation of different upgrade plans given a pair

of library-version as input. Moreover, we also mine the development history of projects to build migration matrices to train deep neural networks. Once being trained, the networks are then used to forecast the subsequent versions of the related libraries. As input, the system accepts a set of library versions and returns as output a set of future versions that developers should upgrade the libraries to.

Tagging GitHub repositories

In the context of software development, GitHub has been at the forefront of platforms to store, analyse and maintain many software repositories. Topics have been introduced by GitHub as an effective method to annotate stored repositories. However, labeling GitHub repositories should be carefully conducted to avoid adverse effects on project popularity and reachability. We support developer with a novel approach to select suitable topics for GitHub repositories being created. We build a project-topic matrix and applied a syntactic-based similarity function to recommend missing topics by representing repositories and related topics in a graph.

1.3 Structure of the dissertation

In Chapter 2, we review notable related work in the domain of recommender systems to support developers. Chapter 3 describes CrossSim, an approach for computing similarities among open source projects. Chapter 4 presents a practical solution to handling user feedback, with the ultimate aim of increasing the capability of existing recommender systems to provide relevant third-party libraries. Chapter 5 presents PostFinder, a recommender system to support developers by giving them insight excerpted from Stack Overflow according to their context. In Chapter 6, we conceive EvoPlan and DeepLib, two recommender systems to support developers in upgrading their third-party libraries. Chapter 7 describes TopFilter, an automated tool to recommend topics for GitHub repositories. TopFilter is an extension of an existing tool based on a Multinomial Naive Bayesian, by employing a collaborative filtering technique to recommend also featured topics. Chapter 8 summarizes and discusses the contributions of this work as well as the challenge addressed and future work.

Chapter 2

Literature review

2.1 Recommending similar open-source projects

Garg *et al.* [24] proposed 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) [25], 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. CLAN was developed to detect similar Java applications by exploiting the semantic layers corresponding to package class hierarchies [26]. CLAN works based on the document framework for computing similarity, semantic anchors, e.g., those that define the documents' features.

To detect clone among Android apps, Wang *et al.* propose WuKong [27] 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 [28, 29] between their feature vectors.

Evaluations on more than 100,000 Android apps collected from five Chinese app markets showed that the approach can effectively detect cloned apps [27]. CrossSim is also able to deal with low-level features as by WuKong if such features are integrated into the graph presented in Section 3.2.

Lo *et al.* develop TagSim,[1] a tool that leverages tags to characterize applications and then to compute similarity between them [30]. Tags are terms that are used to highlight the most important characteristics of software systems [31] 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 [28, 29] between the two vectors. To evaluate TagSim, more than a hundred thousands of projects were collected and analyzed [30]. A total of 20 queries were used to study the performance of the algorithm in comparison with CLAN. The experimental results show that TagSim helps achieve better performance in comparison to CLAN [26].

Being inspired by CLAN, Linares-Vásquez *et al.* develop CLANdroid for detecting similar Android applications with the assumption that similar apps share some semantic anchors [32]. 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* specification.[2] 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

[1] For the sake of clarity, in this chapter we give a name for the algorithms that have not been originally named

[2] https://developer.android.com/guide/topics/manifest/manifest-intro.html

been performed to study which semantic anchors are more effective [32]. 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 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 [33, 12].

With the aim of finding apps with similar semantic requirements, SimApp was conceived to exploit high-level metadata collected from apps markets [34]. If two apps implement related semantic requirements then they are seen as similar. Each mobile application is modeled by a set of features, and 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 [35], and feature vectors are then clustered to find similar apps. Locality Sensitive Hashing [36] 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 was applied to find similar apps by different developers and groups of apps by the same developer with high code reuse (rebranded apps). LibRec is a tool that assists developers in leveraging existing libraries [18]. It suggests the inclusion of libraries that may be useful for a given project using a combination of rule mining and collaborative filtering techniques. *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. The libraries included by these projects are used as recommendations based on a score computed according to their popularity.

	Using source code				Using metadata					
	MUDABlue [24]	**CLAN** [26]	**CLANdroid** [32]	**WuKong** [27]	**SimApp** [34]	**AnDarwin** [35]	**TagSim** [30]	**LibRec** [18]	**RepoPal** [37]	
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 *README.MD* files, used to describe the functionalities of an open source project
Tags	–	–	–	–	–	–	✓	–	–	The tags that are used by OSS platforms, e.g. SourceForge to classify and characterize an OSS project
Updates	–	–	–	–	✓	–	–	–	–	The newest changes made to the considered applications
Permissons	–	–	✓	–	✓	–	–	–	–	This feature is available by mobile apps. It specifies the permission of an app to handle data in a smartphone
Screenshots	–	–	–	–	✓	–	–	–	–	This feature is available by mobile apps. It is a picture representing an app
Contents	–	–	–	–	✓	–	–	–	–	Each app has content rating to describe its content and age appropriateness
Size	–	–	–	–	✓	–	–	–	–	Some similarity metrics assume that two apps whose size is considerably different cannot be similar
Reviews	–	–	–	–	✓	–	–	–	–	All user reviews for an app are combined in a document
Intents	–	–	✓	–	–	–	–	–	–	For a mobile app, an intent is description for an operation to be performed
Sensors	–	–	✓	–	–	–	–	–	–	In mobile devices, sensors can provide raw data to monitor 3-D device movement or positioning, or changes in the environment. A set of features can be built from sensors to characterize an app
Used techniques										
TDM & LSA	✓	✓	✓	–	–	–	–	–	–	TDM [38] and LSA [39] 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 [28, 29] is widely used in several algorithms for computing similarities among vectors
JCS	–	–	–	–	–	✓	–	–	✓	Jaccard index [40] is used for computing similarity between two sets of elements

Table 2.1: Software similarity algorithms and their features.

A summary of all the similarity metrics introduced in Section 3.1 and Section 2.1 is depicted in Table 2.1. 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, MUDABLUE, 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 [41], SimApp [34], TagSim [30].

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 [24]. 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. The matching of words in the descriptions as well as source code to compute similarity is considered to be ineffective as already stated in [26]. To overcome this problem, the application of a synonym dictionary like WordNet [42] 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 [34] 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. The aforementioned 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. 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.

2.2 Recommending third-party libraries

The BRAID framework [43] has been built on top of an existing API recommender system to allow for the specification of a user-based query. Furthermore, it combines LTR and Active Learning techniques to adapt the original recommendations according to the user query.

Elixir [44] supports user feedback expressed on items' explanations using pairwise learning. The list of pair ratings-explanations is encoded to user-specific latent vectors that have been used to improve a well-founded recommender system based on random walk with restart technique.

Wang *et al.* [45] proposed Active Code Search to incorporate explicit user feedback. Given a recommended list, the user can iteratively express feedback on the proposed items to rearrange the final rank. Then, a *refined engine* is employed to embody the collected feedback using NLP techniques. Similarly, WebAPIRec [46] employs a personalized ranking model to recommend an ordered list of web APIs given the project profile. The model is fed with historical data of APIs usages to retrieve the ranked list of APIs. Recently, several studies employed federated learning to collect user feedback by preserving sensitive data [47, 48].

Ouni *et al.* [49] make use of multi-objective algorithm by relying on library usage history. Given a library, the system recommends new items by maximizing the semantic similarity. LibCUP [50] uses a clustering approach to identifying co-usage patterns and suggests similar TPLs. Similarly, LibD [51] provides libraries to mobile apps using a clustering technique.

To the best of our knowledge, the possibility of providing positive, negative, and additive feedback in the context of TPLs recommender systems is still underinvestigated. Our approach allows users to provide feedback on recommended TPLs. The experiments performed on CROSSREC are encouraging, and we plan to consider other TPLs recommender systems.

2.3 Recommending Stack Overflow posts

Stack Overflow 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. We review various studies that share some commonalities with PostFinder as follows.

Zagalsky *et al.* [52] introduce *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 PostFinder is able to retrieve full posts that are related to the user context.

Seahawk [53] has been developed to retrieve SO discussions, which are linked to the source code being developed. The search mechanism exploits code similarity techniques essentially based on TF-IDF. The PostFinder search mechanisms are instead based on different boosting features that are considered when creating queries and when executing them atop of Apache Lucene.

de Souza *et al.* [54] present a tool being able to recommend a ranked list of pairs of SO questions and answers based on the user query, which consists of a list of terms. Furthermore, the approach also allows one to classify SO posts according to defined labels like *how-to*, *debug corrective*, etc. The main difference with PostFinder relies on the way queries are defined. In particular, in PostFinder queries consist of the whole developer context, instead of only lists of terms explicitly defined by the user.

AnswerBot [55] is an automatic approach aiming to summarize answers coming from Stack Overflow, given a specific question. The retrieval of relevant question is performed by exploiting word embeddings techniques combined with classical IDF metrics. To reduce information overloading, AnswerBot filters the answers by following different criteria, i.e., query related features, paragraph content features, and user related features.

Rigby and Robilliard [56] propose ACE, a tool 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. Different from PostFinder, 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).

MFISSO [57] is a system that exploits natural language processing and clustering techniques to obtain facets from the user's query, i.e., concepts expressed by the query. There are SO features with 8 facets, and seven of these are static and determined by NLP techniques and using Apache Lucene indexes. Clustering is applied to retrieve the dynamic facet by labeling titles, text, and tags of the SO posts. Once MFISSO retrieves the initial results, a final user can refine this search by interacting with the system, i.e., changing the facets to be considered. Reversely, PostFinder extracts the facets directly from source code.

Prompter [6] has been conceived to extend Seahawk by focusing on the query creation phase and on the model, which is used to rank the retrieved posts. Differently from Prompter, PostFinder does not rely on public search engines, i.e., Google and Bing to retrieve SO messages. Further than focusing on the query creation phase, PostFinder 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.

PostFinder distinguishes itself from current approaches that deal with the mining of Stack Overflow as it addresses different phases of the whole searching process, i.e., Index Creation, Query Creation and Query Execution. To this end, PostFinder attempts to effectively exploit the well-defined indexing and searching mechanisms provided by Lucene to increase the exposure of queries to the indexed data. Nevertheless, we still believe that more investigations are needed to further improve PostFinder's performance, e.g., by better employing the boosting scheme. This is considered as an open research issue.

Table 2.2: Summary of SO mining approaches.

Tool/Engine	Code search	Mining SO	Availability
MFISSO [57]	✓	✓	✗
PROMPTER [6]	✓	✓	✗
AnswerBot [55]	✗	✓	✗
Google/SO search	✗	✓	✓
Searchcode [58]	✗	✗	✓
Krugle [59]	✗	✗	✓
PublicWWW [60]	✗	✓	✓
ProgramCreek [61]	✗	✗	✓
FaCoY [62]	✓	✓	✓

Table 2.2 summarizes the functionalities as well as the availability of the related Stack Overflow search engines. The table also reveals the rationale behind the selection of FaCoY as baseline. The *Availability* column reports if the considered tool is available for download, including replication packages, and the legacy support provided by the authors. As PostFinder performs search on source code, *Code search* identifies tools that allow directly the usage of source code in the query. Furthermore, since some of the examined approaches do not search over Stack Overflow expressed by *Mining SO*, a comparison with them is not fair. Overall, the table suggests that only the comparison with FaCoY is feasible at the moment.

2.4 Recommending upgrade plans for third-party libraries

Recently, the issue of recommending development of third-party libraries and API usage has been intensively studied [63]. We review notable recommender systems by focusing on those related to the adoption of TPLs and API migrations.

LibRec [18] employs a combination of rule mining and collaborative filtering strategies to retrieve libraries considering similar projects to the one given as input. Ouni *et al.* [49] develops LibFinder that uses a multi-objective algorithm to detect semantic similarity in source code. CrossRec [17] assists developers in selecting suitable TPLs. The system

exploits a collaborative filtering technique to recommend libraries by relying on the set of dependencies, which have been included in the project being developed.

The usage of domain-specific category (DSC) concept has been investigated to foster TPL maintenance [64]. By relying on a rule mining technique, the approach is capable of categorizing GitHub projects exploiting labels available on the Maven Central repository. LibSeek [65] employs the matrix factorization (MF) technique to predict relevant TPLs for mobile apps. It adopts an adaptive weighting scheme to reduce the skewness caused by popular libraries. Furthermore, the MF-based algorithm is used to integrate neighborhood information by computing the similarity of libraries contained in the matrix.

Req2Lib [66] has been recently proposed to recommend TPLs given textual description of project requirements. The tool employs a seq2seq LSTM which is trained with description and libraries belonging to configuration file. Additionally, a domain-specific embedding model obtained from Stack Overflow is used to encode words in high-dimensional vectors.

All the previously mentioned systems can recommend libraries that can be added in the project being developed. However, they do not provide suggestions that can help upgrade already included dependencies as done by DeepLib.

Xu *et al.* propose Meditor [67] to analyze GitHub commits to extract migration-related (MR) changes by mining *pom.xml* files. Once MR updates have been found, the tool employs the WALA framework to check their consistency by analyzing the developer's context and apply them directly. Apiwave [68] infers and retrieves relevant information related to TPLs, i.e., popularity and migration data. The tool uses two different modules to discover the popularity by analyzing import statements of projects, i.e., the removal of certain API decreases its popularity. Additionally, the system can infer migration data from each API replacement.

Teyton *et al.* [69] propose an approach that relies on a graph-based structure to address the migration of TPLs. The tool makes use of a token-based filter applied on pom.xml to discover libraries' information i.e., names and versions. Afterwards, each TPL is encoded as a node on the graph and edges express four different visual patterns to suggest the most suitable migration changes given the input library.

SimilarAPI [70] suggests alternative TPLs by exploiting an unsupervised ML approach. Given a set of projects crawled from GitHub, the tool is able to extract API call sequences by using a well-founded partial program analysis Java tool. Then, the obtained information is encoded by using skip thoughts, an RNN capable of embedding the word semantics in a vector space. Given input TPLs, SimilarAPI retrieves the most similar ones with a ranked list of APIs.

Fazzini *et al.* propose APIMigrator [71] to support the migration of APIs in Android apps. Starting from API usage (AU) information extracted from the documentation, the tool can obtain generic migration examples that represent historical migration data. Then, generic migration patches are extracted from these explanatory AU to migrate the target app. APIMigrator performs the actual migration by mapping the list of obtained patches one at a time to avoid possible issues due to the overlapping.

LibBandAid [72] is an approach to automatic generation of updates for TPLs in Android apps. Given an app with an outdated library and a newer version of the library, the tool recommends how to update the old library in a way that guarantees backward compatibility.

A recent work [73] attracts the community attention over the *migration awareness* problem as well as the efforts required to apply actual changes. As the first step, the authors proposed a model to detect TPLs evolution by excerpting migration data from GitHub projects. The proposed model is based on the assumption that systems with more dependencies tend to have more frequent updates. Additionally, the authors conducted a user study to measure the developer's behavior considering the two main migration awareness mechanisms, i.e., security advisories and new releases announcement. The findings of the work are: *(i)* the majority of the software systems rarely update the older but reliable libraries; *(ii)* security advisories provide incomplete solutions; and *(iii)* developers consider the migration task as a non crucial task for the development.

Differently from the aforementioned approaches, our proposed approach DeepLib is able to learn from what other projects have done to recommend the next upgrades that maintainers should operate on one or more libraries already included in their project. DeepLib is even able to recommend removals of dependencies according to migrations that have been performed.

2.5 Recommending GitHub topics

Immediately after the introduction of topics in the GitHub platform, the Repo-Topix tool was presented [74]. Such a tool relies on parsing the README files and the textual content of a given GitHub repository to suggest topics automatically. As a first step, the tool applies standard NLP techniques on the input artifacts. Then, it filters an initial set of topics by exploiting the TF-IDF scheme and a regression model to exclude *"bad"* topics. As the final step, Repo-Topix computes a custom version of the Jaccard distance to discover additional similar topics. A rough evaluation based on the n-gram ROUGE-1 metrics has been conducted by counting the number of overlapping units between the recommended topics and the repository description. Nevertheless, a replication package with the complete

dataset and the source code of the tool is not available, and this hampers further investigations and comparisons.

A collaborative topic regression (CTR) model has been proposed to extract topics from a given GitHub repository [75]. The final aim is to recommend other similar projects given the input one. For a pair of user-repository, the approach uses a Gaussian model to compute matrix factorization and extract the latent vectors given a pre-computed matrix rating. Additionally, a probabilistic topic modeling is applied to find topics from the repositories by analyzing high frequent terms. The approach was evaluated by conducting five-fold cross-validation on a dataset composed of 120,867 repositories. Such evaluation considers user-repository pairs that have at least 3 watches. Differently from TopFilter, this approach can recommend GitHub repositories that are relevant with respect to the topics of the input repository.

Lia *et al.* [76] propose a user-oriented portrait model to recommend a set of GitHub projects that can be of interest for a given user. An initial set of labels is obtained by running the LDA algorithm on the textual elements of a repository, i.e., issues, commits, and pull requests. Then, the approach exploits a project familiarity technique that relies on the user's behavior, considering the different repositories operation. Such a strategy enables the collaborative filtering technique that exploits two kinds of similarity, i.e., attribute and social similarity. The former takes into account personal user information such as company, geographical information and the time when the account has been created. The latter computes similarity scores considering the proportion of items contributed by the user. The approach was evaluated by considering 80 users with an average of 1,894 different behaviors for each one. By considering the first two months of activity in 2016 as a test set, the assessment shows that the approach improves the performances in terms of precision, recall, and success rate. Though both TopFilter and the presented work [76] make use of collaborative-filtering techniques, the former is designed to recommend topics to be assigned to an input GitHub repository, whereas the latter recommends GitHub repositories that can be of interest for a given user.

A model-based fuzzy C-means for collaborative filtering (MFCCF) has been proposed [77] to recommend relevant human resources during the GitHub project development. Similarly to our approach, the proposed model encodes relevant information about repositories in a graph structure and extracts from it a sparse test sub-graph. This is a preparatory phase to enable the fuzzy C-means clustering technique. Using the computed sparse sub-graph as the centre of the cluster, the model can handle the sparsity issue that generally arises in the CF domain. Then, MFCCF computes the Pearson Correlation for each pair user-item belonging to a cluster and retrieves the top-N results. The evaluation was performed using

the GHTorrent dump to collect the necessary information. Using ten projects as the testing dataset, the results of MFCCF are compared with the ones chosen by the human resource department of the considered company. The results demonstrate the effectiveness of the approach with an accuracy of 80% on average.

The REPERSP tool [78] recommends GitHub projects by exploiting users' behaviour. As the first step, the tool computes the similarities between projects using the TF-IDF weighting scheme to obtain the content similarity matrix. Additionally, REPERSP captures the developer's behaviour by considering her activity on GitHub, i.e., create, star, and fork actions over projects. A different value is assigned for each type of action to create a user-project matrix. Finally, the tool combines the two similarity matrices to deliver the recommended projects. To assess the quality of the work, REPERSP was compared with the traditional collaborative filtering techniques, i.e., user-based and item-based. The study was conducted over two groups with different users, projects, and purposes. The results show that the proposed tool outperforms the considered baseline in terms of accuracy, precision, and recall.

Besides GitHub projects, tags and topics are successfully used in different contexts, i.e., in social networks. Purushotham *et al.* [79] propose a hierarchical Bayesian model that relies on a topic model to provide final users with relevant items. By combining LDA and matrix factorization techniques, the proposed model is able to reduce the sparsity problem that typically occurs when the collaborative filtering is employed. After this preprocessing phase, the hierarchical Bayesian model is tuned with several parameters to maximize the prediction performances. The models were evaluated by using two large real-world datasets tailored for music and bookmark recommendations. The experimental results show that the proposed model outperforms the classical CTR model with respect to recall. TopFilter distinguishes itself from the aforementioned approaches in two main aspects. On the one hand, it improves the capability of a well-known classification approach, i.e., MNBN, by extending the support to non-featured GitHub topics. On the other hand, it demonstrates the efficiency of a classification of GitHub topics based on a collaborative-filtering algorithm.

Chapter 3

Computing similarity between open-source software projects

Open source software (OSS) repositories contain a large amount of data, which can be of high-value when developing new software systems without reimplementing already in place functionalities. The benefits resulting from the reuse of properly selected open source projects are manyfold 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"* [5]. 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 [80].

Mining OSS repositories. In recent years, considerable effort has been made in the domain of mining software repositories to conceive techniques and tools to help developers mine and cope with the large amount of data available in OSS repositories. The main goal is to support software developers by providing meaningful recommendations. This is synthesized by relying on existing systems and past experiences. For instance, when developers work on a new project, it is possible to provide them with suggestions about which libraries they should use, based on a comparison with similar projects [81, 18]. Moreover, it is possible to empower IDEs by means of tools that continuously monitor the developer's activities and contexts in order to activate dedicated recommendation engines [80].

Software Similarity. Copying and pasting is a common practice in software development [82]. One of the main countermeasures to deal with software clone is to measure structural similarity among programs, aiming to evaluate their correctness, style as well as uniqueness [83]. The concept of similarity is a key issue in various contexts, such as detecting cloned code [84–87] software plagiarism [88], or reducing test suite in model-based

testing [89, 90]. Nevertheless, a globally exact definition of similarity is hard to come by since depending on the method used to compare items, various types of similarity may be identified. According to Walenstein *et al.* [91], a workable common understanding for software similarity is as follows: *"the degree to which two distinct programs are similar is related to how precisely they are alike."*

In the context of open source software, two projects are deemed to be similar if they implement some features being described by the same abstraction, even though they may contain various functionalities for different domains [26]. Given a software system being developed, we are interested in *finding a set of similar OSS projects* with respect to different criteria, such as external dependencies, application domain, or API usage [92, 81]. This type of recommendation is beneficial to the development since it allows developer to learn how similar projects are implemented. Understanding the similarities among open source software projects allows for reusing of source code and prototyping, or choosing alternative implementations [37, 93]. To aim for software quality [5], developers normally build their projects by learning from mature OSS projects having comparable functionalities [5]. Furthermore, similarity has been used as a base by both content-based and collaborative-filtering recommender systems to choose the most suitable items for a given user [93]. In this sense, it is necessary to equip software developers with suitable machinery which facilitates the similarity search process.

Nevertheless, measuring similarities among software systems has been considered as a daunting task [34, 26]. Furthermore, considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail. In this sense, choosing the right technique to compute software/projects similarity is a question that may arise at any time.

Goal of the chapter. First, we propose CrossSim, an approach that allows us to represent different project characteristics belonging to different abstraction layers in a homogeneous manner, then SimRank [94], a graph algorithm is applied to compute the similarities among nodes. Second, we present a thorough literature review on various techniques for computing software similarities. In this sense, this chapter makes the following contributions:

- proposing a novel approach to represent the OSS ecosystem by exploiting its mutual relationships;

- developing an extensible and flexible framework for computing similarities among open source software projects; and

- validating the performance of our proposed framework by means of three different existing approaches to computing software similarities, namely MUDABLUE [24], CLAN [26], and REPOPAL [37];
- presenting a comprehensive literature review on the field of software similarity measurement.

Structure of the Chapter. Section 3.1 motivates our work by providing an introduction to three approaches for detecting similar software applications and open source projects. In Section 3.2, CrossSim and its architecture are presented. Section 3.3 describes the evaluation conducted on a real GitHub dataset. Afterward, the experimental results are reported and analyzed in Section 3.4. The chapter is finally concluded in Section 3.5, where a summary of the work done as well as future work are discussed.

3.1 Background

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 [37, 93]. Also, recommender systems rely heavily on similarity metrics to suggest suitable and meaningful items for a given item [18, 93, 95]. 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 [34]:

- *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 chapter as a means to distinguish existing approaches with regards to the input information used for similarity computation. In this section, we provide a summary of three techniques for computing similarities among open source projects, i.e., MUDABLUE [24], CLAN [26], and REPOPAL [37].

3.1.1 MUDABlue

Together with a tool for automatically categorizing open source repositories, Garg *et al.* [24] 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) [25], 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, MUDABLUE 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 [96], and the SVM based method by Ugurel *et al.* [97]. The evaluation shows that MUDABLUE outperforms these observed algorithms with respect to precision and recall.

3.1.2 CLAN

McMillan *et al.* propose CLAN, an approach for automatically detecting similar Java applications by exploiting the semantic layers corresponding to package class hierarchies [26]. 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 [38]. 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 SourceForge[1] applications and shows that it qualifies for the detection of similar applications [26].

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 MUDABLUE as it considers only API calls to represent software systems. As shown later in this chapter, CLAN is more efficient than MUDABLUE as it produces recommendations in a much shorter time.

3.1.3 RepoPal

In contrast to many previous studies that are generally based on source code [24, 26, 98], RepoPal [37] is a high-level similarity metric and takes only repositories metadata as its input. With this approach, two GitHub[2] 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:

$$sim_f(r_i, r_j) = CosineSim(f_i, f_j) \tag{3.1}$$

[1] SourceForge: https://sourceforge.net/
[2] About 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 [40] 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:

$$sim_s(r_i, r_j) = \frac{1}{|U(r_i)| \cdot |U(r_j)|} \sum_{\substack{u_k \in U(r_i) \\ u_l \in U(r_j)}} sim_u(u_k, u_l) \tag{3.2}$$

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:

$$sim_t(r_i, r_j) = \frac{1}{|U(r_i) \cap U(r_j)|} \sum_{u_k \in U(r_i) \cap U(r_j)} \frac{1}{|T(u_k, r_i, r_j)|} \tag{3.3}$$

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

$$sim(r_i, r_j) = sim_f(r_i, r_j) \times sim_s(r_i, r_j) \times sim_t(r_i, r_j) \tag{3.4}$$

REPOPAL has been evaluated against CLAN using a dataset of 1,000 Java repositories [37]. Among them, 50 were chosen as queries. *Success Rate*, *Confidence* and *Precision* were used as the evaluation metrics. Experimental results in the work show that REPOPAL produces better quality metrics than those of CLAN.

3.2 CrossSim

We come to the conclusion that a representation model that incorporates various features and semantic relationships is highly beneficial to similarity computation. We find inspiration from a related field, namely Linked Data and Semantic Web [99], to realize such a model. Linked Data is a representation method that allows for the interlinking and semantic querying of data. The proliferation of Linked Data in recent years has enabled numerous applications. Two prominent examples are Linked Data for building music platform as by BBC Music [100] and for developing map application as by OpenStreetMap [101]. The core of Linked Data is an RDF[3] graph that is made up several nodes and oriented links to represent the

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

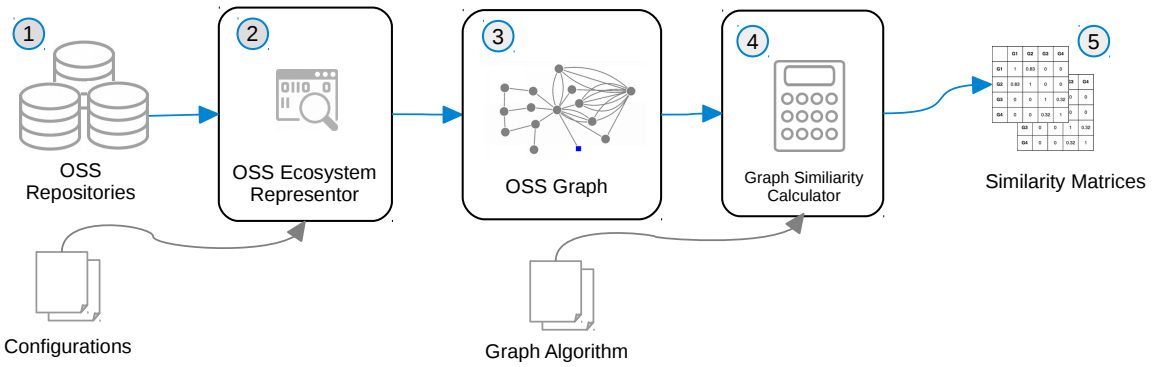


Figure 3.1: Overview of the CrossSim approach.

semantic relationships among various artifacts. Thanks to this feature, the representation paves the way for various computations. One of the main applications of RDF is similarity computation for supporting recommender systems [23, 95].

By considering the analogy of typical applications of RDF graphs and the problem of detecting the similarity of open source projects, we developed CrossSim (**C**ross project **R**elationships for computing **O**pen **S**ource **S**oftware **Sim**ilarity) [92], an approach that makes use of graphs for modeling different types of relationships in the OSS ecosystem [81]. Similar to RDF graphs, the representation model can capture the semantic features and considers the intrinsic connections between various actors. Specifically, the graph model has been chosen since it allows for flexible data integration and facilitates numerous similarity metrics [22]. We consider the community of developers together with OSS projects, libraries, source code, etc., and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependency and implication on the others. There, several connections and interactions prevail, such as developers commit to repositories, users star repositories, or projects contain source code files, just to name a few. The graph representation allows for the computation of similarities among nodes by means of several graph algorithms.

The architecture of CrossSim is depicted in Figure 3.1. In particular, the approach imports project data from existing OSS repositories ① and represents them into a graph-based representation by means of the *OSS Ecosystem Representor* module ②. Depending on the considered repository (and thus to the information that is available for each project) the graph structure to be generated has to be properly configured. For instance in case of GitHub, specific configurations have to be specified in order to enable the representation in the target graphs of the stars assigned to each project. Such a configuration is “forge” specific and specified once, e.g., SourceForge does not provide the star based system available in GitHub. The *Graph Similarity Calculator* module ④, depending on the similarity function to be applied, computes similarity on the source graph-based representation of the input

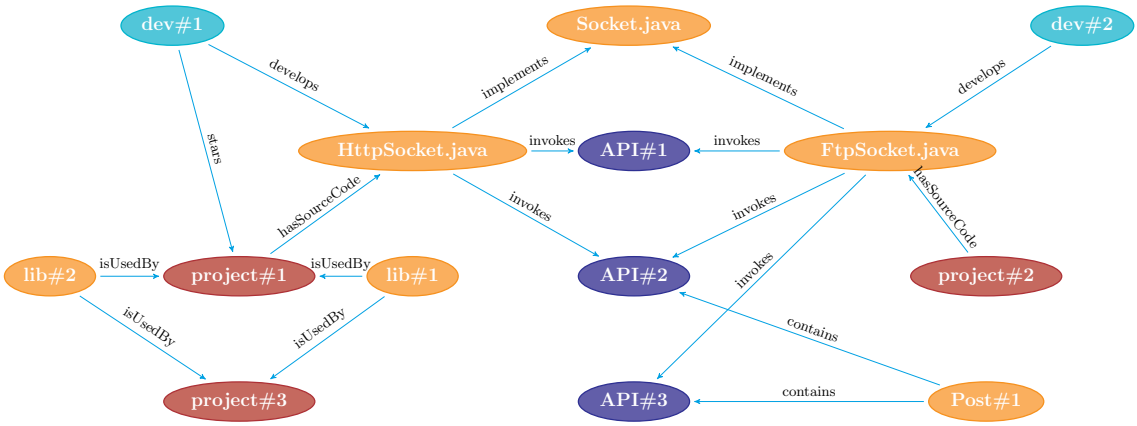


Figure 3.2: A Knowledge Graph for the representation of the OSS ecosystem.

ecosystems to generate matrices ⑤ representing the similarity value for each pair of input projects.

A detailed description of the proposed graph-based representation of open source projects is given in Section 3.2.1. Details about the implemented similarity algorithm are given in Section 3.2.2.

3.2.1 A knowledge graph for the OSS ecosystem

By means of the graph representation, we transform the relationships among various artifacts in the OSS ecosystem into a mathematically computable format. The representation model considers different artifacts in a united fashion by taking into account their mutual, both direct and indirect, relationships as well as their co-occurrence as a whole.

The following relationships are used to build graphs representing the OSS ecosystems and eventually to calculate similarity exploiting the algorithm presented in the next section. These vocabularies can also be flexibly augmented upon additional features of input data, even though our current definition seems to cover the most prevailing relationships [102].

- *commits* $\subseteq$ *Developer* $\times$ *Project*: this relationship represents the projects or libraries that a user contributes to the development;
- *contains* $\subseteq$ *File* $\times$ *API*: a source file or a communication post with code snippets containing an API function from a third-party library;
- *develops* $\subseteq$ *Developer* $\times$ *Class*: a developer contributes to the development of a class in a software project;

- *extends* $\subseteq$ *Class* $\times$ *Class*: a class inherits an abstract class. Two classes extend a same abstract class have a bond since they share a certain number of common functionalities;
- *implements* $\subseteq$ *File* $\times$ *File*: it represents a specific relation that can occur between the source code given in two different files, e.g., a class specified in one file implementing an interface given in another file;
- *hasSourceCode* $\subseteq$ *Project* $\times$ *File*: an OSS project contains a source file;
- *isUsedBy* $\subseteq$ *Library* $\times$ *Project*: a project includes a third-party library to make use of the library's functionalities;
- *invokes* $\subseteq$ *Class* $\times$ *API*: this is the case when a class calls an API function from a third-party library. API calls can be extracted from source files using suitable code parsers;
- *stars* $\subseteq$ *User* $\times$ *Project*: it represents projects that a given user has starred. This relationship is only applicable to GitHub.

Figure 3.2 depicts an example of the graph representation for various OSS artifacts. There are several semantic edges to describe the mutual relationship between graph nodes. For example, the edge *includes* describes the relationship between a project and a third-party library, whereas the edge *hasSourceCode* dictates that a project contains a source code file. The graph structure facilitates similarity computation [22]. For instance, several existing algorithms are able to compute the similarity between *project#1* and *project#2* as they are indirectly connected by the pair of edges, i.e., *hasSourceCode* and *implements* [92, 94]. This semantic path reflects the actual relevance of the projects, they contain classes that implement a common interface. The similarity is further enforced by another path via *hasSourceCode* and *invokes*, leading to two API functions, i.e., *API#1* and *API#2*. The two projects are a bit more similar since they invoke same APIs. Analogously, the similarity between *project#1* and *project#3* can also be inferred since they both include two third-party libraries *lib#1* and *lib#2*, and projects share similar libraries are considered to be similar [26].

The graph structure allows for similarity computation on different artifacts. For instance, it is possible to compute similarities among developers: we see that developers *dev#1* and *dev#2* share a common activity, they develop two classes, i.e., *HttpSocket.java* and *FtpSocket.java* and these classes are somehow similar. In particular, both *HttpSocket.java* and *FtpSocket.java* implement *Socket.java*, furthermore, they all invoke *API#1* and *API#2* in their code.

To understand how to incorporate existing APIs into current code, a developer normally looks for API documentations that describe the constituent functions. For instance, Stack Overflow provides the developer with a broader insight of API usage, and in some cases, with sound code examples [36, 80]. In Figure 3.2, there is a Stack Overflow post, i.e., *Post#1* contains code snippets with two function calls *API#2* and *API#3*. In practice, this is a typical scenario when users discuss the usage of libraries containing *API#2* and *API#3*. In this respect, it might be helpful if we recommend *Post#1* to the developer of class *FtpSocket.java*. This is completely feasible since the graph structure allows one to compute the similarity between *Post#1* and *FtpSocket.java*. In the end, a recommendation engine can provide the developer with a list of Stack Overflow posts that are relevant to the code being developed. The graph structure allows for the integration of different relationships as depicted in Figure 3.2. CrossSim has been designed as a general framework to compute software similarity by exploiting various features, both low-level and high-level. In the scope of this work, we concentrate on studying the performance of CrossSim with respect to high-level information: *isUsedBy*, *develops*, and *stars* as discussed in Section 3.3. In a recent work [33, 12], we exploited CrossSim to compute similarities by considering low-level information, i.e., API function calls as features.

3.2.2 SimRank: computing similarity among graph nodes

Graph similarity is an active field and receives a significant attention from the research community. Computing similarity among graph nodes has been applied to solve different problems in Computer Science. For instance, graph algorithms have been used to measure the similarities among social network nodes [103, 104], proteins [105], RDF graph nodes [95, 23], to name a few.

In this work, we apply graph similarity to solve the problem of computing the similarities among various OSS projects. First, we recall some notations as follows. A directed graph is defined as a tuple $G = (V, E, R)$, where V is the set of vertices, E is the set of edges, and R represents the relationship among the nodes. A graph consists of nodes and oriented links with semantic relationships [99]. A triple $<$*subject*, *predicate*, *object*$>$ with (*subject*, *object* $\in$ V) and *predicate* $\in$ E states that node *subject* is connected to node *object* by means of the edge labelled with *predicate*. To evaluate the similarity of two nodes in a graph, their intrinsic characteristics like nodes, links, and their mutual interactions are incorporated into the similarity calculation [95, 23]. Among others, feature-based semantic similarity metrics gauge the similarity between graph nodes as a measure of commonality and distinction of their hallmarks.

By feature-based similarity, objects are represented as a set of common and distinctive features and the similarity between two objects is computed by comparing their features [29]. An object is represented in one of the following forms: binary values, nominal values, ordinal values, and cardinal values. Feature-based semantic similarity metrics first attempt to characterize resources in a graph as sets of feature and then perform similarity calculation on them. To evaluate the similarity of two nodes in a graph, their intrinsic characteristics like neighbour nodes, links, and their mutual interactions are incorporated into the similarity calculation [95, 23]. SimRank has been developed to calculate similarities based on mutual relationships between graph nodes [94]. Considering two nodes, the more similar nodes point to them, the more similar the two nodes are. We take an example in Figure 3.3 to illustrate how SimRank works in practice. There, Node 1 is similar to Node 2 since both are pointed by Node 5. Comparably, Node 3 is similar to Node 4 as they are pointed by Node 6. As a result, the two nodes α and β are highly similar because they are concurrently pointed by other four nodes in the graph, i.e., 1, 2, 3, and 4, considering that 1 and 2 as well as 3 and 4 are pairwise similar. In this sense, the similarity between α and β is computed by using a fixed-point function, taking into consideration the accumulative similarity by their pointing nodes. Given $k \geq 0$ we have $R^{(k)}(\alpha,\beta) = 1$ with $\alpha = \beta$ and $R^{(k)}(\alpha,\beta) = 0$ with $k = 0$ and $\alpha \neq \beta$, SimRank is computed as follows [94]:

$$R^{(k+1)}(\alpha,\beta) = \frac{\Delta}{|I(\alpha)| \cdot |I(\beta)|} \sum_{i=1}^{|I(\alpha)|} \sum_{j=1}^{|I(\beta)|} R^{(k)}(I_i(\alpha), I_j(\beta)) \tag{3.5}$$

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

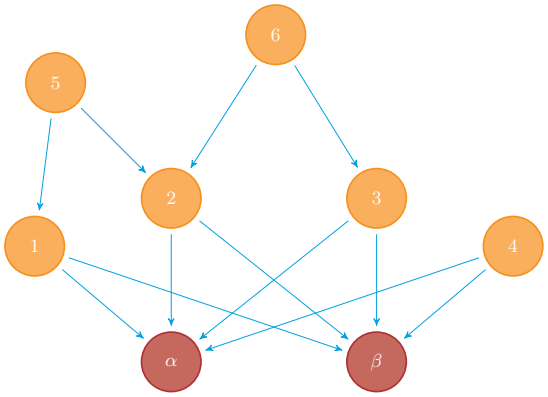


Figure 3.3: An example of how SimRank works.

By using the graph representation as in Section 3.2.1, we are able to transform the OSS ecosystem into a mathematically computable format. This graph structure allows for

the application of various similarity algorithms. In CrossSim we adopt SimRank as the mechanism for computing similarities among OSS graph nodes. However, other similarity algorithms can also be flexibly integrated into CrossSim, as long as they are designed for graph [23]. The utilization of SimRank is convenient and practical also when various relationships are incorporated into the graph. Given the circumstances, the algorithm needs not be changed since it only works on the basis of nodes and edges. In this sense, CrossSim is a versatile similarity tool as it can accept various input features regardless of their format.

To study the performance of CrossSim we conducted a comprehensive evaluation using a dataset collected from GitHub. To aim for an unbiased comparison, we opted for existing evaluation methodologies from other studies of the same type [30, 26, 37]. Together with other metrics typically used for evaluations, i.e., Success rate, Confidence, and Precision, we decided to use also Ranking to measure the sensitivity of the similarity tools to ranking results. The details of our evaluation are given in the next section.

3.3 Evaluation

In this section we describe the process that has been conceived and applied to evaluate the performance of CrossSim compared to some baselines. We opt for MUDABLUE, CLAN, and REPOPAL to compare with CrossSim. The rationale behind the selection of these approaches is that they are well-established algorithms and have demonstrated their effectiveness in various settings. According to *Zhang* et al. [37], by applying the same experiment settings and evaluating on the same dataset, the authors demonstrated that REPOPAL outperforms CLAN in terms of Confidence and Precision. Meanwhile, CLAN has a better performance than that of MUDABLUE, also with respect to *Confidence* and *Precision* [26]. Furthermore, REPOPAL works on GitHub Java repositories containing rich metadata that is suitable for building graph by CrossSim. Intuitively, we consider all these tools as a good starting point for a performance comparison. Furthermore, our evaluation aims at comparing two low-level similarity tools, i.e., MUDABLUE and CLAN with two high-level similarity ones, i.e., REPOPAL and CrossSim.

The evaluation process that has been applied is shown in Figure 3.4 and consists of activities and artifacts that are going to be explained later on this section. In particular, a set of Java projects (see Section 3.3.2) has been crawled to feed as input for the computation by all approaches, i.e., MUDABLUE, CLAN, REPOPAL, and CrossSim. Afterwards, a set of projects is selected as queries to compute similarities against all the remaining OSS projects. Once the scores have been computed, for each similarity tool, some of the top similar projects

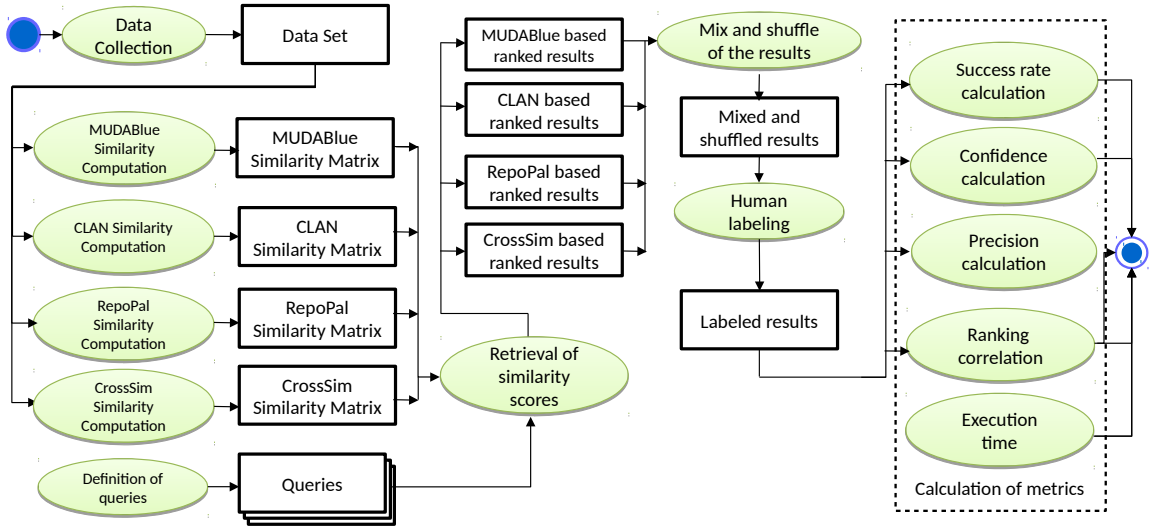


Figure 3.4: The evaluation process.

are chosen, mixed with results by the other tools, and eventually evaluated by humans. The outcomes are then analyzed using various quality metrics.

Since the implementations of the baselines are no longer available for public use, we reimplemented MUDABLUE, CLAN, and REPOPAL by strictly following the descriptions in the original papers [24, 26, 37]. Such implementations, including the CrossSim one, are available online in GitHub [106] to facilitate future research.

3.3.1 Research questions

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

- ***RQ*$_1$*: Which graph configuration brings the best performance to CrossSim?*** Our proposed approach allows for a flexible computation by incorporating different features in a graph. We investigate which types of edges sustain similarity computation by considering different test configurations. In this way, we identify the configuration that fosters the best prediction outcome for CrossSim.

- ***RQ*$_2$*: Which similarity approach between* MUDABLUE*,* CLAN*,* REPOPAL*, and CrossSim yields a better performance?*** By this question, we compare the performance of the approaches concerning the ability to produce accurate recommendations. In the context of software development, providing relevant results is of highly importance since a developer would expect a set of similar projects to the project being developed.

- ***RQ_3: Which similarity approach is more efficient with respect to execution time?*** An important factor for a similarity tool is the ability to compute within an acceptable amount of time. This research question aims at measuring the time needed for a tool to produce a final recommendation.

3.3.2 Dataset

To compare the performance of CrossSim with those of the baselines, it is necessary to execute them on the same set of OSS projects. The collected dataset needs to be suitable as input for all four similarity engines. By MUDABLUE and CLAN, there are no specific requirements since both tools rely solely on source code to function. For REPOPAL and CrossSim, we can consider only projects that satisfy certain criteria. In particular, the collected projects have to meet the following requirements:

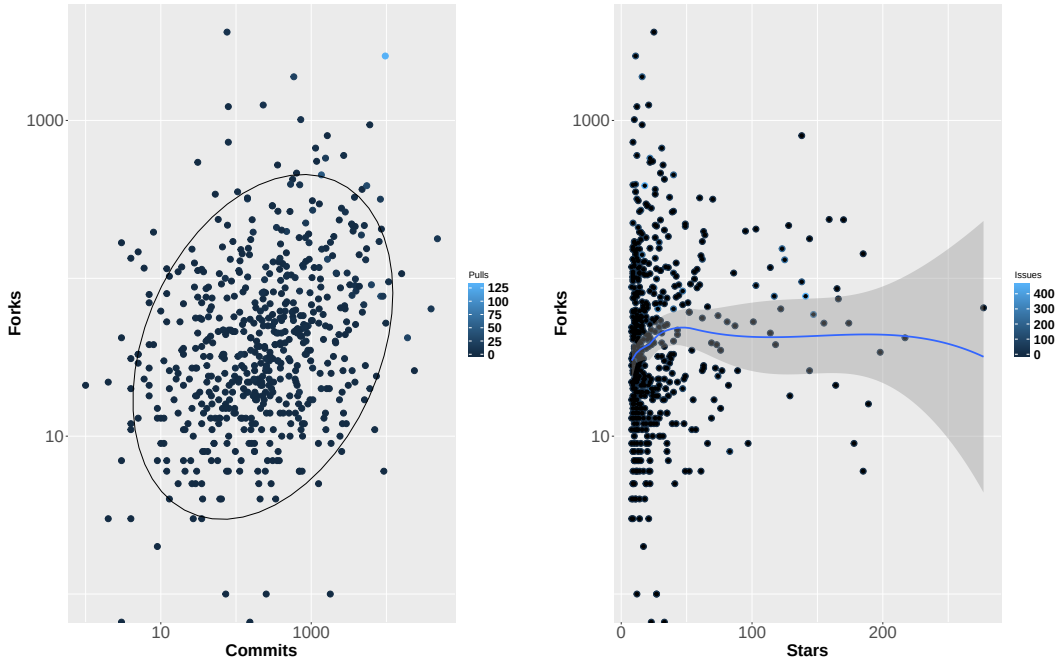
- Providing the specification of their dependencies by means of `code.xml` or `.gradle` files;[4]
- Including at least 9 dependencies – a project with no or little information about dependencies may adversely affect the performance of CrossSim;
- Having the `README.md` file available – this is needed to enable the application of RepoPal;
- Being starred by at least 20 users as required by REPOPAL to work.

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

Using the GitHub API, we crawled projects to provide input for the evaluation. Though the number of projects that fulfill the requirements of a single approach is high, the number of projects that meet the requirements of all approaches is considerably lower. For example, a project contains both `pom.xml` and `README.md`, albeit having only 5 dependencies, does

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

not meet the constraints and must be discarded. The scraping is time consuming as for each project, at least 6 queries must be sent to get the relevant data. In fact, GitHub already sets a rate limit for an ordinary account,[5] with a total number of 5,000 API calls per hour being allowed. And for the search operation, the rate is limited to 30 queries per minute. Due to these reasons, we ended up getting a dataset of 580 projects that are eligible for the evaluation. The dataset we collected is published together with all tools for public usage [106].



(a) Number of forks, commits, and pulls

(b) Number of forks and stars, and issues

Figure 3.5: The projects and the number of forks, commits, pull requests, issues, and stars.

Figure 3.5(a) and Figure 3.5(b) provide a summary on the projects and the number of forks, commits, stars, pull requests, and issues. The number of pull request for most of the projects is considerably low, i.e., lower than 100; however their number of forks and commits is high. Forking is a means of contributing to the original repositories [107]. Furthermore, there is a strong correlation between forks and stars [108], as it is further witnessed in Figure 3.5(b). A project with a high number of forks means can be considered as a sign of a well-maintained and received project. Similarly, as commits have a significant influence on the source code [109], the number of commits is also a good indicator of how a project has been developed.

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

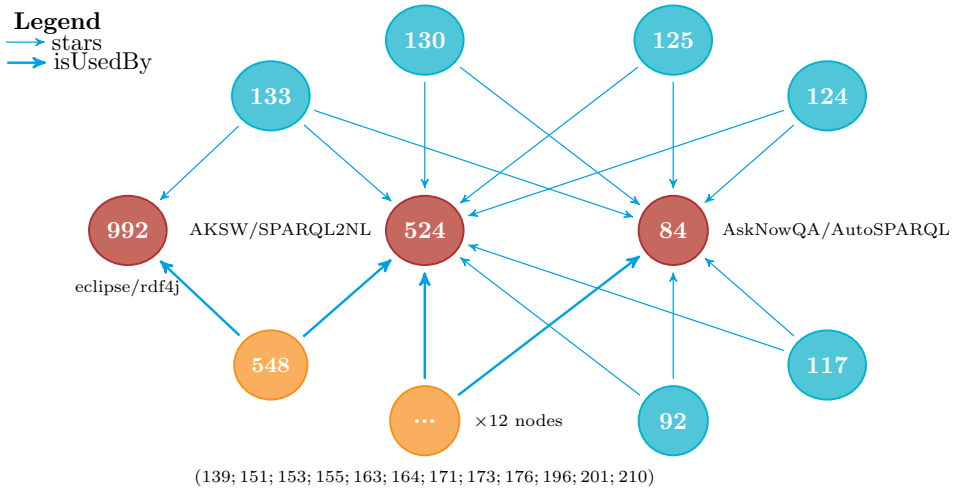


Figure 3.6: Sub-graph showing a fragment of the representation for three projects.

Table 3.1: List of software categories.

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

Further than collecting projects for each category, we also started collecting random projects. These projects serve as a means to test the stability of the algorithms. If the algorithms work well, they will not perceive randomly added projects as similar to projects of some other specific categories. To this end, the categories and their corresponding cardinality to be studied in our evaluation are listed in Table 3.1. This is an approximate classification since a project might belong to more than one category.

As can be seen in Table 3.1, among 580 considered projects, 309 of them belong to some specific categories, such as SPARQL, RDF, Jena Apache, Selenium Test, Elastic Search, Spring MVC, etc. The other 271 projects being selected randomly belong to Miscellaneous Categories. These categories disperse in several domains and sometimes it happens that there is only one project in a category.

3.3.3 Similarity computation

To explain how the graph representation is exploited in CrossSim, Figure 3.6 sketches the sub-graph for representing the relationships between two projects AskNowQA/AutoSPARQL and AKSW/SPARQL2NL. The orange nodes are dependencies and their real names are depicted in Table 3.2. The turquoise nodes are developers who already starred the repositories. Every node is encoded using a unique number across the whole graph. To compute similarity between the two projects, SimRank is applied following Equation 3.5 and the damping factor Δ is empirically set to 0.85 according to some existing studies [23, 94]. The final result ranging from 0 to 1 is the similarity between the two projects AskNowQA/AutoSPARQL and AKSW/SPARQL2NL.

Table 3.2: Shared dependencies in Figure 3.6.

ID	Name
139	org.apache.jena:jena-arq
151	org.dllearner:components-core
153	net.didion.jwnl:jwnl
155	net.sourceforge.owlapi:owlapi-distribution
163	net.sf.jopt-simple:jopt-simple
164	jaws:core
171	com.aliasi:lingpipe
173	org.dllearner:components-ext
176	org.apache.opennlp:opennlp-tools
196	org.apache.solr:solr-solrj
201	org.apache.commons:commons-lang3
210	javax.servlet:servlet-api
548	org.slf4j:log4j-over-slf4j

3.3.3.1 Query definition

Among 580 projects in the dataset, 50 have been selected as queries. We did not sample them randomly but selected those coming from some specific categories. This is due to the fact that the dataset is rather small and if we randomly selected queries which do not belong to any categories, we may end up retrieving irrelevant projects, and this is not useful for the validation process. We assume that the random selection can be done only when more projects available for training. To aim for variety, the queries have been chosen to cover different categories, e.g., SPARQL and RDF, Selenium Test, Elastic Search, Spring MVC, Hadoop, Music Player as listed in Table 3.4.

Table 3.3: Most frequent dependencies in the considered dataset.

Dependency	Frequency
junit:junit	447
org.slf4j:slf4j-api	217
com.google.guava:guava	171
log4j:log4j	156
commons-io:commons-io	151
org.slf4j:slf4j-log4j12	129

Table 3.4: List of queries for evaluation [106].

No.	Project name	No.	Project name
1	neo4j-contrib/sparql-plugin	26	mariamhakobyan/elasticsearch-river-kafka
2	AskNowQA/AutoSPARQL	27	OpenTSDB/opentsdb-elasticsearch
3	AKSW/Sparqlify	28	codelibs/elasticsearch-cluster-runner
4	AKSW/SPARQL2NL	29	opendatasoft/elasticsearch-plugin-geoshape
5	pranab/beymani	30	huangchen007/elasticsearch-rest-command
6	sayems/java.webdriver	31	pitchpoint-solutions/sfs
7	psaravan/JamsMusicPlayer	32	javanna/elasticsearch-river-solr
8	webdriverextensions/webdriverextensions	33	mesos/hadoop
9	dadoonet/spring-elasticsearch	34	pentaho/big-data-plugin
10	seleniumQuery/seleniumQuery	35	asakusafw/asakusafw
11	bonigarcia/webdrivermanager	36	klarna/HiveRunner
12	selenium-cucumber/selenium-cucumber-java	37	sonalgoyal/hiho
13	conductor-framework/conductor	38	pyvandenbussche/sparqles
14	caelum/vraptor	39	lintool/Ivory
15	caelum/vraptor4	40	GoogleCloudPlatform/bigdata-interop
16	KEN-LJQ/WMS	41	Conductor/kangaroo
17	white-cat/jeeweb	42	datasalt/pangool
18	livrospringmvc/lojacasadocodigo	43	laserson/avro2parquet
19	spring-projects/spring-mvc-showcase	44	Knewton/KassandraMRHelper
20	sonian/elasticsearch-jetty	45	blackberry/KaBoom
21	testIT-WebTester/webtester-core	46	jt6211/hadoop-dns-mining
22	elastic/elasticsearch-metrics-reporter-java	47	xebia/Xebium
23	elastic/elasticsearch-support-diagnostics	48	TheAndroidMaster/Pasta-Music
24	SpringDataElasticsearchDevs/spring-data-elasticsearch	49	SubstanceMobile/GEM
25	javanna/elasticshell	50	markzhai/LyricHere

3.3.3.2 Retrieval of similarity scores

Our evaluation has been conducted in line with some other existing studies [30, 26, 37]. In particular, for each query in the set of the 50 projects defined in the previous step, similarity is computed against all the remaining projects in the dataset using the SimRank algorithm discussed in Section 3.2.2. From the retrieved projects, only top 5 are selected

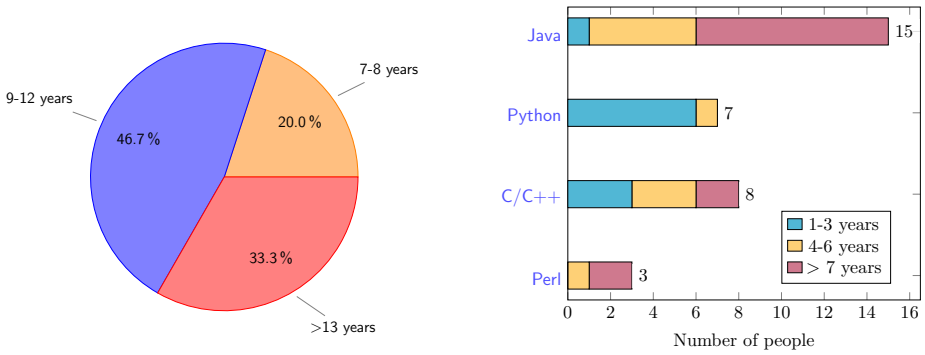
for the subsequent evaluation steps. For every query, similarity is also computed using MUDABLUE, CLAN, and REPOPAL to get the top-5 most similar retrieved projects.

3.3.4 User study

For each similarity tool, the outcomes of the computation are a ranked list of similar projects. It is necessary to evaluate how relevant the projects are, compared to the query project. Since user study is the only way to evaluate the outcome [26, 37], we involved a group of 15 software developers to participate in the manual evaluation. Some of the participants are master students, and most of them work as software developers or researchers in academic and industry. Before the evaluation, we sent each evaluator a tutorial on how to conduct the scoring process. Furthermore, to get information about the participants related to their development background, we sent them a questionnaire similar to the one presented in a relevant work [26]. According to the survey, all the participants are capable of at least two different programming languages, and their favorite code platform is Stack Overflow, where they normally search for posts that are useful for their current development tasks. In addition, most of them tend to re-use code fragments collected from external sources quite often.

Figure 3.7(a) depicts the number of years that the developers have spent for software development activities. All the participants have at least 7 years of programming experience, two of them have more than 20 years. In Figure 3.7(b), we show the number of people and their corresponding number of years of programming experience for different languages. Among others, Java is the programming language that all developers are knowledgeable about, with at least two years of experience. Nine of the developers have spent more than 7 years working with Java. This is highly advantageous for our user study since all projects included in the dataset introduced in Section 3.3.2 are written in Java, and we assume that skillful developers shall have a better judgment about the similarities among projects. The knowledge of different programming languages, i.e., Perl, Python, C/C++, is also a plus for the evaluation process.

By the user study, in order to have a fair evaluation, for each query we mixed and shuffled the top-5 results generated from the computation by each similarity metric in a single Google form and present them to the evaluators who then inspect and give a score to every pair. This mimics a *taste test* where users are asked to evaluate a product, e.g., food or drink, without having a priori knowledge about what is being addressed [110, 111]. This aims at eliminating any bias or prejudice against a specific similarity metric. In particular, given a query, a manual labeling process is performed to evaluate the similarity between the query and the corresponding retrieved projects. The participants are asked to label the similarity



(a) People and years of experience (b) Languages and years of experience

Figure 3.7: A summary of the participants' software development experience.

for each pair of projects (i.e., <*query, retrieved project*>) with regards to their application domains and functionalities using the scales listed in Table 3.5 [26].

Table 3.5: Similarity scales.

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

For example, an OSS project p_1 that performs the sending of files across a TCP/IP network is somehow similar to an OSS project p_2 that exchanges text messages between two users, i.e., $Score(p_1, p_2) = 3$. However, an OSS project p_3 with the functionalities of a pure text editor is dissimilar to both p_1 and p_2, i.e., $Score(p_1, p_2) = Score(p_1, p_3) = 1$. Given a query, a retrieved project is considered as a *false positive* if its similarity to the query is labeled as *Dissimilar* (1) or *Neutral* (2). In contrast, *true positives* are those retrieved projects that have a score of 3 or 4, i.e., *Similar* or *Highly similar*. A good similarity approach should produce as much true positives as possible.

3.3.5 Evaluation Metrics

To evaluate the outcomes of the algorithms with respect to the user study, the following metrics have been considered as typically done in related work [30, 26, 37]:

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

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

- *Ranking*: The obtained human evaluation has been analyzed to check the correlations among the ranking calculated by the similarity tools and the scores given by the human evaluation. To this end the Spearman's rank correlation coefficient r_s [112] is used to measure how well a similarity metric ranks the retrieved projects given a query. Considering two ranked variables $r_1 = (\rho_1, \rho_2, .., \rho_n)$ and $r_2 = (\sigma_1, \sigma_2, .., \sigma_n)$, r_s is defined as: $r_s = 1 - \frac{6\sum_{i=1}^{n}(\rho_i - \sigma_i)^2}{n(n^2-1)}$. Because of the large number of ties, we also used *Kendall's tau* [113] coefficient, which is used to measure the ordinal association between two considered quantities. Both r_s and τ range from -1 (perfect negative correlation) to +1 (perfect positive correlation); $r_s = 0$ or $\tau = 0$ implies that the two variables are not correlated.

Finally, we consider also the *execution time* related to the application of the four approaches on the dataset to obtain the corresponding similarity matrices.

3.4 Results

In Section 3.4.1, the data that has been obtained as discussed in the previous section is analyzed to answer the research questions, i.e., RQ_1, RQ_2, and RQ_3. Afterwards, Section 3.4.2 presents discussions related to the experimental outcomes. Finally, threats to the validity of the evaluation are also discussed in Section 3.4.3.

3.4.1 Data analysis

RQ$_1$: Which graph configuration brings the best performance to CrossSim?

We investigate the implication of graph structure on the outcome of CrossSim by considering various types of edges. This aims at identifying the set of features that contribute to performance gain. First, only *star events* are used to build the graph and this configuration is called CROSSSIM$_1$. Correspondingly, in CROSSSIM$_2$ the graph is built by using only *dependencies*. By CROSSSIM$_3$ we consider both *star events* and *dependencies* together to compute similarity. Afterwards, we extend CROSSSIM$_3$ by incorporating also *Committers*, and this yields configuration CROSSSIM$_4$. Similarly, with CROSSSIM$_5$ we investigate the effect of *frequent dependencies* by adding them to CROSSSIM$_3$. Finally, we take all the above mentioned edges into account and this results in CROSSSIM$_6$. Table 3.6 gives a detailed description of the test configurations used to internally compare CrossSim.

Since the manual evaluation is a time consuming process, we decided to exploit only a subset of the queries to address this research question. In particular, we selected the first 20 queries in Table 3.4, i.e., from number 1 to 20 and provided as input for CrossSim and the results are depicted in Table 3.7.

Table 3.6: CrossSim test configurations.

Configuration	Star events	Dependencies	Committers	Frequent Deps
CROSSSIM$_1$	✓	✗	✗	✗
CROSSSIM$_2$	✗	✓	✗	✗
CROSSSIM$_3$	✓	✓	✗	✗
CROSSSIM$_4$	✓	✓	✓	✗
CROSSSIM$_5$	✓	✓	✗	✓
CROSSSIM$_6$	✓	✓	✓	✓

Among the configurations, CROSSSIM$_2$ gains the lowest prediction performance. In particular, it gets 0.90 as success rate and 0.51 as precision. This implies that using only dependencies as features does not contribute to a good performance. Compared to CROSSSIM$_2$, CROSSSIM$_1$ has a slightly better performance as its success rate and precision are 0.95 and 0.61, respectively. By referring back to the REPOPAL approach (see Section 3.1.3), where information related to the star event such as stargazers and star time gap is used to compute similarity, we come to the conclusion that stars are useful for the detection of similar GitHub repositories. However, we assume that more investigations are needed to understand better the effect of stars on similarity computation. This issue remains as a future work.

We consider CROSSSIM$_5$ in combination with CROSSSIM$_6$ to observe the effect of the adoption of committers. According to Table 3.7, CROSSSIM$_5$ gains a success rate of 100%, with a precision of 0.75. The number of false positives by CROSSSIM$_6$ goes up, thereby worsening the overall performance considerably with 0.70 being as the precision. The

performance degradation is further witnessed by considering CROSSSIM$_3$ and CROSSSIM$_4$ together. Both get 100% as success rate, however CROSSSIM$_3$ obtains a better precision, i.e., 0.80 compared to 0.76 by CROSSSIM$_4$. We come to the conclusion that the inclusion of all developers who have committed updates at least once to a project in the graph is counterproductive as it adds a decline in precision. In this sense, we make an assumption that the deployment of a weighting scheme for developers may help counteract the degradation in performance.

Table 3.7: Comparison of different CrossSim configurations.

	CROSSSIM$_1$	CROSSSIM$_2$	CROSSSIM$_3$	CROSSSIM$_4$	CROSSSIM$_5$	CROSSSIM$_6$
Success rate (%)	**95**	**90**	**100**	**100**	**100**	**100**
Precision	0.62	0.51	**0.80**	0.76	0.75	0.70

Next, CROSSSIM$_3$ and CROSSSIM$_5$ are considered together to analyze the effect of the removal of the most frequent dependencies. CROSSSIM$_3$ outperforms CROSSSIM$_5$ as it gains a precision of 0.80, the highest value among all, compared to 0.75 by CROSSSIM$_5$. The removal of the most frequent dependencies helps also improve the performance of CROSSSIM$_4$ in comparison to CROSSSIM$_6$. Together, this implies that the elimination of too popular dependencies in the original graph is a profitable amendment. This is understandable once we get a deeper insight into the design of SimRank presented in Section 3.2.2. There, two projects are deemed to be similar if they share a same dependency, or in other words their corresponding nodes in the graph are pointed by a common node. However, with frequent dependencies as in Table 3.3 this characteristic may not hold anymore. For example, two projects are pointed by junit:junit because they use JUnit[6] for testing. Since testing is a common functionality of many software projects, it does not help contribute towards the characterization of a project and thus, needs to be removed from the considered graph.

Answer to RQ$_1$. Among the considered experimental configurations, using star events and dependencies to build the graph brings the best prediction performance. The graph structure considerably affects the outcome of the similarity computation. In this sense, finding a graph structure that facilitates similarity computation is of paramount importance.

RQ$_2$: Which similarity approach between* MUDABLUE, CLAN, REPOPAL, *and CrossSim yields a better performance? For this research question, we compared the best configuration CROSSSIM$_3$ with the baselines. The experimental results are shown in Table 3.8 and Figure 3.8. In particular, Table 3.8 depicts Success rate, Precision, Execution time, Spearman's (r_s), and Kendall's tau (τ) for all similarity tools. The obtained results demonstrate that REPOPAL is a good choice for computing similarity among OSS projects. Its success

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

rate and precision are superior to those of MUDABLUE and CLAN. Both MUDABLUE and CLAN obtain a success rate of 60%, however REPOPAL gets a success rate of 100%. Furthermore, the precision of MUDABLUE and CLAN is 0.22 which is considerably lower than 0.71, the corresponding value for REPOPAL.

Table 3.8: Comparison of the similarity approaches.

	MUDABLUE	CLAN	REPOPAL	CROSSSIM$_3$
Success rate (%)	60	60	**100**	**100**
Precision	0.22	0.22	0.71	**0.78**
Execution time (min)	380	22	240	**12**
Spearman's (r_s)	-0.01	-0.08	-0.132	**-0.230**
Kendall's tau (τ)	-0.01	-0.07	-0.108	**-0.214**

In comparison with the other tools, CROSSSIM$_3$ has a better performance concerning Success rate and Precision. Both REPOPAL and CROSSSIM$_3$ gain a Success rate of 100%. However, CROSSSIM$_3$ gets 0.78 as precision which is higher than 0.71, the corresponding value by REPOPAL. In this sense, CrossSim outperforms the baselines with respect to success rate and precision.

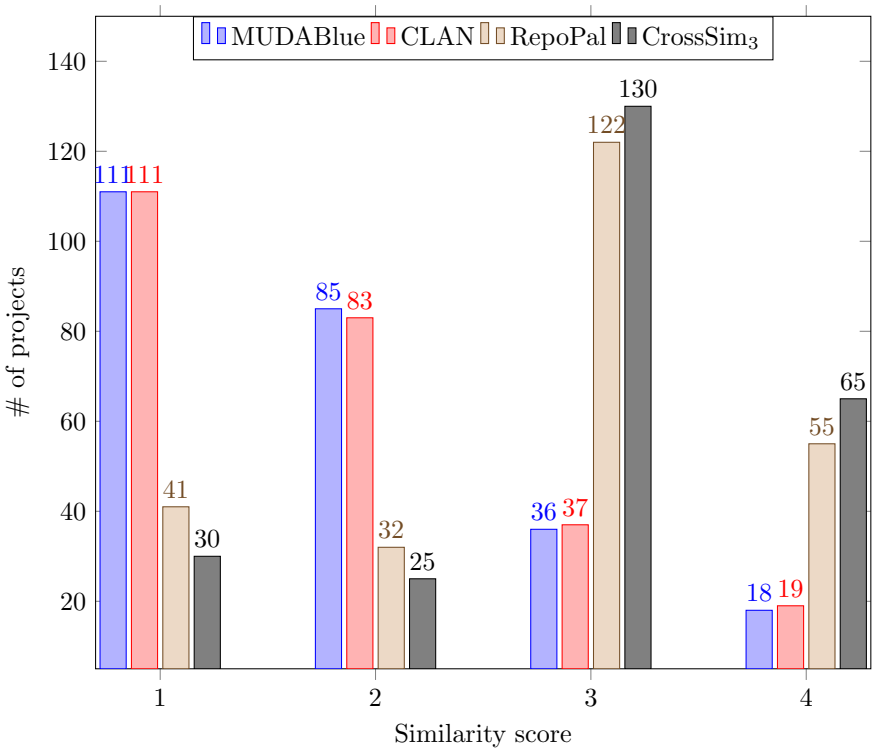


Figure 3.8: Confidence.

The Confidence for the similarity tools is shown in Figure 3.8. MUDABLUE has more false negatives than CLAN does, i.e., pairs that are scored with 1 or 2. In particular, the

number of project pairs that have been assigned the score of 1 is 111 for both MUDABLUE and CLAN. Meanwhile, the corresponding number of pairs that have the score of 2 is 85 and 83 for MUDABLUE and CLAN, respectively. In addition, CLAN has more true positives, i.e., scores of either 3 or 4. In this sense, we conclude that CLAN has a slightly better performance in comparison to MUDABLUE.

By this index, i.e., Confidence, CROSSSIM$_3$ yields a better outcome as it has more scores that are either 3 or 4 and less scores that are 1 or 2. For the similarity score of 3, CROSSSIM$_3$ finds 130 relevant projects, whereas REPOPAL returns 122 projects. The same trend can also be witnessed for the similarity score of 4: the number of projects that REPOPAL and CROSSSIM$_3$ retrieve is 55 and 65, respectively. It is evident that CROSSSIM$_3$ achieves a better Confidence compared to that of REPOPAL.

In addition to the conventional quality indexes, we investigated the ranking produced by the metrics using the Spearman's (r_s) and Kendall's tau (τ) correlation indexes. The aim is to see how good is the correlation between the rank generated by each metric and the scores given by the developers, which are already sorted in descending order. In this way, a lower r_s (τ) means a better ranking. Both indexes r_s and τ are computed for all 50 queries and related first five results. The value of r_s is -0.230 for CROSSSIM$_3$ and -0.132 for REPOPAL. The value of τ is -0.214 for CROSSSIM$_3$ and -0.108 for REPOPAL. By this quality index, CROSSSIM$_3$ performs slightly better than REPOPAL. Compared to MUDABLUE, CLAN obtains a superior ranking with respect to both r_s and τ. In particular, CLAN obtains -0.08 and -0.07 for r_s and τ, respectively; and MUDABLUE gets -0.01 for both r_s and τ.

The experimental results confirm the claim made by the authors of REPOPAL [37]: The system obtains a better recommendation performance than CLAN in terms of Success rate, Precision and Confidence. Furthermore, the ranking by REPOPAL is also better to those of MUDABLUE and CLAN.

Answer to RQ$_2$. Compared to MUDABLUE, CLAN, and REPOPAL, CrossSim gains a better performance in terms of Success rate, Precision, Confidence, and Ranking. The obtained results confirm our hypothesis that the incorporation of various features, e.g., dependencies and star events into graph facilitates similarity computation.

RQ$_3$: Which similarity approach is more efficient with respect to execution time? We measure the time needed to produce recommendations for all the four similarity approaches. The execution time related to the application of the analyzed techniques is shown in the third row of Table 3.8. For the experiments, a laptop with Intel Core i5-7200U CPU @ 2.50GHz $\times$ 4, 8GB RAM, Ubuntu 16.04 was used. In such a configuration, REPOPAL takes $\approx$4 hours to generate the similarity matrix, whereas the execution of CROSSSIM$_3$, including both the time for generating the input graph and that for generating the similarity matrix,

takes ≈12 minutes. Such an important time difference is due to the time needed to calculate the similarity between `README.md` files, on which REPOPAL relies. MUDABLUE takes 380 minutes to complete the computation, even though most of the time (310 minutes) is devoted to the creation of the internal matrices needed to perform the similarity calculation. Meanwhile, CLAN needs only 22 minutes to do the complete computation. Also in this case the creation of the internal structures is the most computational demanding phase, it takes 21 minutes. Such execution times make evident the distinction between high-level and low-level similarity approaches. In particular, MUDABLUE takes into consideration all source code artifacts for computation, such as variable names, method names, the matrix used to represent the input may become huge. CLAN considers only API calls for computation. This is the reason why the computation times for MUDABLUE and CLAN are much higher than those of REPOPAL and CrossSim that instead consider just project metadata.

Answer to RQ$_3$. Among the considered systems, CrossSim is the most efficient one as it generates the similarity matrix in the shortest time.

3.4.2 Discussion

To aim for a reliable evaluation, the experiments showed in this chapter have been performed in line with existing studies [26, 37]. As can be seen, CrossSim has a better performance than that of MUDABLUE, CLAN, and REPOPAL with respect to different quality indicators. The gain in CrossSim's performance demonstrates that the consideration of different artifacts, e.g., projects, libraries in a mutual relationship, instead of individual items brings a substantial benefit. Referring back to the REPOPAL paper [37], we see that the success rate of REPOPAL in our evaluation is considerably higher than that in the original experiments. This can be explained as follows: According to our investigation on the dataset considered by REPOPAL,[7] the chosen projects scatter in several categories and the number of projects belonging to certain categories is rather low. That means, the similarities among the projects are low by their origin. However, in our dataset, projects have been deliberately selected so as to converge on some specific categories, thus increasing their mutual similarities. This makes the possibility that a query gets a relevant project which comes from the same category become superior to that by the original REPOPAL dataset. As a result, the success rate in our experiments increases considerably.

In our evaluation, compared to MUDABLUE, CLAN gains a better confidence: it returns fewer false negatives and more true positives. This partly confirms the findings by McMillan *et al.* in their work [26]. However, both MUDABLUE and CLAN obtain comparable success rate and precision. This is not completely consistent with their claim since they

[7] https://github.com/yunzhang28/RepoPal/blob/master/1000repo.xlsx

demonstrated that CLAN gained a better performance compared to that of MUDABLUE. Our intuition is that the discrepancy between our work and that of McMillan *et al.* [26] may attribute to following reasons: *(i)* although we attempted to strictly follow descriptions by the related papers to implement the tools, the final implementations might not be exactly identical to the original ones. Particularly, the selection of various parameters for the LSA implementation used in MUDABLUE and CLAN may considerably contribute towards the difference; *(ii)* the dataset in our evaluation is different from the one used for evaluating CLAN. Altogether, this should introduce some fluctuations in the performance of both approaches.

Currently, CrossSim supports the incorporation of *isUsedBy*, *develops*, and *stars* to compute similarities [106], and thus being a high-level similarity metric. However, it is feasible to consider also low-level features, such as package names, class names, or API function calls (as partly shown in Figure 3.1). In this way, CrossSim becomes a *hybrid* similarity metric as it deals with both metadata (high-level) and source code (low-level). Furthermore, since CrossSim allows for the integration of various graph algorithms for calculating similarity, the deployment of different techniques rather than SimRank might possibly improve the overall recommendation performance, depending on the set of features. To this end, the selection of a proper similarity technique for each type of graphs can be considered as an open research topic.

We assume that the graph structure may have a dramatic influence on its performance. Thus, finding a suitable graph that facilitates the computation is an interesting research topic and needs further investigations. Moreover, since very frequent nodes are not useful for similarity computation, it is necessary to define a threshold at which a node is considered to be frequent.

3.4.3 Threats to Validity

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

Internal validity concerns any confounding factor that could influence our results. We attempted to avoid any bias in the evaluation and assessment phases: (*i*) by involving 15 developers with decent programming experience in the user study. In particular, the labeling results by one participant were then double-checked by another one to aim for soundness of the outcomes; (*ii*) by completely automating the evaluation of the defined metrics without any manual intervention. Indeed, the implemented tools could be defective. To contrast and mitigate this threat, we strictly followed the descriptions in the original

papers to re-implement the tools. Furthermore, we have run several manual assessments and counter-checks to validate the evaluation outcomes. Another possible internal threat is related to the similarity scales used in the evaluation (Table 3.5), which were adopted from an existing work [26]. The 4-point scale includes one *"negative"*, one *"neutral"* and two *"positive"* outcomes, thus appearing to be unbalanced. However, in the evaluation, *"neutral"* has been considered as a negative score, and thus we still have a balance for the scale. More importantly, we use the same scores to compare the considered approaches, as a result there exists no bias against any specific approach.

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

Construct validity 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 similar GitHub repositories. In this way, we were able to investigate if the tools are really applicable to authentic usage.

Conclusion validity 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, Ranking, and Execution time might cause a threat to conclusion validity. To confront the issue, we employed the same metrics for evaluating all the similarity approaches.

3.5 Conclusion

In this chapter we presented CrossSim, a framework for computing similarities among OSS projects. Through a review on some of the most notable methods for detecting similarity in software applications and open source projects, we came to the conclusion that a representation model that flexibly incorporates various features and semantic relationships is highly beneficial to similarity computation in the context of an OSS ecosystem. We considered the community of developers together with OSS projects, libraries as well as various artifacts and their mutual interactions as whole by using the graph representation. In the graph, either humans or non-human factors have mutual dependency and implication on the others. By means of a graph, we are able to transform the relationships among various artifacts, e.g.,

developers, API utilizations, source code, interactions, into a mathematically computable format. The implementation of CrossSim exploits SimRank to compute similarity in graphs and it can handle the following relationships: *isUsedBy*, *develops*, and *stars*. CrossSim is a versatile similarity tool as it can accept various input features regardless of their format.

We conducted an evaluation of computing similarities among OSS projects on a dataset of 580 GitHub Java projects. Using MUDABLUE, CLAN, REPOPAL as baselines, we studied the performance of our approach. The obtained results are promising, among the test configurations, CROSSSIM$_3$ has the best performance, where dependencies and star events are considered as features for the similarity computation. It is our belief that CrossSim is a good candidate for computing similarities among OSS projects. In order to enable the reproducibility of the performed experiments, we made available the source code implementation of MUDABlue, CLAN, REPOPAL, and CrossSim as also the dataset exploited and the corresponding user study in our GitHub repository [106]. For future work, we are going to investigate in more detail the influence of graph structure on similarity computation. In addition, we plan to incorporate also low-level similarity features such as API function calls, package names, etc., into the graph to see if the recommendation performance can be improved. Last but not least, we are going to exploit CrossSim to automatically cluster OSS projects.

Chapter 4

Endowing recommender systems with user feedback

In this chapter, we present a novel approach to exploiting user feedback to enhance TPLs recommendations. We address the motivating question: "*Is it possible to increase the relevance of the items provided by a TPL recommender system by augmenting it with user feedback?*" taking into consideration three types of feedback, i.e., positive, negative, and additive. To this end, we built an initial prototype based on the Learning to Rank [114] (LTR) model to rearrange the recommended list of items produced by CROSSREC [17], a well-founded TPLs recommender system, according to the given user feedback. Furthermore, we also propose a method to enable additive feedback, i.e., a user can endorse a library that was not in the original recommendations. The preliminary study conducted on CROSSREC shows that the approach obtains encouraging results in equipping the system with the proposed feedback mechanism. Moreover, our findings suggest that the envisioned technique can be incorporated into different kinds of recommender systems.

Structure of the chapter. In Section 4.1 we motivate our work and give an overview of the underlying technologies. The approach and its architecture are described in Section 4.2. Section 4.3 presents the results obtained from the conducted experiments, and Section 4.4 concludes the chapter.

4.1 Motivation and background

4.1.1 Explanatory example

To highlight our contribution, we take a motivating example as shown in Figure 4.1. Given a ranked list of TPLs provided by a recommender system and that is supposed to be relevant

for the current development context, a user can express three different types of feedback, namely *positive*, *negative* or *additive*. For instance, the system provides as a first item *junit* by relying on its internal mechanism. However, the user may prefer another one, for instance, *mockito*, according to the requested functionalities. To this end, a possible mechanism to *upvote* or *downvote* a library is represented in Figure 4.1, where *mockito* has been preferred over *junit* that is, instead, marked with a "dislike." Furthermore, the user can suggest a new item that does not belong to the original recommended list. In the example, the user wants to add the *jackson* library since it provides similar functionalities of *json* that was already suggested. Such an envisioned feedback mechanism aims to increase the performance of the used recommender systems, e.g., in terms of the relevance of recommended items to the current development context.

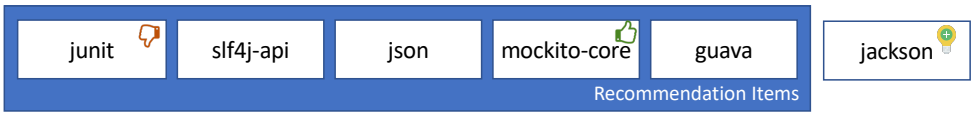


Figure 4.1: Explanatory example.

4.1.2 Background

Third-party library recommendation. TPLs recommender systems provide developers with third-party libraries that are considered to be relevant to the projects under development [49, 17, 65, 51]. This section recalls CROSSREC [17], as it is considered as among state-of-the-art library recommenders. CROSSREC works based on the assumption that *"if projects share some third-party libraries, then they will probably share additional libraries."* In particular, the system encodes the relationships among OSS projects in a graph and utilizes a collaborative-filtering technique [8] to retrieve TPLs. CROSSREC returns a ranked list of libraries collaboratively mined from the most similar projects given an input project.

To the best of our knowledge, though several TPLs recommender systems exist, no work has been done to enable them to exploit user feedback to increase the relevance of the recommended items. As detailed in the next section, this chapter proposes a novel approach to equip recommender systems with the management of user feedback.

Learning to Rank (LTR). It is a supervised learning technique widely used to cope with the ranking task [115]. Given a query q and a set of documents $D(d_1,..,d_n)$, LTR ranks them according to their relevance with respect to the query. To this end, the feature vectors are extracted from the initial dataset and are used to feed the model using the stochastic gradient descent method. It eventually retrieves a ranked list of documents and the corresponding relevance score.

We utilize a particular LTR model, namely the Weighted Approximate-Rank Pairwise (WARP) model [116]. The rationale behind this choice is that it works better in sorting top-K elements of a recommended list. In particular, the WARP model is employed to sort the list of TPLs as recommended by CROSSREC to align it with user preferences that have been previously expressed in terms of feedback.

4.2 The proposed approach

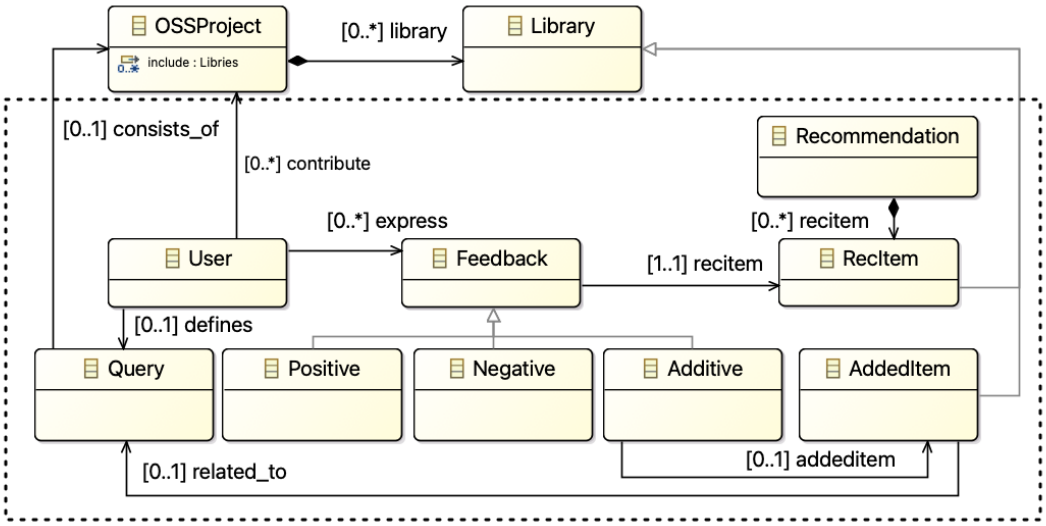


Figure 4.2: Conceptual model of the user feedback mechanism for TPLs recommendations.

We conceptualize an approach to endow recommender systems with the management of user feedback. Even though the final goal is to define such a mechanism generically, in this work, we focus on the problem of recommending TPLs and supporting user feedback for them. Figure 4.2 represents a conceptual model covering the concepts of interest. In particular, we focus on open-source projects that depend on a set of TPLs (see the concepts *OSSProject* and *Library*). A developer that is working on a given software project can ask the available recommender system to provide her with a list of further TPLs that might be added to the project under development (see the concept *User* that *defines* a *Query* consisting of the project under development and gets back a *Recommendation* element consisting of different *Items*). Users might want to express their *Feedback* for each returned item to increase the relevance for future similar requests. Users can like or dislike recommended items (see the

concept *Positive* and *Negative*, respectively) or can even suggest additional items that were not included in the original recommended list (see the concept *Additive* feedback).

It is worth noting that the entities encapsulated in the dashed frame represent the agnostic part of the methodology, meaning that the same concepts can be adapted to any kind of recommendations, e.g., API calls or snippets. However, thoroughly assessing the genericity of the proposed approach is planned as future work. The architecture implementing the conceptual model previously described is shown in Figure 4.3 and consists of the following components:

- *TPLs RecSys:* it is the recommender system, which is able to recommend third-party libraries for the project under development. Given the current development context, the system generates a ranked list of recommended libraries ①.
- *User feedback:* the user can express three different types of explicit feedback for each item in the recommended list, i.e., positive, negative and additive ②. Including or removing a TPL from the project under development is mapped to positive or negative feedback, respectively. Additive feedback consists of injecting endorsed libraries into the training data of *TPLs RecSys*. Thus, during the next iteration of the whole process, the system will take into account the new addition. It is worth mentioning that only one injection may not be sufficient to promote the new library for the next recommendation requests;
- *LTR Ranking:* LTR is applied to the ranked list provided by the adopted recommender system by considering previously stored user feedback ③. We make use of the *LightFM* Python library [117] which implements the WARP model. Using the feedback collected

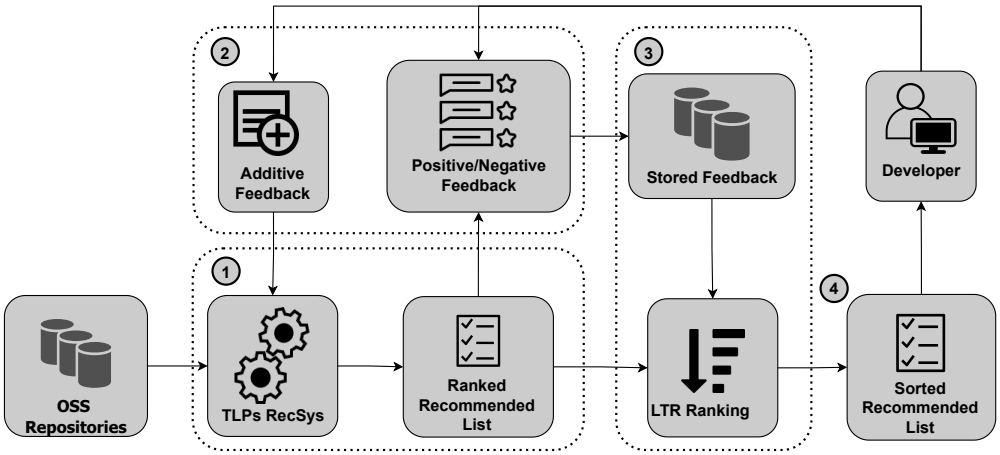


Figure 4.3: System architecture of the user feedback mechanism for TPLs recommendations.

in the previous phase, a set of feature vectors v are extracted to train the LTR model according to the following format: v=$(1,0,0,1,0.....1,0)$, where 1 is a positive rate and 0 is a negative one expressed for each TPL. To feed the ranking model, such vectors need to be transformed into a *scipy* coordinate matrix,[1] i.e., each pair user-library has a rating. Concerning the hyperparameters, we set the learning rate to 0.02 and the number of epochs to 70, given that higher values seem to have negligible effects on the performance. As final output, this component produces the rearranged list of TPLs by considering positive and negative feedback;

- *Sorted Recommended List:* The sorted list of recommended items is eventually presented to the user ④ who decides to either *(i)* accept it or *(ii)* express additional feedback to trigger another recommendation session.

4.3 Results

This section presents the results of the conducted preliminary evaluation to assess the feasibility and the effectiveness of the proposed approach. To this aim we considered CROSSREC [17] as TPL recommender system.

The rationale behind such a selection is that it is considered as among state-of-the-art library recommender systems, e.g., it achieves a better performance compared to three well-founded baselines, namely LibRec [18], LibFinder [49] and LibCUP [50]. To simulate binary feedback, we introduce a rating mechanism that assigns a vote to each library suggested by CROSSREC. Furthermore, we modify the original graph by adding projects and libraries to measure the impact of additive feedback.

4.3.1 Experimental parameters

- $\mathscr{L}$ is the set of libraries on which the user expresses feedback, i.e., positive, negative or additive;

- p and p' are the original position and position after the ranking phase respectively of $l \in \mathscr{L}$;

- R_l is the set of user feedback for a specific library $l \in \mathscr{L}$ expressed as a binary rate r, i.e., 0 and 1 for negative and positive feedback, respectively;

[1] https://bit.ly/30v9sVD

- $REC(l)$ is the recommended list provided by CROSSREC that includes a specific library $l \in \mathscr{L}$;
- c is the cut-off value (the number of recommended libraries);
- N is the number of positive feedback given to library l;
- K is the set of new OSS projects that includes intentionally seeded libraries to simulate additive feedback.

4.3.2 Metrics

To measure the capability of the methodology to upvote/downvote a given library, we use $hit\text{-}rank_l@N$ [43] defined as: $hit\text{-}rank_l@N=\frac{count_{r \in R_l}(|\Delta p|>0)}{|R_l|}$, where r is the feedback expressed on the library $l \in R_l$ by the user and $|\Delta p|$ is the delta between the original position p of the item and its new position p' after rearranging. This metric represents how many times the considered library $l \in L$ should be upvoted/downvoted to modify its starting position p. For positive feedback, we consider a match if the $p' > p$. Similarly, $p' < p$ means that l was successfully downvoted using negative feedback.

We measure the effectiveness of additive feedback by $hit\text{-}count_l@K$ computed as: $hit\text{-}count_l@K=count_{l \in REC(l)}$, where K is the number of projects that include the new library $l \in \mathscr{L}$ added by the user, and $REC(l)$ represents the recommendations of CROSSREC. In other words, $hit\text{-}count_l@K$ measures how many times $REC(l)$ includes the added library l according to the number of additive feedback K.

4.3.3 Dataset

We use the original CROSSREC replication package and dataset made available online.[2] The dataset consists of 1,200 projects with 13,498 libraries. Due to the lack of real user feedback, we mimic explicit feedback by counting the frequencies of the examined libraries, i.e., if a given project p *includes* a certain library l, the value for the pair (p,l) is 1 otherwise it is set to 0. Such an occurrence is mapped to a *positive feedback*, and the rate of a library is its frequency on the whole dataset, i.e., the number of projects that invoke it.

We chose six among the libraries that are representative in terms of popularity, i.e., from the least to the most popular libraries, and counted the occurrence of the recommended libraries. Table 4.1 describes the libraries, their frequency, and percentage of occurrences in the results, considering two CROSSREC cut-off values, i.e., 10, and 20.

[2] https://github.com/crossminer/CrossRec

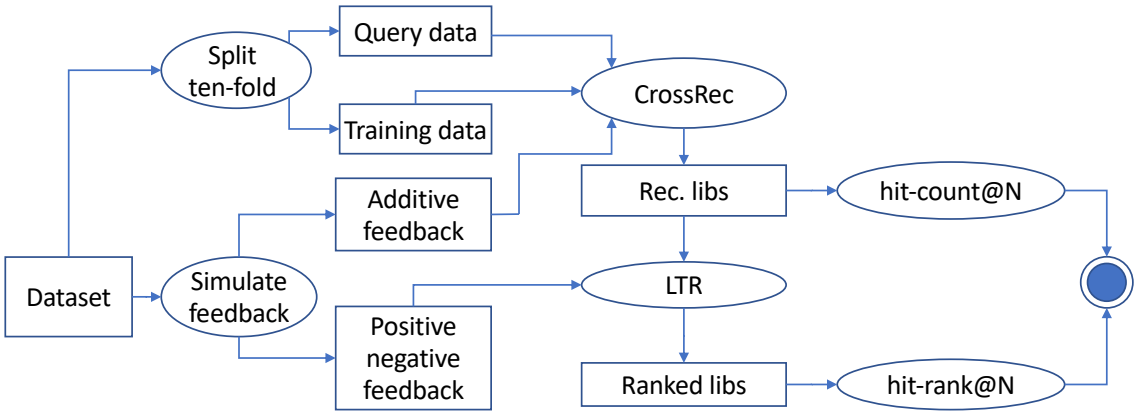


Figure 4.4: The evaluation process.

Table 4.1: Features of the examined TPLs.

	Library					
Value	avro	guice	mockito	guava	slf4j	junit
Freq.	26	74	213	306	473	969
$\mathbf{f}_{c=10}$	0.007	0.010	0.232	0.294	0.415	0.754
$\mathbf{f}_{c=20}$	0.025	0.043	0.359	0.398	0.478	0.755

4.3.4 Methodology

We set up different configurations as follows. Concerning the cut-off values, c=10 and c=20 are chosen to measure the effectiveness of the LTR model when different sizes are considered. We empirically vary the number of positive and negative feedback by varying N, and K, i.e., N=$\{20, 40, 100, 200, 600, 1000\}$, K=$\{0, 20, 50, 100, 200\}$, to assess the contribution of additive feedback. Figure 4.4 depicts the evaluation process consisting of three phases, i.e., data preparation, recommendations, and evaluation. Starting from the dataset (see Section 4.3.3), positive, negative, and additive feedback is simulated according to c, N, and K. We applied the ten-fold cross-validation technique, and the query data is generated from testing projects by removing half of their libraries. To inject an additive feedback for a library l_k with respect to the project context p=$\{l_1, ..., l_n\}$, a new row $\{p, l_1, ..., l_n, l_k\}$ is added to the training data. Given such query data, the system retrieves the list of recommended items. Then, given the first c items, positive and negative feedback, LTR ranks the result provided by CROSSREC.

4.3.5 Results

We analyze the results using two research questions.

Table 4.2: Results obtained with positive feedback.

	Lib.	p	hr_{20}	hr_{40}	hr_{100}	hr_{200}	hr_{600}	hr_{1000}
c=10	avro	7	**0.56**	0.44	0.44	0.56	0.44	0.44
	guice	8	0.46	0.53	**0.62**	0.38	0.46	0.61
	mockito	6	0.49	0.52	0.48	0.54	0.47	**0.55**
	guava	6	0.48	0.48	0.48	**0.52**	0.47	0.50
	slf4j	3	0.64	**0.71**	0.61	0.70	0.64	0.70
	junit	3	0.89	0.90	0.89	**0.92**	0.84	0.90
c=20	avro	13	0.20	0.30	0.20	0.27	**0.33**	0.27
	guice	12	0.37	0.37	**0.41**	0.41	0.35	0.33
	mockito	10	0.44	0.44	0.45	0.42	**0.48**	0.48
	guava	10	0.44	0.45	0.46	**0.48**	0.46	0.46
	slf4j	8	**0.57**	0.48	0.53	0.48	0.44	0.54
	junit	6	0.87	0.79	**0.90**	0.88	0.86	0.88

▷ **RQ**$_1$: *How does the positive and negative feedback contribute to rearranging* CROSSREC*'s recommendations?* We measure the impact of positive and negative feedback in upvoting and downvoting a certain library, through a series of experiments using the configurations described in Section 4.3.1. Table 4.2 shows the results obtained by the proposed methodology when positive feedback is considered.[3] Given a library l, we compute the average initial position p by setting the rating of all users to 0 as described by the p column, i.e., l has not been rated by any user yet. Starting from this initial state, we increase the number of *positive* ratings according to N. In this way, we resemble the situation where an unrated library grows in popularity by exploiting explicit feedback.

The results show that the mechanism is able to promote a library. Such an improvement is more evident for the most popular libraries, i.e., *junit*, *slf4j-api*. In fact, the number of positive rates needed to upvote a library is not the same for all libraries, i.e., the LTR module reduces the popularity impact. For instance, assigning 100 positive rates to *guice* improves its ranking 0.62 of the time with c=10. Meanwhile, *mockito* reaches the maximum value at *hit-rank*$_l$@1000, though it is more popular with the frequency of 213.

While the mechanism works better in upvoting popular libraries, i.e., *hit-rank*$_l$@N reaches around 0.90 of effectiveness for *junit*, the most popular library, it suffers from degradation of performance when a less popular item is considered, e.g., the system improves the ranking for *avro* only 0.56 of the time with $c = 10$. The performance is negatively affected when increasing c from 10 to 20 for almost the libraries, except *junit*. This can be reasoned by referring to Table 4.1. Since *junit* is the most popular item, it appears in the CROSSREC top rank items in almost all the tests.

We conduct a similar experiment to measure the impact of a *negative* feedback to downgrading a popular library. The different configurations and the corresponding results are

[3]For the sake of presentation, hr_N and hc_K stand for *hit-rank*$_l$@N and *hit-count*$_l$@K, respectively.

Table 4.3: Results obtained with negative feedback.

	Lib.	p	hr_{20}	hr_{40}	hr_{100}	hr_{200}	hr_{600}	hr_{1000}
c=10	avro	5	0.11	**0.33**	0.11	0.11	0.22	0.11
	guice	6	**0.15**	0.15	0.15	0.00	0.15	0.00
	mockito	4	**0.25**	0.23	0.22	0.21	0.18	0.24
	guava	5	0.19	0.16	0.19	**0.24**	0.19	0.19
	slf4j	3	0.16	0.15	0.14	**0.17**	0.13	0.16
	junit	2	0.08	0.04	**0.19**	0.04	0.08	0.07
c=20	avro	12	0.27	0.20	0.27	0.13	**0.43**	0.20
	guice	12	**0.22**	0.18	0.18	0.22	0.14	0.18
	mockito	7	**0.29**	0.28	0.28	0.28	0.28	0.27
	guava	8	0.21	0.26	0.21	0.22	0.24	0.27
	slf4j	7	**0.29**	0.24	0.24	0.28	0.23	0.23
	junit	2	0.07	0.07	0.11	0.06	**0.14**	0.05

shown in Table 4.3. In this setting, the p column represents the initial position of the library when its rate is equal to $1,200$, i.e., every user upvotes the library. In such a way, we are able to simulate negative feedback by decreasing the votes using the same threshold defined for the previous experiment.

Downvoting a popular library is more difficult as the corresponding $hit\text{-}rank_l@N$ scores are lower for all libraries, compared to the results in Table 4.2. Generally speaking, the negative feedback successfully downvotes the target library only 0.30 of the attempts. The better results are reached with *avro*, i.e., $hit\text{-}rank_l@40$ =0.33 with c=10 and $hit\text{-}rank_l@600$ =0.46 with c=20. The findings suggest that it is easier to downvote a less popular library than a most used one. This claim is confirmed by the results obtained with *junit* since $hit\text{-}rank_l@1000$ is extremely small for all the configurations, i.e., the maximum value is 0.07. This is expected since users usually follow the *wisdom of the crowd* during their development activities, i.e., they tend to select libraries used by the majority of the community [73].

Answer to RQ$_1$. Either positive or negative feedback has a clear impact on the recommendation results. The effectiveness strongly depends on the popularity of the considered libraries.

▷ **RQ$_2$**: *How does the additive feedback impact on the original recommended list?* We study the influence of additive feedback by simulating the addition of new projects relying on libraries recommended by users. Such additions induce the modification of the CROSSREC's original matrix and allow us to measure the number of additions needed to promote the user suggested library to make it appear in the recommended list. The results obtained for additive feedback are shown in Table 4.4. Column hc_0 contains data that is obtained without operating any change to the CROSSREC's original matrix. Subsequent columns instead, contain values that are obtained after adding fake projects including the library of the corresponding row. For instance, the cells *[avro,*hc_{20}*]* contain how many times out of 1,200 queries, the library

Table 4.4: Results obtained with additive feedback.

	Lib.	hc_0	hc_{20}	hc_{40}	hc_{100}	hc_{200}	hc_{600}	hc_{1000}
c=10	avro	9	86	172	368	511	711	**758**
	guice	15	122	221	377	514	695	**740**
	mockito	322	449	520	640	732	836	**864**
	guava	382	471	541	631	691	759	**777**
	slf4j	536	564	569	609	625	659	**672**
	junit	**995**	986	993	985	985	983	992
c=20	avro	31	146	264	470	615	784	**822**
	guice	55	200	341	492	619	768	**798**
	mockito	491	618	680	782	838	930	**944**
	guava	518	594	668	733	781	822	**829**
	slf4j	616	637	637	669	693	704	**726**
	junit	997	989	**998**	990	989	988	997

avro has been recommended after having added 20 artificial projects containing it. The table shows a dominant trend: the more projects are seeded, the more often the endorsed library gets recommended. While this is more visible with less frequent libraries, e.g., *avro*, it is less evident with popular libraries, such as *junit*, for instance, with c=20, from hc_0 to hc_{1000}, there is only a small fluctuation in the number of recommendations. Altogether, we see that similar to results with positive feedback, the additive ones have a remarkable impact on the less popular libraries, i.e., they increase their probability of being suggested by CROSSREC.

Answer to RQ$_2$. Introducing additive feedback helps increase popularity of the endorsed libraries. The more frequent a library is (e.g., *junit*), the less the additive feedback impacts on the number of times it is recommended.

4.4 Conclusion

The introduction of user feedback in recommender systems has brought considerable benefits in different domains. This work conceived a novel approach to incorporate user feedback into TPLs recommender systems. The proposed methodology supports binary and additive feedback by relying on a well-founded LTR model to rearrange the delivered items. The preliminary evaluation of an existing TPLs recommender system using simulated feedback confirms the approach's feasibility. For future work, we plan to integrate the technique in one of the existing IDEs e.g., Eclipse or VSCode, before conducting a user study to collect real feedback. Moreover, further than improving the adopted LTR model by tuning the hyperparameters, we will investigate the application of other ranking models e.g., vector space and probabilistic ones. Lastly, we will extend the proposed methodology to different TPLs recommender systems.

Chapter 5

PostFinder: A search engine for retrieving Stack Overflow posts

Developing complex software systems requires mastering several languages and technologies [11]. Thus, software developers need to devote effort to continuously 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 productivity impact [118].

Over the last few years, many studies have been conducted to develop methods and tools being able to provide automated assistance to developers. The introduction of recommender systems to the domain of software development has brought substantial benefits. Among others, recommender systems assist the developer in navigating large information spaces and getting instant recommendations that might be helpful to solve the particular development problem at hand [12, 6]. A recommender system in software engineering is defined as *"... a software application that provides information items estimated to be valuable for a software engineering task in a given context"* [11]. In general, developers have to master a vast number of information sources [7], often at a short time. In such a context, the problem is not the lack of information but instead an information overload coming from heterogeneous and rapidly evolving sources. Thus, recommender systems aim at giving developers recommendations, which may consist of different items, including code examples, issue reports, reusable source code, possible third-party components, and documentation.

Stack Overflow (SO) [119] is the most popular question-and-answer website [36], which is a good source of support for developers who seek for probable solutions from the Web [120, 121]. SO discussion posts provide developers with a broader insight into API usage, and in some cases, with sound code examples. Moreover, in a recent development, Stack Overflow has been exploited by a code-to-code search engine to enrich code queries, with the

ultimate aim of getting relevant source code. In particular, FaCoY [62] has been developed to recommend relevant GitHub code snippets to a project being developed. First, the system retrieves related SO posts to get more relevant source code. Afterwards, it exploits the newly obtained source code to expand the query and search from GitHub for more snippets, which are eventually introduced to developers. The module to retrieve posts plays a decisive role: it is a blocking issue in FaCoY's performance, if the module cannot retrieve any relevant posts, the system is unable to generate recommendations.

In this sense, we see the importance of getting related SO posts, given a code snippet as context. 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 [11]. 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 [6]. 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.

In this work, we propose PostFinder, a Stack Overflow posts recommender system, which is based on a two-phase approach to retrieve posts from Stack Overflow by taking various measures on both the data collection and query phases. To improve efficiency, we make use of Apache Lucene [122] to index the textual content and code coming from Stack Overflow. During the first phase, posts are retrieved and augmented with additional data to make them more exposed to queries. Afterwards, 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. Our work is twofold: *(i)* by providing SO posts, PostFinder can be used to replace FaCoY's SO module; more importantly, *(ii)* PostFinder can work as a standalone tool: given a snippet as context, the tool can provide the developer with highly relevant posts. 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, our work makes the following contributions:

- Identification of augmentation measures to automatically refine the considered input SO dump by considering various pieces of information;
- Characterizing the context code by automatically boosting the constituent terms to improve their exposure to the indexed data, and eventually build a proper query transparently for the developer;
- Two empirical evaluations of the proposed approach to evaluate the performance of PostFinder and to compare it with FaCoY;

- An implementation of the tool, which was successfully integrated into the Eclipse IDE, has been released together with the corresponding metadata to facilitate future research [123].

Structure of the chapter. Section 5.1 provides background and describes the motivations for our work. In Section 5.2 we describe PostFinder, an approach to automatically capture the context under development and recommend relevant post from Stack Overflow. The evaluation procedure is described in Section 5.3, whereas the results are discussed in Section 5.4. Finally, Section 5.5 concludes the chapter.

5.1 Background

Over the last decade, several approaches have been conceived to leverage the use of crowdsourcing in software engineering [124]. Those exploiting Stack Overflow as the main source of information [6, 52–54, 62], can be classified into two main categories:

C1. approaches that focus on the automated creation of queries to be executed by search engines, and on the visualization of the retrieved posts according to some ranking model [6, 53, 54];

C2. approaches that deal with query creation and advanced indexing mechanisms specifically conceived for storing and retrieving SO posts [52, 62].

PROMPTER [6] 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 also considering the entropy 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 Stack Overflow. 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 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 in relation to several baselines [58, 59]

In the scope of this work, we pay our attention to the FaCoY's module for searching SO posts. By a careful observation on the system, we found out that it suffers a setback for incomplete data as well as a brief input query. As we can see later in the 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 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 5.1. The code declares a `CamelContext` variable, and invokes functions `addRoutes()` and `configure()`. This illustrates the situation when a developer invokes Camel and searches for posts discussing how to use XML. The input code is pretty simple, and the developer would benefit from being suggested with SO posts that provide discussions related to the input source code.

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

Listing 5.1: 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 post[1] shown in Figure 5.1.

[1] https://tinyurl.com/yydp8lwd

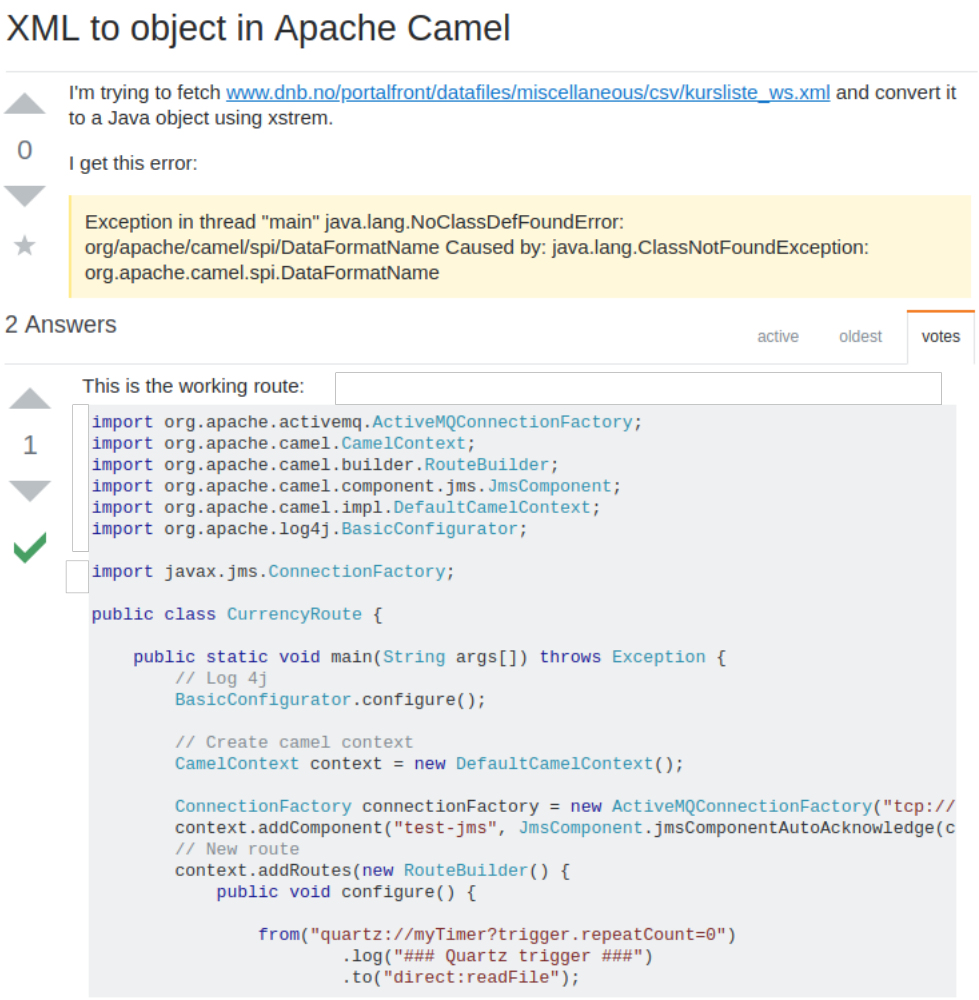


Figure 5.1: A Stack Overflow post relevant to Listing 5.1.

For the sake of clarity, 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 both of them are useful for the context. For instance, the depicted snippet contains class `CurrencyRoute` where `addRoutes()` is informative and the function `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 5.1 since it provides a better insight into how to use the related APIs.

When the context code shown in Listing 5.1 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 crucial 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 a 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. By doing this, we are able to enrich the initial query with multiple pieces of information. Furthermore, we also increase the exposure of the background data collected from Stack Overflow by means of the boosting mechanism provided by Lucene. In fact, the post in Figure 5.1 is recommended by PostFinder when we feed it with the query in Listing 5.1. This indicates that for the motivating example, our tool is more effective compared to FaCoY. In the next section, we are going to explain in detail the proposed approach.

5.2 The proposed approach

Given a user context consisting of the source code under development, we aim at searching for posts that contain highly relevant answers from Stack Overflow. 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 PostFinder, 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 against the indexed data. Concerning *Query Execution*, the actual searching is performed employing Apache Lucene.

The index creation, query creation, and query execution phases are completely transparent to the user, who can ask for SO posts directly from the implemented Eclipse IDE plugin as shown in Figure 5.2. In particular, the user selects the source code that she wants to include in the query ①, asks for the recommendation and gets a list of retrieved SO posts ②.

An overview of the PostFinder building components is depicted in Figure 5.3. The three constituent phases, i.e., *Index Creation*, *Query Creation*, and *Query Execution* are described in the following subsections.

5.2.1 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

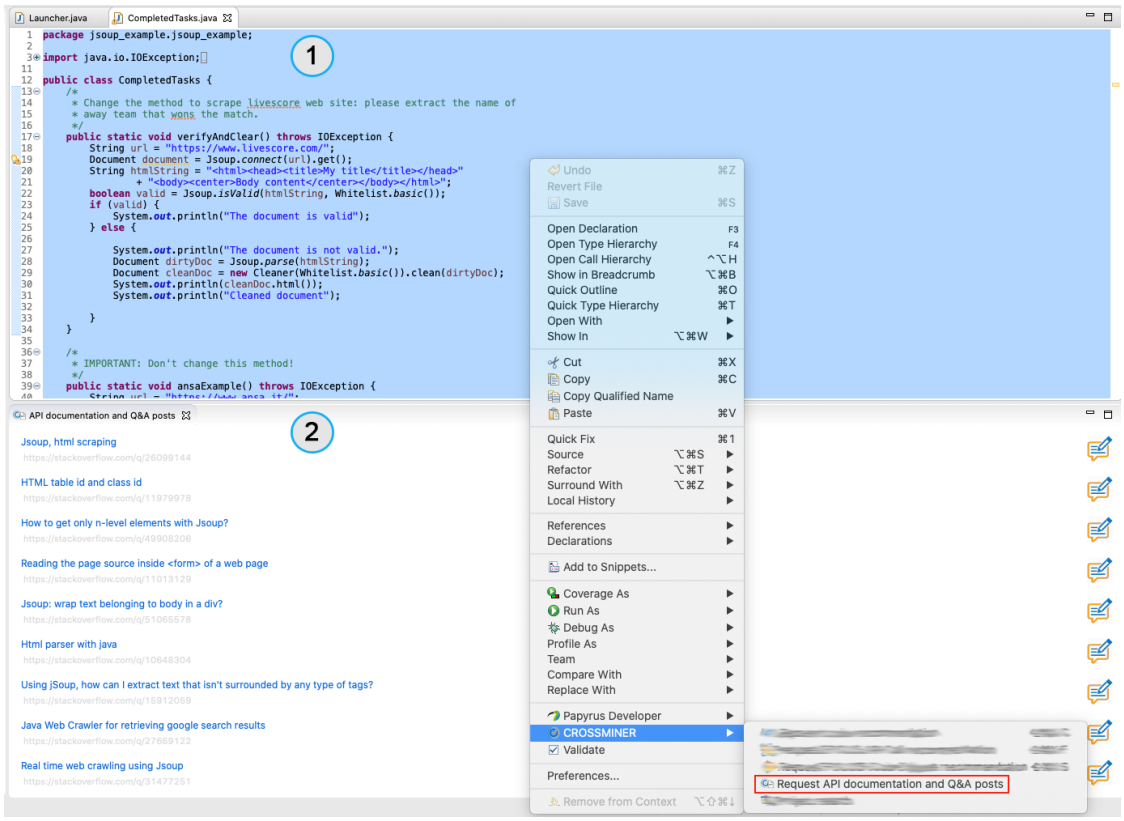


Figure 5.2: PostFinder at work.

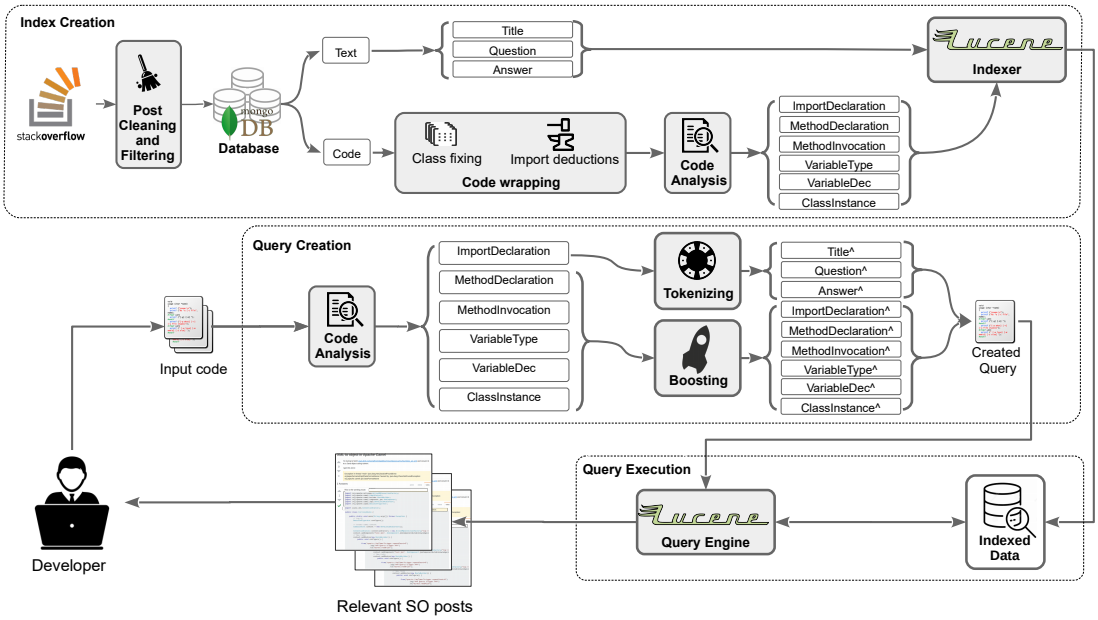


Figure 5.3: The PostFinder architecture.

point, it is necessary to use indexed data for future look up. We opt for Apache Lucene as it is a powerful IR tool 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 using 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 the `Code` part of Table 5.1. However, code snippets in Stack Overflow posts may neither be complete nor compilable [125], and thus they cannot 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.

Table 5.1: The used facets.

	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

5.2.1.1 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, and they are considered to be 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` tag[2] to include in their bodies Java source code are accepted for further processing.

5.2.1.2 Code wrapping

To deal with incomplete and uncompilable snippets, e.g., those that have neither class structure nor import directives, we use the parsing option to yield more tokens. For code snippets without any imports, we wrap up with relevant classes to make them more informative. We exploit an archive provided by Benelallam *et al.* [126] to find the right library that an invocation belongs to, and thus, the corresponding `import` directives. The dataset contains

[2]We 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.

more than 2.8M artifacts together with their dependencies as well as other relationships, e.g., versions. We count the frequency that each 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 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. It is important to remark that the choices related to the different design alternatives, threshold values, augmentation mechanisms, etc. have been performed iteratively and empirically to maximize the accuracy of the overall approach.

```
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 5.2: Original code snippet.

We consider an example as follows: Listing 5.2 depicts a code snippet extracted from SO. The code contains just function calls, and it is incomplete since there is neither class declaration nor 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.

```
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 5.3: Augmented code snippet.

By adding class fix and import, we obtain a new code snippet as shown in Listing 5.3. The snippet resembles a real hand-written code, which probably facilitates the matching process later on.

Once the refinement steps have been done, all terms corresponding to the tokens specified in the `Code` part of Table 5.1 are indexed and stored into Lucene for further look up.

5.2.2 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.

5.2.2.1 Boosting

The original code is parsed to obtain the six tokens listed in the last half of Table 5.1. Each term in the code is assigned a concrete weight to boost the level of importance. Entropy [127] is exploited to compute the quantity of information of a document using the following formula: $H = -\sum p(x)logp(x)$, 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 2^{nd}, 3^{rd}, and 4^{th} 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.

5.2.2.2 Tokenizing

By the *Index Creation* phase in Section 5.2.1, nine different tokens have been populated (see Table 5.1). Among them, there are three textual tokens, i.e.,`Title`, `Answer`, and `Question`. However, by the developer's side, the input context contains just 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 [128] shows that in an SO post, the title is more important than the description. In particular, the importance ratio

between description and title of a given post is 1/3 [128]. Accordingly, we set a boost value of 4 to the title and 1.4 to both the answer and question.Empirical evaluations conducted on the dataset demonstrate that the boost value has a smaller impact than that of the ratio between the title and the body of the post. Thus, we integrate it into the best query configuration.

By considering the code in Listing 5.1, the query ready to be executed that PostFinder creates after the boosting and tokenizing phases is shown in Listing 5.4.

```
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
```

Listing 5.4: Sample query produced after the *Boosting* and *Tokenizing* phases.

5.2.3 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.

$$R(q,d) = \sum_{t \in q} \frac{f_t^d}{k_1((1-b) + b\frac{l_d}{avgl_d} + f_t^d} \tag{5.1}$$

where f_t^d is the frequency of term t in document d; l_d is the length of the document d; $avgl_d$ 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 constrast, 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 [129].

By considering the query in Listing 5.4 as input, the post shown in Figure 5.1 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.

5.3 Evaluation

As mentioned above, PostFinder is a multi-purpose tool: On the one side, it can work as an independent engine to search for suitable Stack Overflow posts to directly support developers while they are programming. On the other side, it can be used to replace the first module of FaCoY. For the former, the most relevant system with PostFinder is PROMPTER [6] whose original implementation is, unfortunately, no longer functioning.[3] Considering the fact that the re-implementation of the tool is not straightforward and time consuming, we decided to investigate what queries would be potentially hard for PROMPTER to answer but for PostFinder would be easy, and vice versa by exploiting some use case studies from the original PROMPTER paper and apply PostFinder on them.

By the latter, since the source code of FaCoY is available, in this work we concentrate on comparing PostFinder with FaCoY by means of an evaluation on a common dataset. For the sake of representation, from now on the baseline is addressed as FaCoY (unless otherwise stated), despite the fact that it is the first module, albeit the most decisive one, of the whole FaCoY system [62].

The following subsections describe the evaluations in detail. In particular, the research questions are introduced in Section 5.3.1. Afterwards, we present a discussion about the high-level differences between PostFinder and PROMPTER in Section 5.3.2. The experimental configurations for supporting the comparison of PostFinder and FaCoY are explained in Section 5.3.3. Section 5.3.4 presents the dataset used in the evaluation. Section 5.3.5 explains the evaluation methodology, and Section 5.3.6 introduces the evaluation metrics.

5.3.1 Research questions

The evaluations are conducted to answer the following research questions:

- **RQ**$_1$: *Which experimental configuration brings the best PostFinder performance?* We compare the flat configuration with the augmented ones to see which setting fosters the best recommendation outcome for PostFinder.

[3] Through private communications, the authors of PROMPTER informed us that the tool had not been maintained for a long time, and they also had difficulties in making it run again.

- **RQ$_2$**: *How does PostFinder compare with FaCoY?* Compared to FaCoY, our tool 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.

- **RQ$_3$**: *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.

5.3.2 Differences between PostFinder and PROMPTER

We investigate the differences between the first ranked recommended post of the two approaches by relying on a set of queries and the corresponding results obtained by PROMPTER, provided by its authors.[4] To make the comparison fair as much as possible, we filter PostFinder's posts by selecting only those that date back to the year 2014, i.e., the date of the dump used by PROMPTER [6]. As the first example, depicted in Listing 5.5, we present a particular developer's context in which PROMPTER performs better than PostFinder with respect to the retrieved first top rank posts. As we can see, the post[5] recommended by PROMPTER shown in Figure 5.4 is more relevant given the context rather than the one[6] retrieved by PostFinder shown in Figure 5.5. The rationale behind this fact can be found in the query's lines of code. In particular, PostFinder misses some valuable results when it is fed with a very small context, as there are few import statements in such cases.

```
1   /*
2   List Filesystem roots
3   */
4   import java.io.*;
5   public class ListFileSystemRoots {
6           public static void main(String[] args) {
7                   File[] rootDirectories = File.listRoots();
8                   System.out.println("Available root directories in filesystem are: ");
9                   for(int i=0 ; i < rootDirectories.length ; i++)
10                          System.out.println(rootDirectories[i]);
11          }
12  }
```

Listing 5.5: Comparing PostFinder with PROMPTER: First use case.

In contrast, the example shown in Listing 5.6 demonstrates that PostFinder is capable of providing a more relevant post. Given a more detailed context, PROMPTER's post[7]

[4] We gratefully acknowledge the data provided by Prof. Dr. Michele Lanza and Dr. Luca Ponzanelli.
[5] https://tinyurl.com/ybmny6p8
[6] https://tinyurl.com/y8zarqsb
[7] https://tinyurl.com/ybe37xch

Figure 5.4: A post related to Listing 5.5 retrieved by PROMPTER.

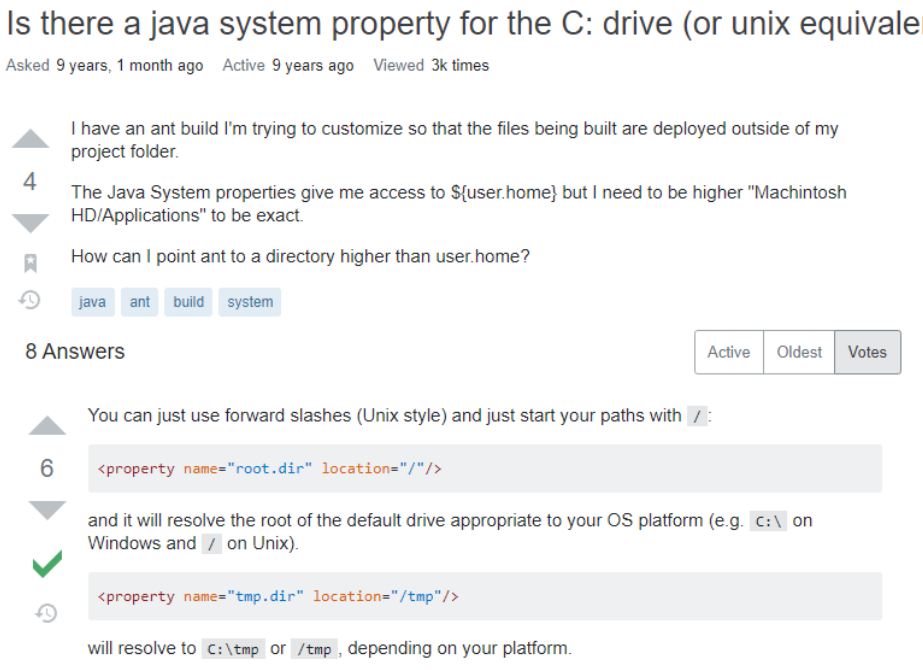


Figure 5.5: A post related to Listing 5.5 retrieved by PostFinder.

reported in Figure 5.6 misses some additional information that PostFinder can deliver[8] as shown in Figure 5.7. Moreover, more results could be retrieved by PostFinder using the wrapping technique described in Section 5.2.1.2. We cannot directly verify this claim as the PROMPTER's queries included in the provided set do not fall into this category, i.e., they contain compilable snippets of code.

```
1  package varius;
2
3  import javax.swing.JFrame;
4  import javax.swing.JPanel;
5  import javax.swing.JProgressBar;
6
7  //Define a vertical progress bar
8  public class ProgressBar extends JFrame {
9          JProgressBar current = new JProgressBar(JProgressBar.VERTICAL, 0, 2000);
10         int num = 0;
11         public ProgressBar() {
12                 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13                 JPanel pane = new JPanel();
14                 current.setValue(0);
15                 current.setStringPainted(true);
16                 pane.add(current);
17                 setContentPane(pane);
18         }
19
20         public void iterate() {
21                 while (num < 2000) {
22                         current.setValue(num);
23                         try {
24                                 Thread.sleep(1000);
25                         } catch (InterruptedException e) {      }
26                         num += 95;
27                 }
28         }
29         public static void main(String[] arguments) {
30                 ProgressBar frame = new ProgressBar();
31                 frame.pack();
32                 frame.setVisible(true);
33                 frame.iterate();
34         }
35 }
```

Listing 5.6: Comparing PostFinder with PROMPTER: Second use case.

According to the performed light-weight comparison, we noticed that PostFinder performs better than PROMPTER when the input query contains more import statements even in case of code that is not compilable. However, to deliver a fair and a thorough comparison of both approaches, it is necessary to re-implement the PROMPTER tool by strictly following the descriptions in the original paper [6], and we consider this as a possible future work.

[8] https://tinyurl.com/y9o38kq6

Figure 5.6: A post related to Listing 5.6 retrieved by PROMPTER.

5.3.3 Configurations

To study PostFinder's performance, we perform two main evaluations by means of user studies. The first one is done to evaluate the role of each augmentation proposed in Section 5.2. To this end, we consider six experimental configurations, i.e., **A**, **B**, **C**, **D**, **E**, **F** with 10 queries for each (see Table 5.2). The second evaluation compares PostFinder with FaCoY, and this corresponds to the last configuration **G**. 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 rationale

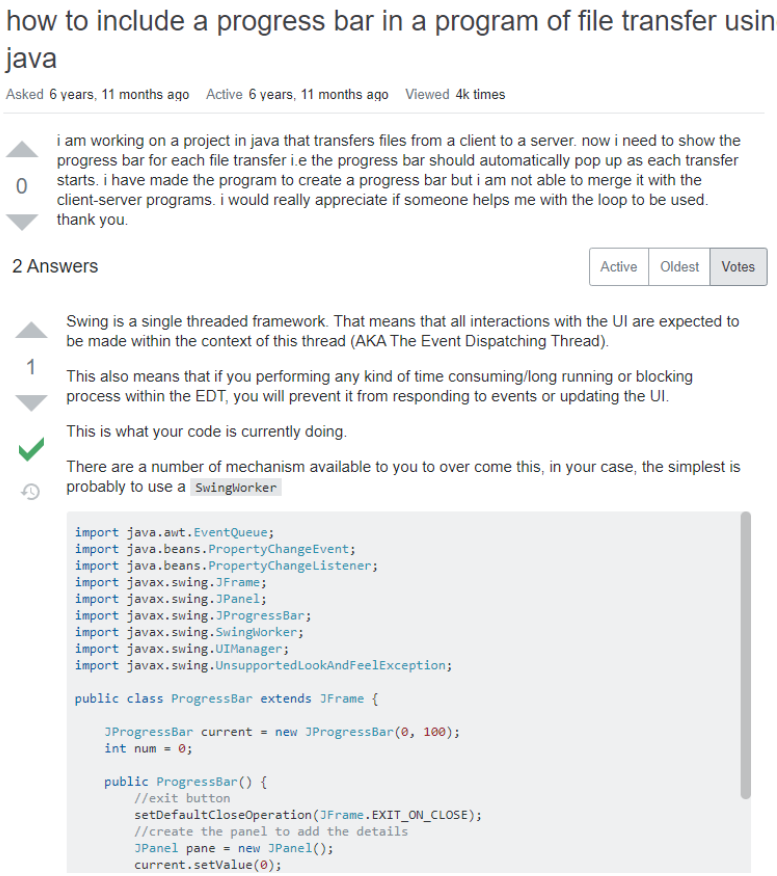


Figure 5.7: A post related to Listing 5.6 retrieved by PostFinder.

behind the selection of such libraries is that through a careful observation, we realized that generally, feeding FaCoY with queries coming from random libraries might not yield any results, i.e., the anwer is blank, and such an outcome is not useful for any comparison. Thus, we have to select queries that return some posts which can eventually be used to evaluate both FaCoY and PostFinder.

The test configurations are explained in Table 5.2, whose the 2^{nd} to 5^{th} columns specify the presence of the techniques mentioned in Section 5.2, with the corresponding section being shown in parentheses. For example, the 2^{nd} column **Wrapping** is a combination of *class fixing* and *library import deduction* (see Section 5.2.1). The column **Boosting** indicates the usage of the entropy for the calculation of the boosting values (see Section 5.2.2.1), **Tokenizing** refers to the usage of a *Maven dataset* in order to augment the imports (see Section 5.2.2.2). Finally column **Ranking** dictates the use of the BM-25 weighting scheme (see Section 5.2.3). To facilitate future research, we made available the PostFinder tool together with the related data in GitHub [123].

Table 5.2: Experimental configurations.

Configuration	Wrapping	Boosting	Tokenizing	Ranking	# queries	Description
A	✗	✗	✗	✗	10	Flat queries, without considering any proposed augmentations
B	✓	✗	✗	✗	10	Wrapping is introduced to queries in Configuration 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 PostFinder and FaCoY

5.3.4 Dataset

To provide input for the evaluation, we exploited an Stack Overflow dump[9] of June 2017, 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 have no imports. We fixed them as presented in Section 5.2.1.2. Eventually, we indexed and parsed all the posts following the paradigm described in Section 5.2.1. More details of the dataset used in our evaluation are shown in Table 5.3.

Table 5.3: A summary of the SO dump used in the evaluation.

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

5.3.5 User study

We resort to user studies as this is the only way to investigate whether the recommendation outcomes are really helpful to solve a specific task [30, 26, 6]. The input is expressed as a snippet of code together with a set of recommended posts, and the task is to judge how relevant the retrieved posts are, with respect to the input code. A group of 11 developers

[9] https://archive.org/details/stackexchange

was asked to participate in the user studies. Six participants were master students attending a Software Engineering course. Three of them were first year PhD students, and the other two were postdoc researchers. Through a survey sent to each participant, we found out that more than a half of them had at least seven 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, Stack Overflow, or Maven on a daily basis. Furthermore, they frequently re-use code fragments collected from these external sources. This is advantageous for the manual evaluation, since in the scope of this work we consider only source code written in Java, and we assume that skilful Java developers shall have a better judgment about the relevance among the code being developed and the recommended posts. The knowledge of different programming languages, e.g., Perl, Python, is also a plus for the evaluation process. Given a query, each system, i.e., FaCoY or PostFinder produces as outcome a ranked list of posts, considered to be relevant. To aim for a fair comparison, we mixed the top-5 results generated by each system in a single Google form and presented them to the evaluators together with the corresponding context code. This simulates a *taste test* [111] 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 possible bias or prejudice against a specific system. Each pair of code and post, i.e., <*query, retrieved post*> is examined and evaluated by at least two participants. Each evaluator first inspects the input code snippet, then reads the post including text and source code to comprehend its purpose, and finally judges the relevance using the criteria listed in Table 5.4.

Table 5.4: Relevance scores.

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 and the related code snippets are useful considering the development's context

Apart from 11 developers mentioned before, we also involved one more researcher in validating the evaluation outcome by playing the role of a *mediator*. In case there is a substantial disagreement between any two participants, e.g., the first person assigned the score of 2 and the other one gave 4 to a same pair (see Table 5.4), the mediator examines the pair again to eventually reach a consensus. It is worth noting that to avoid bias, also in this phase the mediator did not know ex-ante where each pair comes from, i.e., FaCoY or PostFinder. In most cases, the evaluators agreed on the scores. The disagreement happened

mainly within closely relevant scores: 1 with 2, or 3 with 4. Only when there were pairs with very different scores, e.g., 1 with 3, or 2 with 4, the mediator was asked to re-evaluate and judge. The ratio of such a disagreement was 15%. Moreover, an inter-rater agreement analysis using Cohen's Kappa test [130] revealed that the mediator substantially agreed with both evaluators, $\kappa(eval_1, mediator) = 0.677$ and $\kappa(eval_2, mediator) = 0.761$.

5.3.6 Evaluation metrics

As typically done in related work, the following metrics have been considered to evaluate the recommendation outcomes [30, 26]:

- *Relevance*: it is the score given to a pair of <*query, retrieved post*> following Table 5.4;
- *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.

5.4 Results

This section presents the results obtained from the experiments as well as related discussions. In Section 5.4.1, we analyze the outcomes obtained by performing PostFinder with six configurations (see Table 5.2), to answer $\mathbf{RQ}_1$. Afterwards, we compare PostFinder with FaCoY by answering $\mathbf{RQ}_2$. We attempt to reason what constitutes the performance differences between the two systems in $\mathbf{RQ}_3$. Threats to validity of the performed experiments are discussed in Section 5.4.2.

5.4.1 Result analysis

$\mathbf{RQ}_1$: *Which experimental configuration brings the best PostFinder performance?*

Every 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 relevance scores for the configurations and represent them in a Likert bar chart shown in Figure 5.8. It is evident that performing PostFinder with flat queries, i.e., configuration **A**, yields the worst performance: 30% of the posts are either totally irrelevant, or irrelevant, and 70% of them are relevant or highly relevant. This suggests that feeding queries without incorporating

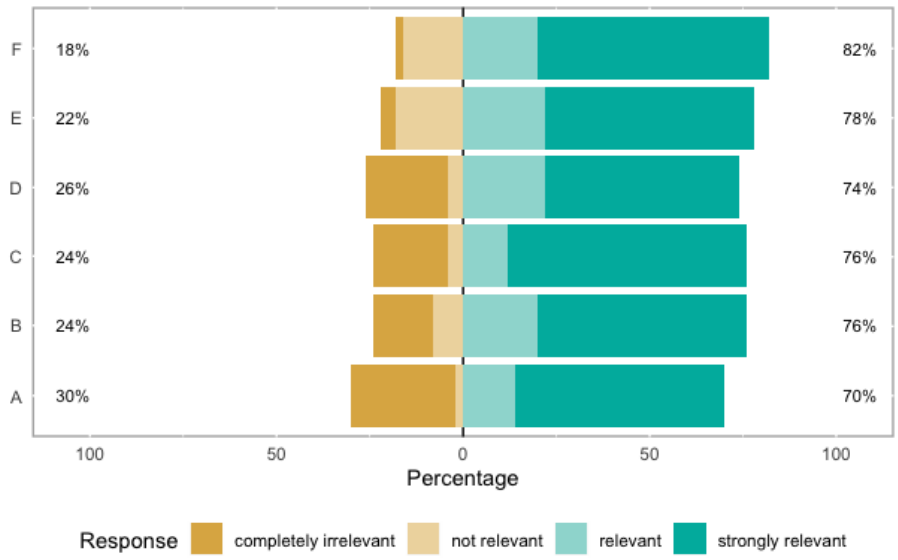


Figure 5.8: Relevance scores for PostFinder.

any proposed augmentations brings least 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 bars have a similar shape. Among others, the best relevance is seen when running PostFinder with **F**, i.e., all proposed augmentations are incorporated. In particular, no query pair gets 1 as the relevance score and 82% of them are considered as relevant or strongly relevant. This necessarily means that augmenting queries with all the proposed measures helps retrieve highly relevant posts.

Table 5.5: PostFinder success rate and precision.

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

Success rate is a superficial metric and it does not reflect well the outcome's traits. For instance, given a query, a system that gets 1 matched result has a success rate of 100%, which is exactly the same as that of another system which gets all 5 matched posts for the same query, however, the two systems are not equal in quality. Thus, further than *Success rate* we also measured *Precision* as shown in Table 5.5. Concerning *Precision*, we see that using flat query obtains the lowest precision, i.e., $Precision = 0.6$ and this is consistent with *Relevance*

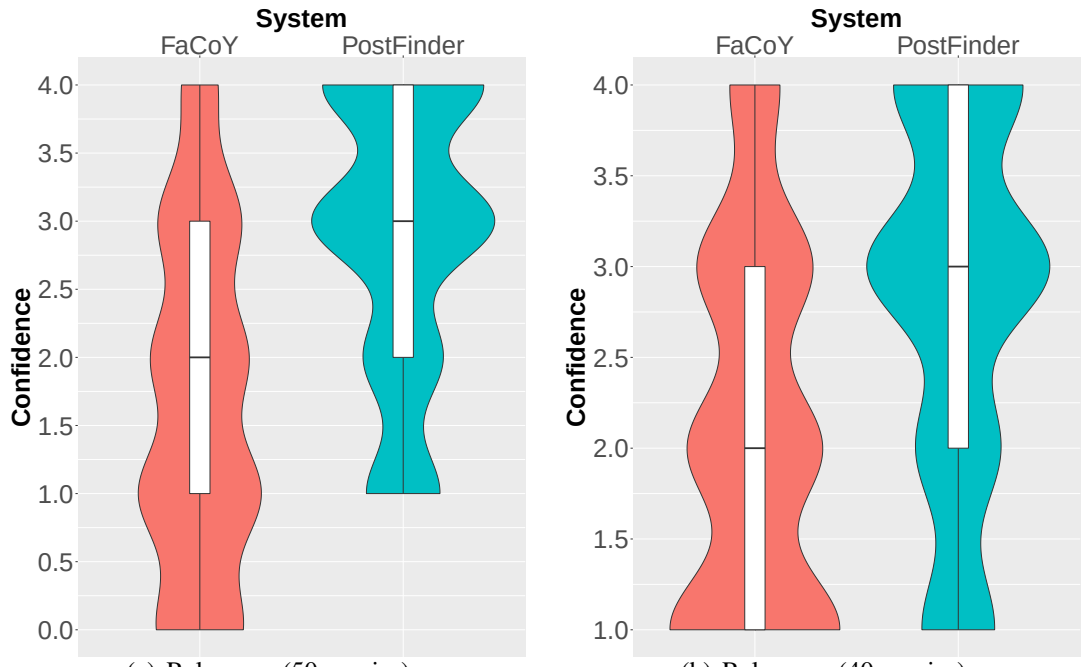
in Figure 5.8. Again, the best *Precision*, i.e., 0.82 is obtained when all augmentations are imposed on the queries. Running PostFinder on the dataset always gets a minimum success rate of 0.90, regardless of the configuration.

According to the performed experiments, PostFinder seems to perform better with Configuration **F** compared to **E**. However, we anticipate that to assess to what extent Tokenizing (see Section 5.2.2.2) contributes to better results in a statistically significant manner, a larger dataset would be needed. In fact, tests that can be used for measuring the significance of the results, e.g., Wilcoxon rank sum test, do not work well for small samples [131]. Thus, according to the performed experiments, tokenizing contributes much to the matching of relevant posts. Understanding if this is statistically significant needs a bigger dataset, and this is deferred as a future work.

Answer to RQ$_1$. In summary, running PostFinder by deploying all proposed augmentations provides the best performance with respect to relevance, success rate, and precision.

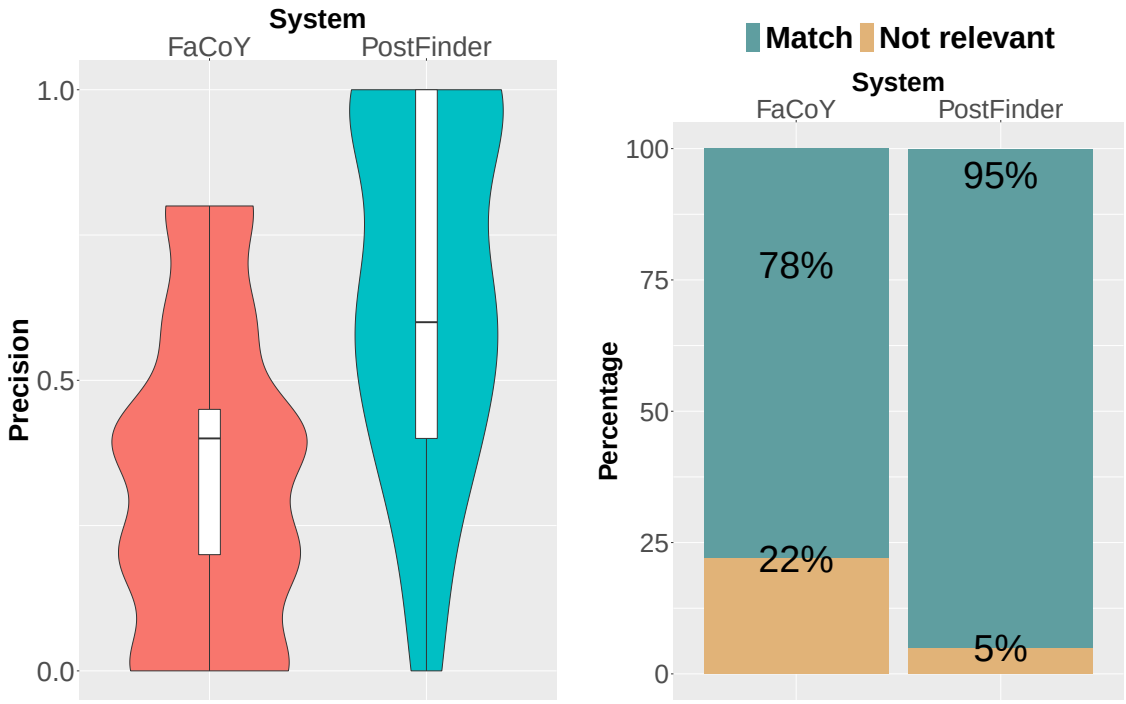
RQ$_2$: *How does PostFinder compare with FaCoY?*

Considering the set of 50 queries, PostFinder 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 5.4). We depict the relevance scores



(a) Relevance (50 queries) (b) Relevance (40 queries)

Figure 5.9: Relevance for Configuration **G**.



(a) Precision (40 queries) (b) Success rate (40 queries)

Figure 5.10: Precision and success rate for Configuration **G**.

of both systems using violin boxplots in Figure 5.9(a). 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 [132]. The boxplots demonstrate that PostFinder gains a considerably better relevance than that of FaCoY. In particular, PostFinder 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 5.1 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 relevance scores in Figure 5.9(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 PostFinder lies on the upper part of the figure, starting from 3 in the vertical axis. We conclude that PostFinder obtains a better relevance compared to that of FaCoY.

We further investigate the systems by considering Figure 5.10(a) 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

Table 5.6: Statistical analysis for the used metrics.

	Success rate		**Precision**		**Relevance**	
	PostFinder	FaCoY	PostFinder	FaCoY	PostFinder	FaCoY
Mean	0.95	0.77	0.66	0.33	2.78	2.09
std	0.22	0.42	0.30	0.26	1.01	1.01
$1^{st} quartile$	1.0	1.0	0.40	0.20	2.0	1.0
$2^{nd} quartile$	1.0	1.0	0.60	0.40	3.0	2.0
$3^{rd} quartile$	1.0	1.0	1.0	0.45	4.0	3.0
p-value	2.00e-02		8.90e-06		1.08e-10	
Cliff's delta	-0.175 (small)		-0.568 (large)		-0.360 (medium)	

precision lower than 0.5. In the opposite side, PostFinder gains better precisions that are larger than 0.5, and agglomerate to the upper bound, i.e., 1.0. The metric shows that, given a same query PostFinder returns more relevant posts than the baseline does.

The obtained success rates for both systems are shown in Figure 5.10(b). 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, PostFinder achieves a better percentage of success, 38 among 40 queries are successful, yielding a success rate of 95%.

It is important to understand if the performance difference is statistically significant. Thus, we performed a statistical analysis and the obtained results are shown in Table 5.6. We compute Wilcoxon rank sum test [133] on the scores obtained by the systems and get the following results: p-value for Success rate is *2.00e-02*; p-value for Precision is *8.90e-06*; p-value for Relevance is *1.08e-10*. The null hypothesis is that there are no differences between the performance of FaCoY and that of PostFinder. 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 PostFinder is statistically significant. We also calculated the effect size by means of Cliff's delta [134], and the results reveal that the effect size is *small* with Success rate, but *medium* and *large* by Relevance and Precision, respectively. Considering all the metrics, we see that the improvement gained by PostFinder is significant and meaningful.

Answer to RQ$_2$. PostFinder outperforms FaCoY in terms of relevance, success rate, and precision. Furthermore, the performance difference between the two systems is statistically significant.

RQ$_3$: *What are the reasons for the performance difference?*

We refer back to the example introduced in Section 5.1. Actually, the post in Figure 5.1 is recommended by PostFinder when the code in Listing 5.1 is used as query. Compared to the baseline, PostFinder 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 PostFinder, we see that the transformation of import directives to produce textual tokens as shown in Listing 5.4 is beneficial to the search process: it equips the query with important terms which then match with the post's title. Table 5.7 distinguishes between the two systems by listing the facets exploited by each of them.

Table 5.7: Comparsion between FaCoY and PostFinder.

Technique	FaCoY	PostFinder
Code analysis	✓	✓
Class fixing	✓	✓
Import mining	✗	✓
Entropy	✗	✓
BM25	✗	✓
Tokenizing	✗	✓

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 5.1 to search. As shown in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, 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, PostFinder attempts to improve the baseline by imposing various boosting measures. By the *Index Creation* phase, PostFinder enriches incomplete code snippets with class and import directives and then tokenizes them. By the *Query Creation* and *Execution* phases, PostFinder exploits import directives from source code to build indexes to match against indexed textual data. In other words, we make source code compilable, which was originally uncompilable, though the augmented code is inauthentic. This is important since a lot of code in Stack Overflow is uncompilable and cannot be indexed to Lucene. Since FaCoY 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.

Answer to $\mathbf{RQ}_3$. Altogether, the query boosting scheme and the considered facets for the creation of indexes are attributed to the performance difference between the two systems.

5.4.2 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 an additional 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.

The generalizability might be negatively affected by the conducted user study. In particular, the 11 selected participants have the same academic background, though their programming skills are of different levels. On the one hand, generalizing the obtained results to a larger population that includes different backgrounds can contribute to enriching the results in terms of coverage. On the other hand, gathering people with a wide range of programming experiences is a real challenge. Thus, we limit ourselves to the mentioned study to obtain a stable evaluation. Moreover, we mitigate this threat by involving one additional researcher who plays the role of mediator in the evaluation.

5.5 Conclusion

We introduced the PostFinder approach to provide highly relevant SO posts, given an input code snippet as the context. PostFinder addresses both the problem of adequately indexing SO posts, and that of automatically creating queries in a transparent manner for the developer. In particular, PostFinder performs different augmentations of SO posts for indexing them, and of input contexts for creating corresponding queries.

To study the performance of PostFinder, we performed two different user studies. The first study has been done in order to understand which combination of the conceived augmentations is the best one in terms of PostFinder performance. While the second one, which is a larger user study has been done to compare PostFinder with FaCoY. The experimental results show that PostFinder outperforms the corresponding module of FaCoY, which is devoted to searching SO posts that are relevant with input developer contexts.

The implementation of PostFinder is twofold. First, we already integrated it into the Eclipse IDE to directly support developers in real-world settings. Second, the tool can be used to substitute the module for searching Stack Overflow posts by FaCoY, and we are now working on the replacement, aiming to boost up the system's overall performance.

To showcase our contribution, we evaluated PostFinder using different metrics and assessed the relevance of the retrieved posts compared to those obtained by the baseline. The results confirm that the improvement of our tool is significant and meaningful. Our future research agenda focuses on performing further evaluations, especially to compare PostFinder with those approaches that rely on general-purpose search engines and that focus only on the query creation phase (e.g., PROMPTER [6]). Last but not least, though our proposed approach 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 this as future work.

Chapter 6

Upgrading third-party libraries with migration graph and deep learning

While working with a software project, developers often use third-party libraries (TPLs) that offer tailored functionalities [17, 135–137]. Existing TPLs enable developers to exploit ready-to-use programming utilities without needing to reinvent everything from scratch. However, TPLs are subject to change [135], and to keep up with the new functionalities, in a software project, developers need to replace an old library with a more updated one. In fact, adopting the wrong version of a library might cause unavoidable disruptions [138] due to conflicts or breaking changes in the hosting code [137]. In this respect, the migration of TPLs is complex and susceptible to errors that need to be thoroughly managed to avoid any negative impacts on the entire project [135]. Being afraid of incompatibility and breaking changes [139], developers tend to procrastinate the upgrade of TPLs [73, 138]. They prefer to stay in the *comfort zone*, keeping the most stable versions and continuingly ignoring the accumulative *maintenance debt* [136]. A recent empirical study [140] shows that more than 50% of the considered projects never update more than half of their libraries. However, delaying the updates of used libraries, due to such difficulties and the related ripple effects, can harm software systems from different points of view, including security. Moreover, it can increase the accumulated technical debt, which is a measure reflecting *"the implied cost of additional rework, caused by deciding for an easy solution now instead of deciding for a better choice that would take longer to be implemented"* [141].

In this chapter we present two approaches conceived to support developers in upgrading their third-party libraries. First, we propose EvoPlan, a recommender systems for providing upgrade plans for TPLs. By exploiting the experience of other projects that have already performed similar upgrades and migrations, EvoPlan recommends the upgrade plan that should be considered to migrate from the current library version to the desired one. A

graph-based representation are inferred by analyzing GitHub repositories and their *pom.xml* files. During this phase, EvoPlan assigns weights representing the number of client projects that have already performed a specific upgrade. EvoPlan employs a shortest-path algorithm to minimize the number of upgrade steps considering such weights. The tool eventually retrieves multiple upgrade plans to the user with the target version as well as all the intermediate passages.

Afterward, we propose DeepLib, a novel approach to the recommendation of library upgrades, exploiting cutting-edge deep learning techniques. In particular, DeepLib has been built on top of long short-term memory and encoder-decoder neural networks to predict future versions of libraries. Such techniques have been successfully applied in various domains, including machine translation [2]. By analyzing the migration history of mined projects, we build matrices containing libraries and their versions in chronological order, which are fed to the recommendation engine. As output, DeepLib delivers *(i)* the next version for a single library *lib* that the developer would like to upgrade for the project at hand; and *(ii)* the next version for a set of libraries that should also be upgraded due to the recommended upgrade for library *lib*. For the former, we built a long short-term memory recurrent network [142] (LSTM), while for the latter, we developed an Encoder-Decoder LSTM [143] to predict a set of versions.

Structure of the chapter. Section 6.1.2 presents the architecture of EvoPlan. In Section 6.1.3 and Section 6.1.4, we report the conducted evaluation and the obtained results, respectively. Section 6.1.5 highlights the possible threats to validity. DeepLib is presented in the second part of the chapter. In particular, Section 6.2.1 reviews the underpinning technology, and Section 6.2.2 provides a detailed description of the DeepLib approach. We describe the evaluation procedure in Section 6.2.3 and the collected results in Section 6.2.4. The applicability of DeepLib is discussed in Section 6.2.5. Finally the chapter is concluded in Section 6.3.

6.1 EvoPlan: Providing upgrade plans with migration graph

6.1.1 Background

TPLs offer several tailored functionalities, and invoking them allows developers to make use of a well-founded infrastructure, without needing to re-implementing from scratch [17]. Eventually, this helps save time as well as increase productivity. However, as libraries evolve over the course of time, it is necessary to have a proper plan to migrate them once they have

been updated. So far, various attempts have been made to tackle this issue. In this section, we introduce two motivating examples, and recall some notable relevant work as a base for further presentation.

Explanatory examples. This section discusses two real-world situations that developers must cope with during the TPLs migration task, i.e., code refactoring and vulnerable dependencies handling. In the first place, it is essential to consider different TPL releases that are conformed to the semantic versioning format.[1] A standard version string follows the pattern *X.Y*, in which *X* represents the *major* release and *Y* represents the *minor* one. Sometimes, releases can include a *patch* version *Z*, resulting in the final string *X.Y.Z*. We present an explanatory example related to *log4j*,[2] a widely used Java logging library. When it is upgraded from version *1.2* to version *1.3*, as shown in Listing 6.1 and Listing 6.2, respectively, a lot of internal changes happened which need to be carefully documented.[3] As we can notice, the main change affects the `Category` class which is replaced by the `Logger` class. Furthermore, all former methods that were used by the deprecated class cause several failures at the source code level. For instance, the `setPriority` method is replaced by `setLevel` in the new version.

```
1  Category root = Category.getRoot();
2  root.debug("hello");
3
4  Category cat = Category.getInstance(Some.class);
5  cat.debug("hello");
6
7  cat.setPriority(Priority.INFO);
```

Listing 6.1: log4j version 1.2.

```
1  Logger root = Logger.getRootLogger();
2  root.debug("hello");
3
4  Logger logger = Logger.getLogger(Some.class);
5  logger.debug("hello");
6
7  logger.setLevel(LEVEL.INFO);
```

Listing 6.2: log4j version 1.3.

Though this is a very limited use case, it suggests that the code refactoring that takes place during the migration is an error-prone activity even for a single minor upgrade, i.e., from version *1.2* to version *1.3*. Additionally, the complexity dramatically grows in the case

[1] https://semver.org/
[2] https://logging.apache.org/log4j/
[3] http://articles.qos.ch/preparingFor13.html

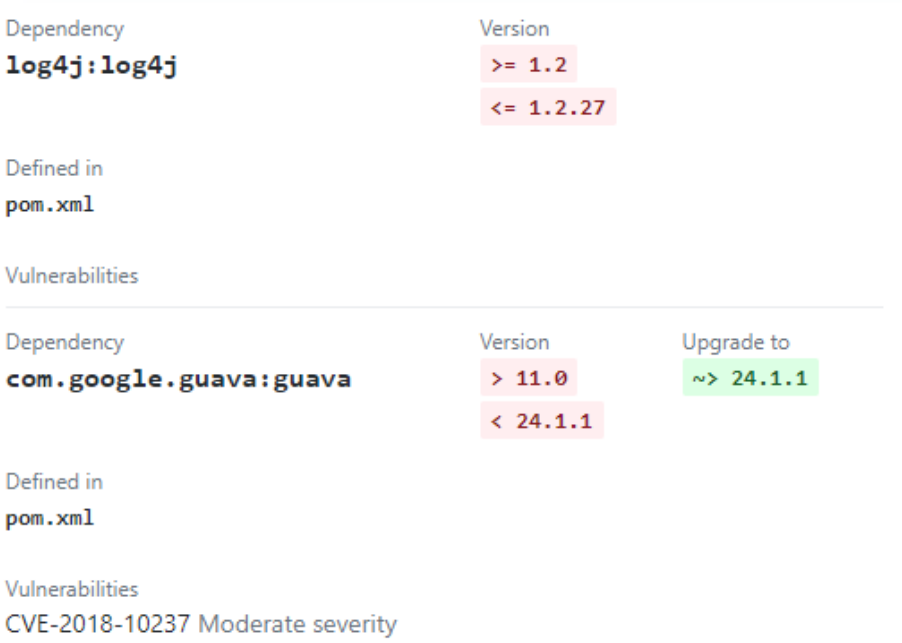


Figure 6.1: GitHub Dependabot alert.

of a major release as it typically requires extra efforts rather than a minor one which are not welcome by the majority of developers [73]. Considering such context, the reduction of the time needed for a single migration step, even a minor one, is expected to improve the overall development process.

Concerning vulnerable dependencies, GitHub Dependabot[4] provides weekly security alert digests that highlight possible security issues for outdated dependencies of a repository [144], which can be of different languages, e.g., Python, Java, JavaScript.[5] An example of a Dependabot report is shown in Figure 6.1.

As shown in Figure 6.1, Dependabot suggests possible TPL upgrades to solve vulnerabilities in the given project. For instance, the **guava** dependency seems to be outdated, and thus the system automatically suggests jumping to the latest version, i.e., *24.1.1*. Though this alert can raise awareness of this evolution, it does not offer any concrete recommendations on how to perform the actual migration steps. In some cases, the bot does not provide any recommended version to update the project, e.g., for the **log4j** dependence. In this respect, we see that there is an urgent need for providing recommendations of the most suitable plan, so as to upgrade the library, as this can significantly reduce the migration effort.

[4] https://dependabot.com/blog/github-security-alerts/
[5] https://dependabot.com/#languages

Existing techniques. This section reviews some relevant work that copes with the migration problem.

Table 6.1: Main features of TPLs migration systems.

System	Inferring migration	Incremental plan	Popularity	GitHub issues	Upgrading	Replacement	Applying Migration
Meditor [67]	✓	✗	✓	✗	✓	✓	✓
Apivawe [68]	✓	✗	✓	✗	✗	✓	✗
Graph Mining [69]	✓	✗	✓	✗	✗	✓	✗
RAPIM [145]	✓	✗	✓	✗	✗	✓	✓
Diff-CatchUp [146]	✓	✗	✓	✗	✓	✓	✗
M^3 [147]	✓	✗	✗	✗	✗	✓	✓
A4 [148]	✓	✗	✗	✗	✗	✓	✓
EvoPlan	✓	✓	✓	✓	✓	✗	✗

Meditor [67] is a tool aiming to identify migration-related (MR) changes within commits and map them at the level of source code with a syntactic program differencing algorithm. To this end, the tool mines GitHub projects searching for MR updates in the *pom.xml* file and check their consistency with the WALA framework.[6]

Hora and Valente propose Apiwave [68], a system that excerpts information about libraries' popularity directly from mined GitHub project's history. Afterwards, it can measure the popularity of a certain TPL by considering the import statement removal or addition.

Teyton *et al.* [69] propose an approach that discovers migrations among different TPLs and stores them in a graph format. A token-based filter is applied on *pom.xml* files to extract the name and the version of the library from the artifactid tag. The approach evetually exhibits four different visual patterns that consider both ingoing and outgoing edges to highlight the most popular target.

RAPIM [145] employs a tailored machine learning model to identify and recommend API mappings learned from previously migration changes. Given two TPLs as input, RAPIM extracts valuable method descriptions from their documentation using text engineering techniques and encode them in feature vectors to enable the underpinning machine learning model.

Diff-CatchUp [146] has been conceived with the aim of proposing usage examples to support the migration of reusable software components. The tool makes use of the UMLDiff algorithm [149] to identify all relevant source code refactorings. Then, a heuristic approach is

[6] https://github.com/wala/WALA

adopted to investigate the design-model of the evolved component and retrieve a customizable ranked list of suggestions.

Collie *et al.* recently proposed the M^3 tool [147] to support a semantic-based migration of C libraries. To this end, the system synthesizes a behavioral model of the input project by relying on the LLVM intermediate representation.[7] Given a pair of source and target TPLs, the tool generates abstract patterns that are used to perform the actual migration.

Conceived for mobile applications, A4 [148] performs TPL migration by extracting migration candidates directly from the Android documentation. Once the AST parsing phase has taken place, the system identifies a migration mapping to detect potential code changes to be implemented.

Table 6.1 summarizes the features of the above-mentioned approaches by considering the different tasks involved in migration processes by starting with the discovery of possible migration changes up to embedding them directly into the source code as explained below.

- *Inferring migration*: To extract migration-related information, tools can analyze existing projects' artifacts, i.e., commits, *pom.xml* file, or tree diff. This is the first step of the whole migration process.

- *Incremental plan*: The majority of the existing approaches perform the migration just by considering the latest version of a TPL. This could increase the overall effort needed to perform the actual migration, i.e., developers suffer from accumulated technical debt. In contrast, considering a sequence of intermediate migration steps before going to the final one can reduce such refactoring.

- *Popularity*: This is the number of client projects that make use of a certain library. In other words, if a TPL appears in the *pom.xml* file or in the import statement, its popularity is increased.

- *GitHub issues*: As an additional criterion, the migration process can include data from *GitHub issues* that may include relevant information about TPLs migration. Thus, we consider them as a possible source of migration-related knowledge.

- *Upgrading*: This feature means that the tool supports the upgrading of a TPL from an older version to a newer one. For instance, the migration described in Section 6.1.1 falls under this class of migration.

- *Replacement*: Differently from upgrading, replacement involves the migration from a library to a different one that exposes the same functionalities.

[7] https://llvm.org/

- *Applying migration*: It represents the final step of the migration process in which the inferred migration changes are actually integrated into the project.

Dimensions to be further explored. Even though several approaches successfully cope with TPL migration, there are still some development dimensions that need to be further explored. However, providing an exhaustive analysis is out of the scope of this section. Thus, we limit ourselves to identify some of them by carefully investigating the approaches summarized in Table 6.1. The elicited dimensions are the following ones:

- *D1: Upgrading the same library.* Almost all of the presented approaches apart from Meditor, focus on replacing libraries and very few support the upgrades of already included ones (see columns *Upgrading* and *Replacement* in Table 6.1).
- *D2: Varying the migration data sources.* During the inferring migration phase, strategies to obtain migration-related data play a fundamental role in the overall process. A crucial challenge should be investigating new sources of information besides the well-known sources e.g., Bug reports, Stack Overflow posts, and GitHub issues.
- *D3: Aggregating different concepts.* The entire migration process is a complex task and involves notions belonging to different domains. For instance, GitHub issues could play a relevant role in the migration process. A recent work [150] shows that the more comments are included in the source code, the lesser is the time needed to solve an issue. Neil *et al.* [151] extracted security vulnerabilities from issues and bug reports that could affect library dependencies.
- *D4: Identification of the upgrade plan.* Existing approaches identify and apply migrations by taking as input the explicit specification of the target version of the library that has to be upgraded. Providing developers with insights about candidate upgrade plans that might reduce the migration efforts can represent valuable support to the overall upgrade process.

In the present work, we aim to explore and propose solutions for the dimensions D1 and D4 by providing multiple possible upgrade plans given the request of upgrading a given library to target a specific target version. Furthermore, we also perform an initial investigation on the D2 and D3 dimensions, relying on GitHub issues. As it can be seen in Table 6.1, EvoPlan covers five out of the seven considered features. In particular, our approach is able to *infer migration*, make use of *incremental plan* by considering the *popularity* and *issues*, so as to eventually recommend an *upgrade plan*. Compared to the existing tools, EvoPlan tackles most of the issues previously presented.

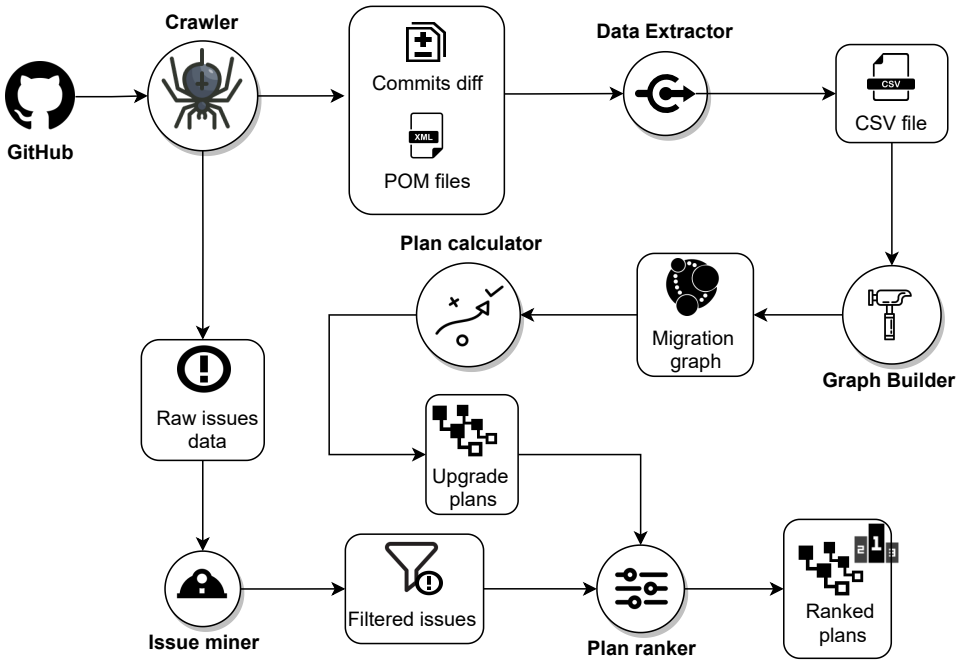


Figure 6.2: The EvoPlan architecture.

6.1.2 The proposed approach

In this work we propose an approach to support the first phase of the migration process, i.e., inferring the possible upgrade plans that can satisfy the request of the developer that might want to upgrade a given TPL used in the project under development.

Our approach aims at suggesting the most appropriate migration plan by taking into consideration two key factors: the *popularity* of the upgrade plan and the *availability of discussions* about it. Popularity means how many clients have performed a given upgrade plan, while discussions are GitHub issues that have been open and closed in projects during the migration phase.

By mining GitHub using the dedicated API,[8] we are able to extract the information required as input for the recommendation engine of EvoPlan.

The conceived approach is depicted in Figure 6.2 and consists of six components, i.e., *Crawler*, *Data Extractor*, *Graph Builder*, *Issues Miner*, *Plan Calculator* and *Plan Ranker*. With the *Crawler* component, the system retrieves information about GitHub repositories and downloads them locally. These repositories are then analyzed by the *Data Extractor* component to excerpt information about commits and history version. Once all the required information has been collected, *Graph Builder* constructs a migration graph with multiple

[8] https://developer.github.com/v3/

weights, and *Issues Miner* generates data related to GitHub issues. The *Plan Calculator* component relies on the graph to calculate the k-best paths available. Finally, *Plan Ranker* sorts these paths by considering the number of issues. In the succeeding subsections, we are going to explain in detail the functionalities of each component.

Crawler. Migration-related information is mined from GitHub using the *Crawler* component. By means of the `JGit` library,[9] *Crawler* downloads a set P of GitHub projects that have at least one *pom.xml* file, which isa project file containing the list of all adopted TPLs. In case there are multiple *pom.xml* files, they will be analyzed separately to avoid information loss. Then, the *Crawler* component analyzes all the repository's commits that affect the *pom.xml* to find added and removed TPLs. Additionally, raw issue data is obtained and stored in separate files. In particular, we count the number of opened and closed issues for each project $p \in P$ in a specific time interval D. The starting point of this interval is identified when a certain version v of a given library l that is added as dependencies of the *pom.xml* file in client C. A previous study [152] demonstrates that the monthly rate of open issues tends to decrease over time. Thus, the endpoint of D is obtained by considering the first two months of development to extract relevant data concerning the considered library l without loss of data. In such a way, we obtain the GitHub issues that have been opened and closed for each TPL that has been added in p.

Data Extractor. In this phase we analyze GitHub projects' data, which has been gathered by means of `JGit`. The first step makes use of the GitHub *log* command to retrieve the list of every modification saved on GitHub for a specific file. Furthermore, the command provides the code *SHA* for every commit, which allows us to identify it. For instance, Figure 6.3.a depicts a commit related to a given *pom.xml* file taken as input. The identifier of the commit is used to retrieve the list of the corresponding operated changes as shown in Figure 6.3.b. In particular, inside a commit we can find a large number of useful information like what was written or removed and when. The *Data Extractor* component focuses on the lines which contain an evidence of library changes. In a commit, the added lines are marked with the sign '+', whereas the removed ones are marked with '-' (see the green and red lines, respectively shown in Figure 6.3.b). In this way, the evolution of a library is obtained by analyzing the sequence of added/removed lines. With this information we are also able to count how many clients have performed a specific migration. The information retrieved by the *Data Extractor* component is stored in a target CSV file, which is taken as input by the subsequent entity of the process as discussed below.

[9] https://www.eclipse.org/jgit/

```
rick@rick-HP-EliteBook-830-G5:~/juddi$ git log pom.xml
commit a28db0a7a6f30a15bcdd588057a1cfb410a9ff28
Merge: 2c8d0f433 444c35a72
Author: spyhunter99 <spyhunter99@users.noreply.github.com>
Date:   Sun Mar 15 16:48:15 2020 -0400

    Merge pull request #7 from JLLeitschuh/fix/JLL/use_https_to_resolve_dependencies

    [SECURITY] Use HTTPS to resolve dependencies in Maven Build
```

a) Example of *log*

```
rick@rick-HP-EliteBook-830-G5:~/juddi$ git diff a28db0a7a6f30a15bcdd588057a1cfb410a9ff28
diff --git a/docs/asciidoc/ClientGuide/pom.xml b/docs/asciidoc/ClientGuide/pom.xml
index dd09f6d82..193da5bf5 100644
--- a/docs/asciidoc/ClientGuide/pom.xml
+++ b/docs/asciidoc/ClientGuide/pom.xml
@@ -19,7 +19,7 @@
     <parent>
         <groupId>org.apache.juddi.juddi-docs</groupId>
         <artifactId>juddi-guide-parent</artifactId>
-        <version>3.3.8-SNAPSHOT</version>
+        <version>3.3.8</version>
     </parent>
     <artifactId>juddi-client-guide</artifactId>
     <packaging>jdocbook</packaging>
```

b) Example of *diff*

Figure 6.3: Example of artifacts used by the *Data Extractor* component.

Graph Builder. This component creates nodes and relationships by considering the date and library changes identified in the previous phase. To this end, EvoPlan exploits the Cypher query language[10] to store data into a NEO4J graph. For instance, we extract from CSV files two pairs library-version *(l,v1)* and *(l,v2)* with signs '-' and '+', respectively. In this way, the component creates an oriented edge from *(l,v1)* to *(l,v2)*. Once the first edge is created, any further pairs containing the same library upgrade will be added as an incremented weight on the graph edge. The date value contained in the CSV record is used to avoid duplicated edges or loops. Furthermore, each edge is weighted according to the number of clients as described in *Data Extractor* phase. That means if we find w times the same couple *(l,v1)* to *(l,v2)* (i.e., a number of w projects have already migrated the library l from $v1$ to $v2$), the edge will have a weight of w. Thus, the final outcome ofthis component is a migration graph that considers the community's interests as the only weight. For instance, Figure 6.4 represents the extracted migration graph for the *slf4j-api* library. The graph contains all the mined version of the library and for each pair the corresponding number of clients that have performed the considered upgrade is shown. For instance, in Figure 6.4 the edge from

[10] https://neo4j.com/developer/cypher-query-language/

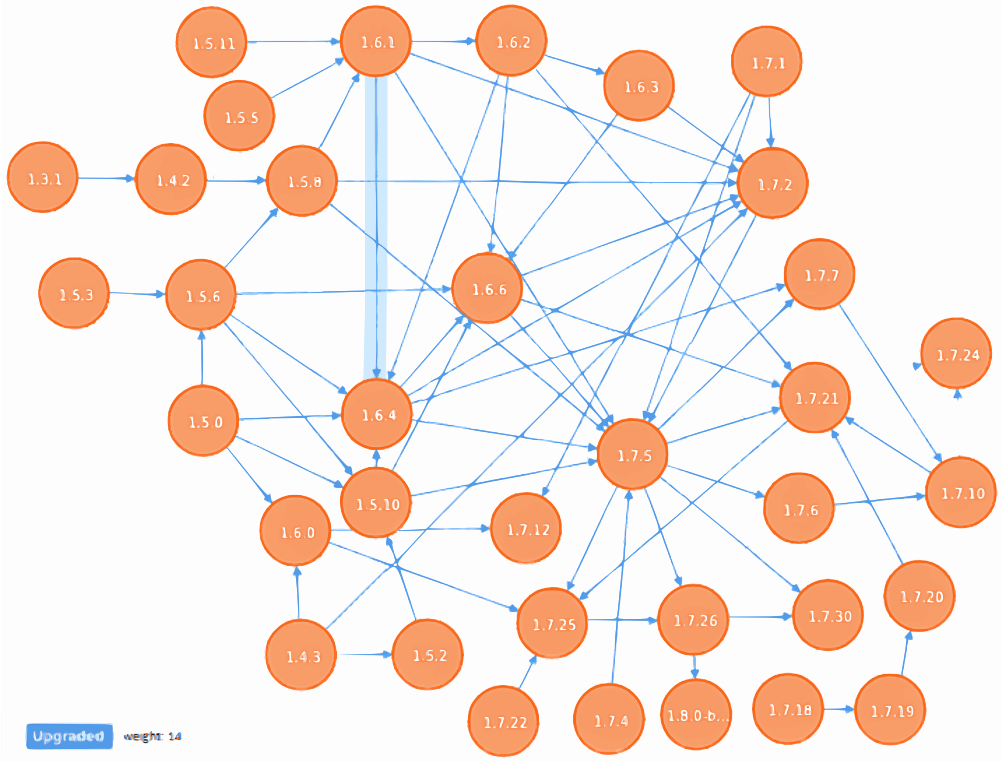


Figure 6.4: Migration graph of the *slf4j* library.

the version *1.6.1* to *1.6.4* is selected, and 14 clients (see the details on the bottom) have performed such a migration.

Plan Calculator. Such a component plays a key role in the project. Given a library to be upgraded, the starting version, and the target one, *Plan Calculator* retrieves the k shortest paths by using the well-founded *Yen's K-shortest paths algorithm* [153] which has been embedded into the NEO4J library. As a default heuristic implemented in EvoPlan, the component retrieves all the possible paths that maximize the popularity of the steps that can be performed to do the wanted upgrade. Thus, *Plan Calculator* employs the aforementioned weights which represent the popularity as a criteria for the shortest path algorithm.

By considering the graph shown in Figure 6.4, there are several possibilities to upgrade *slf4j* from version *1.5.8* to *1.7.25*. By taking into account the available weights, EvoPlan can recommend the ranked list depicted in Figure 6.5. The first path in the list suggests to follow the steps *1.6.1*, *1.6.4*, and *1.7.5* to reach the final version considered in the example, i.e., *1.7.25*.[11] Such a plan is the one that is performed most by other projects, which rely on

[11] It is worth noting that the popularity values are inversely proportional to the popularity of the corresponding upgrade plans. In the example shown in Figure 6.5 the most popular upgrade is the one with popularity value 0.898.

Figure 6.5: List of *k*-shortest paths for *slf4j*.

Table 6.2: Issues information extracted for *commons-io*.

Version	Open Issues	Closed Issues	Delta
1.0	14	33	19
1.3.2	150	420	270
1.4	87	408	321
2.0	5	10	5
2.0.1	133	457	324
2.1	129	516	387
2.2	67	999	932
2.3	5	20	15
2.4	939	3,283	2,344
2.5	64	918	854
2.6	64	548	484

slf4j and that have already operated the wanted library migration. Thus, such a path is more frequent than directly updating the library to the newest version.

Issues Miner. Issues play an important role in project development. For instance, by solving issues, developers contribute to the identification of bugs as well as the enhancement of software quality through feature requests [154]. In the scope of this work, we exploit issues as criteria for ordering upgrade plans. In particular, we rely on the availability of issues that have been opened and closed due to upgrades of given third-party libraries.

The *Issue Miner* component is built to aggregate and filter raw issues data gathered in the early stage of the process shown in Figure 6.2. However, due to the internal construction of NEO4J, we cannot directly embed this data as a weight on the migration graph's edges. Thus, as shown in Section 6.1.2, we collect the number of open and closed issues considering a specific time window, i.e., two months starting from the introduction of a certain TPL in the project. Then, this component filters and aggregates the issues data related by using Pandas,

a widely-used Python library for data mining [155]. For instance, Table 6.2 shows the mined issues related to the *commons-io* library. In particular, for each version of the library, the number of issues that have been opened and closed by all the analysed clients since they have migrated to that library version is shown. EvoPlan can employ the extracted data to enable a ranking function based on GitHub issues as discussed in the next section.

Issues Miner works as a stand-alone component, thus it does not impact on the time required by the overall process. In this way, we have an additional source of information that can be used later in the process as a supplementary criterion to choose the ultimate upgrade plan from the ranked list produced by the *Plan Calculator* component.

Plan Ranker. In the final phase, the k-paths produced by the *Plan Calculator* are rearranged according to the information about issues. For every path, we count the average value of opened/closed issues. A large value means that a certain path potentially requires less integration effort since there are more closed issues than the opened ones [154], i.e., issues have been tackled/solved rather than being left untouched.

Thus, the aim is to order the plans produced by *Plan Calculator* according to the retrieved issues: among the most popular plans we will propose those with the highest issue values.

Table 6.3: An example of the ranking results.

Proposed Path	**Pop. Value**	**Issues Value**
1.5.8, 1.6.1, 1.6.4, 1.6.6, 1.7.5, 1.7.25	1.446	57
1.5.8, 1.6.1, 1.6.4, 1.7.5, 1.7.25	0.898	58
1.5.8, 1.7.5, 1.7.25	1.0	58
1.5.8, 1.6.1, 1.7.5, 1.7.25	1.0	61
1.5.8, 1.6.1, 1.6.4, 1.7.2, 1.7.5, 1.7.25	1.238	58

Table 6.3 shows an example of the ranking process. There are two highlighted paths, the gray row corresponds to the best result according to the plan popularity only. In fact, the gray highlighted plan is the one with lower popularity value. Meanwhile, the orange row is recommended according to the issues criteria (in this case, the higher the issue value, the better). The path that should be selected is the orange one because it represents the one characterized by the highest activity in terms of opened and closed issue, among the most popular ones. In this way, EvoPlan is able to recommend an upgrade plan to migrate from the initial version to the desired one by learning from the experience of other projects which have already performed similar migrations.

6.1.3 Evaluation

To the best of our knowledge, there are no replication packages and reusable tools related to the approaches outlined in Section 6.1.1 that we could use to compare EvoPlan with them. As a result, it is not possible to compare EvoPlan with any baselines. Thus, we have to conduct an evaluation of our proposed approach on a real dataset.

Research questions. To study the performance of EvoPlan, we consider the following research questions:

- **RQ**$_1$: *How effective is EvoPlan in terms of prediction accuracy?* To answer this question, we conduct experiments following the ten-fold cross-validation methodology [156] on a dataset considering real migration data collected from GitHub. Moreover, we compute *Precision*, *Recall*, and *F-measure* by comparing the recommendation outcomes with real migrations as stored in GitHub;
- **RQ**$_2$: *Is there any correlation between the GitHub issues and the popularity of a certain migration path?*
  We analyze how the number of opened and closed issues could affect the migration process. To this end, we compute three different statistical coefficients to detect if there exists any correlation among the available data.
- **RQ**$_3$: *Is EvoPlan able to provide consistent recommendations in reasonable time?* Besides the recommended migration steps, we are interested in measuring the time of the overall process, including the graph building phase. This aims at ascertaining the feasibility of our approach in practice.

Overall process. As depicted in Figure 6.6, we perform experiments using the ten-fold cross-validation methodology on a well-founded dataset coming from an existing work [73]. Given the whole list of ≈11,000 projects, we download the entire dataset using the *Crawler* component. Then, the dataset is split into testing and ground truth projects, i.e., 10% and 90% of the entire set, respectively, by each round of the process. This means that in each round we generate a new migration graph by using the actual 90% portion. Given a single testing project, the *Analyzing commits* phase is conducted to capture the actual upgrade path followed by the repository, as stated in Section 6.1.2. To build the ground-truth graph, i.e., the real migration in GitHub, we consider projects not included in the testing ones and calculate every possible upgrade plan for each TPLs.

To aim for a reliable evaluation, we select the starting and the end version of a certain TPL from the actual plan of a testing project. The pair is used to feed the *Plan Calculator*

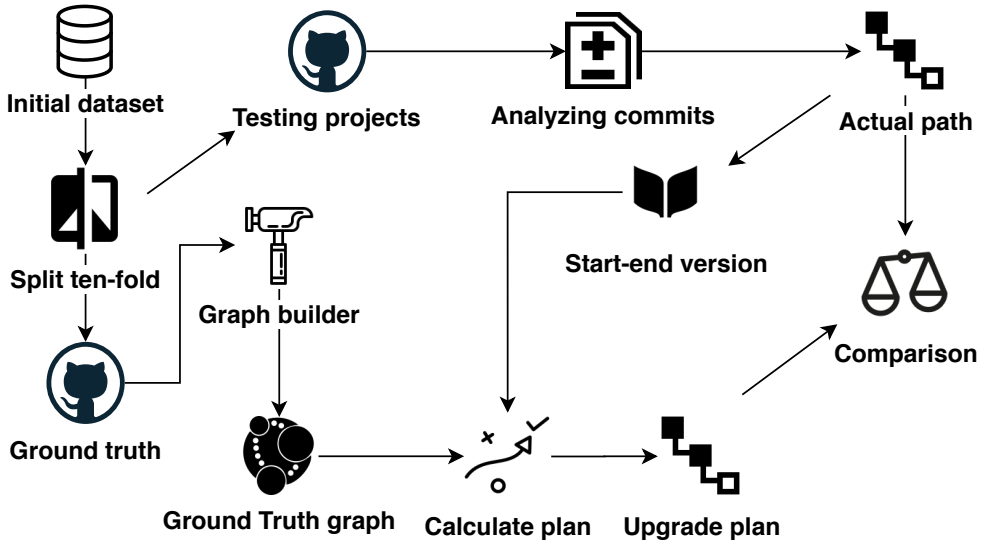


Figure 6.6: The evaluation process.

component which in turn retrieves the proposed plan. In this respect, by following the two paths we are able to compute the metrics to assess the overall performance, namely precision, recall, and F-measure.

Data collection. We make use of an existing dataset which has been curated by a recent study available on GitHub.[12] The rationale behind this selection is the quality of the repositories which were collected by applying different filters, i.e., removing duplicates, including projects with at least one *pom.xml* file, and crawling only well-maintained and mature projects. Table 6.4 summarizes the number of projects and *pom.xml* files. The dataset consists of 10,952 GitHub repositories, nevertheless we were able to download only 9,517 of them, as some have been deleted or moved. Starting from these projects, we got a total number of 27,129 *pom.xml* files. Among them, we selected only those that did not induce the creation of empty elements by the *Data Extractor* component while analyzing *logs* and *diffs* as shown in Figure 6.3. The filtering process resulted in 13,204 *pom.xml* files. The training set is used to create a migration graph to avoid any possible bias. For each round, we tested 420 projects, and 3,821 projects are used to build the graph.

Table 6.5 summarizes the set of libraries in the dataset, obtained by employing the *Crawler* module (cf. Section 6.1.2). There are seven popular libraries,[13] i.e., *junit*, *httpclient*, *slf4j*, *log4j*, *commons-io*, *guava*, and *commons-lang3*. Among others, *junit* has the largest

[12] https://bit.ly/2Opd1GH
[13] https://mvnrepository.com/popular

Table 6.4: Statistics of the dataset.

Total number of projects	10,952
Number of downloaded projects	9,517
Total number of *pom.xml* files	27,129
Number of screened *pom.xml* files	13,204

number of migrations, i.e., 2,972. Concerning the number of versions, *slf4j* has 71 different versions, being the densest library. Meanwhile, *commons-lang3* is associated with the smallest number of migrations, i.e., 162, and *commons-io* is the sparsest library with only 16 versions. The last column shows the number of versions that we could exploit to get the issues. The difference means that no issues data was available for the whole versions dataset.

Table 6.5: Number of migrations and versions.

Library	**# migrations**	**# versions**	**# issue vers.**
junit	2,972	30	19
httpclient	218	53	35
slf4j	209	71	26
log4j	229	42	19
commons-io	186	16	11
guava	627	70	34
commons-lang3	162	16	13

Metrics. Given a migration path retrieved by EvoPlan, we compare it with the real migration path extracted from a testing project. To this end, we employ *Precision*, *Recall*, and *F-measure* (or F_1-score) widely used in the Information Retrieval domain to assess the performance prediction of a system. In the first place, we rely on the following definitions:

- A *true positive* corresponds to the case when the recommended path matches with the actual path extracted from the testing projects; *TP* is the total number of true positives;
- A *false positive* means that the recommended upgrade plan is not present in the ground-truth paths; *FP* is the total number of false positives;
- A *false negative* is the migration steps that should be present in the suggested plan but they are not; *FN* is the total number of false negatives.

Considering such definitions, the aforementioned metrics are computed as follows:

$$P = \frac{TP}{TP + FP} \tag{6.1}$$

$$R = \frac{TP}{TP + FN} \tag{6.2}$$

$$F - measure = \frac{2 \times P \times R}{P + R} \tag{6.3}$$

Rank correlation: We consider the following coefficients:

- *Kendall's tau* [156] measures the strength of dependence between two variables. It is a non-parametric test, i.e., it is based on either being distribution-free or having a specified distribution but with the distribution's parameters unspecified.
- *Pearson's correlation* [157] is the most widely used correlation statistic to measure the degree of the relationship between linearly related variables. In particular, this coefficient is suitable when it is possible to draw a regression line between the points of the available data.
- *Spearman's correlation* [112] is a non-parametric test that is used to measure the degree of association between two variables. Differently from Pearson's coefficient, Spearman's correlation index performs better in cases of monotonic relationships.

All the considered coefficients assume values in the range [-1,+1], i.e., from perfect negative correlation to perfect positive correlation. The value 0 indicates that between two variables there is no correlation.

In the next section, we explain in detail the experimental results obtained through the evaluation.

6.1.4 Results

We report and analyze the obtained results by answering the research questions introduced in the previous section.

RQ$_1$: *How effective is EvoPlan in terms of prediction accuracy?* Table 6.6 reports the average results obtained from the cross-validation evaluation. EvoPlan achieves the maximum precision for *commons-io*, i.e., 0.90 in all the rounds. The tool also gets a high precision for *junit*, i.e., 0.88. Meanwhile, the smallest precision, i.e., 0.58 is seen by *httpclient*. Concerning

recall, EvoPlan obtains a value of 0.94 and 0.96 for the *junit* and *commons-io* libraries, respectively. In contrast, the tool achieves the worst recall value with *httpclient*, i.e., 0.64. Overall by considering the F-Measure score, we see that EvoPlan gets the best and the worst performance by *commons-io* and *httpclient*, respectively.

Table 6.6: Precision, Recall, and F-Measure considering popularity.

Library	**Precision**	**Recall**	**F-measure**
junit	0.88	0.94	0.91
httpclient	0.58	0.64	0.61
slf4j-api	0.65	0.74	0.69
log4j	0.88	0.93	0.91
commons-io	**0.90**	**0.96**	**0.94**
guava	0.60	0.73	0.65
commons-lang3	0.66	0.67	0.65

Altogether, we see that there is a substantial difference between the performance obtained by EvoPlan for different libraries. We suppose that this happens due to the availability of the training data. In particular, by carefully investigating each library used in the evaluation, we see that the libraries with the worst results in terms of performance have a few migrations that we can extract from the *pom.xml* on average (cf. Table 6.5). For instance, there are 162 and 209 migrations associated with *commons-lang3* and *slf4j-api*, respectively and EvoPlan gets a low performance on these libraries. Meanwhile, there are 2,972 migrations for *junit* and EvoPlan gets high precision, recall, and F_1 for this library. It means that less data can negatively affect the final recommendations.

Another factor that can influence the conducted evaluation could be the number of versions involved in an upgrade for each library i.e., the availability of fewer versions dramatically reduce the migration-related information. This hypothesis is confirmed by the observed values for *log4j* and *junit* that bring better results with 39 and 40 analyzed versions respectively. However, there is an exception with *guava*, i.e., EvoPlan yields a mediocre result for the library (F_1=0.65), though we considered 627 migration paths and 49 different versions. By examining the library, we realized that it has many versions employed in the Android domain as well as abandoned versions. Thus, we attribute the reduction in performance to the lack of decent training data.

Answer to RQ_1. EvoPlan is capable of predicting the correct upgrade plan given a real-world migration dataset. Although for some libraries we witness a reduction in the overall performances, the main reason can be found in the lack of migration paths in the original dataset.

RQ$_2$: *Is there any correlation between the GitHub issues and the popularity of a certain migration path?* To answer this question we measure the correlation among observed data, i.e., the number of clients that perform a certain migration step and the issues delta considering the time interval described in Section 6.1.2.

The number of clients performing migration is defined with the term *popularity* as described in Section 6.1.2. Meanwhile, as its name suggests, the *delta* is the difference between the number of closed issues and the number of open ones. It assumes a positive value when the number of closed issues is greater than the opened ones. In contrast, negative values are observed when open issues exceed the number of closed ones. In other words, deltas characterizes migration steps in terms of closed issues.

The results of the three indexes are shown in Table 6.7. As we can see, all the metrics show a positive correlation between the number of clients that perform a certain migration and the corresponding delta issues. In particular, Kendall's tau τ is equal to 0.458, while Spearman's rank ρ reaches the value of 0.616. The maximum correlation is seen by Pearson's coefficient, i.e., r = 0.707.

The strong correlation suggests that given a library, the more clients perform a migration on its versions, the more issues are solved. As it has been shown in a recent work [154], the act of solving issues allows developers to identify bugs and improve code, as well as enhance software quality. Summing up, having a large number of migrated clients can be interpreted as a sign of maturity, i.e., the evolution among different versions attracts attention by developers.

Answer to RQ$_2$. There is a significant correlation between the upgrade plan popularity and the number of closed issues. This implies that plans to be given highest priority should be those that have the majority of issues solved during the migration.

RQ$_3$: *Is EvoPlan able to provide consistent recommendations in reasonable time?* We measured the average time required for running experiments using a mainstream laptop with the following information: i5-8250U, 1.60GHz Processor, 16GB RAM, and Ubuntu 18.04 as the operating system. Table 6.8 summarizes the time for executing the corresponding phases.

Table 6.7: Correlation coefficients with a *p-value* $< 2.2e{-}16$.

Metric	**Value**
Kendal's (τ)	0.458
Pearson (r)	0.707
Spearman (ρ)	0.616

Table 6.8: Execution time.

Phase	Time (seconds)
Graph building	15,120
Querying	0.11
Testing	145.44

The most time consuming phase is the creation of graph with 15,120 seconds, corresponding to 252 minutes. Meanwhile, the querying phase takes just 0.11 seconds to finish; the testing phase is a bit longer: 145.44 seconds. It is worth noting that the testing consists of the sub-operations that are performed in actual use, i.e., opening CSV files, extracting the actual plan, and calculating the shortest path. This means that we can get an upgrade plan in less than a second, which is acceptable considering the computational capability of the used laptop. This suggests that EvoPlan can be deployed in practice to suggest upgrade plans.

Answer to RQ$_3$. The creation of the migration graph is computationally expensive. However, it can be done offline, one time for the whole cycle. EvoPlan is able to deliver a single upgrade plan in a reasonable time window, making it usable in the field.

6.1.5 Threats to Validity

This section discusses possible threats that may affect the proposed approach. Threats to *internal validity* could come from the graph building process. In particular, the crawler can retrieve inaccurate information from *pom.xml* files or GitHub commits. To deal with this, we employed a similar mining technique used in some related studies presented in Section 6.1.1, i.e., Meditor, APIwave, aiming to minimize missing data. Another possible pitfall lies in downgrade migrations, i.e., a client that moves from a newer version to an older one. Moreover, there is an issue related to the migration performing time. In particular, we do not consider the library release time window as a criteria to filter the possible paths. This aspect will be taken into consideration in our future work.

Concerning *external validity*, the main threat is related to the generalizability of the obtained results. We try to mitigate the threat by considering only popular Java libraries. Nevertheless, EvoPlan relies on a flexible architecture that can be easily modified to incorporate more TPLs. Concerning the employed GitHub issues data, they are coarse-grained, i.e., we can have a huge number of issues that do not have a strong tie with the examined TPLs. We addressed this issue in the work by considering the ratio of the delta instead of absolute numbers.

Concerning the supported data sources, EvoPlan employs *Maven* and GitHub to mine migration histories and retrieve issues, respectively. Thus, currently, upgrade plans can be

recommended for projects that rely on these two technologies. However, the architecture of EvoPlan has been designed in a way that supporting additional data sources would mean operating localized extensions in the *Crawler*, *Data Extractor*, and *Issue Miner* components without compromising the validity of the whole architecture.

Finally, threats to *construct validity* concern the ten-fold cross-validation shown in Section 6.1.3. Even though this technique is used mostly in the machine learning domain, we mitigate any possible incorrect values by considering a different ground-truth graph for each evaluation round. Additionally, the usage of GitHub issues could be seen as a possible threat. We mitigate this aspect by using such information as post-processing to reduce possible negative impacts on the recommended items, i.e., ranking the retrieved upgrade plans according to the total amount of issues.

6.2 DeepLib: Machine translation techniques to recommend the next version for TPLs

To keep their code up-to-date with the newest functionalities as well as bug fixes offered by third-party libraries, developers often need to replace an old version of third-party libraries (TPLs) with a newer one. However, choosing a suitable version for a library to be upgraded is complex and susceptible to error. So far, Dependabot is the only tool that supports library upgrades; however, it targets only security fixes and singularly analyzes libraries without considering the whole set of related libraries. In this work, we propose DeepLib as a practical approach to learn upgrades for third-party libraries that have been performed by similar clients. Such upgrades are considered safe, i.e., they do not trigger any conflict, since, in the training clients, the libraries already co-exist without causing any compatibility or dependency issues. In this way, the upgrades provided by DeepLib allow developers to maintain a harmonious relationship with other libraries [3]. By mining the development history of projects, we build migration matrices to train deep neural networks. Once being trained, the networks are then used to forecast the subsequent versions of the related libraries, exploiting the well-founded background related to the machine translation domain. As input, DeepLib accepts a set of library versions and returns a set of future versions to which developers should upgrade the libraries.

6.2.1 Motivations and background

To facilitate the presentation, hereafter, we consider the following terms:

– *library* or *dependency*: A software module which is developed by a third party, and provides tailored functionalities. A library evolves over the course of time by offering new functionalities or bug fixes [138, 158];

– *repository* or *client*: A software project that is hosted in OSS platforms, e.g., GitHub, Maven and that makes use of some third-party libraries;

To replace the constituent third-party libraries, the developer can either *(i)* migrate an existing library to another library with similar functionalities; or *(ii)* upgrade a library from an old version to a newer one. While the former has been intensively investigated [135], the latter remains largely unexplored. Thus, in the scope of this work, we focus on upgrading libraries that are used by the software project at hand.

We first describe a motivating example in Section 6.2.1.1, and then make an overview of Dependabot, which is highly related to the problem addressed in this work (Section 6.2.1.2). Afterwards, we briefly recall background related to long short-term memory recurrent neural networks (Section 6.2.1.3) and sequence-to-sequence learning (Section 6.2.1.4).

6.2.1.1 Motivating example

During the development cycle, with respect to library usage, a repository is normally updated by adopting a new version of libraries, even adding new (or removing deprecated) libraries. To better motivate our work, we consider in Table 6.9 a running example with maintainers working on the repository named *org.apache.hadoop:hadoop-auth*[14] which depends on a set of four libraries as follows:

- lib_1: *log4j:log4j*;
- lib_2: *org.slf4j:slf4j-log4j12*;
- lib_3: *org.apache.httpcomponents:httpclient*;
- lib_4: *commons-codec:commons-codec*.

In Table 6.9, the latest version of the *hadoop-auth* repository is *3.0.0-alpha3* (the green row), and let us assume that the maintainers would like to upgrade the used libraries. However, they do not know for sure which version should be used for the constituent libraries, i.e., all the cells are filled with a question mark (**?**). One may think of a simple heuristic that migrates a library to the next version, or the latest one. However, by carefully investigating

[14] https://bit.ly/2WP3ysS

Table 6.9: Migration path of the *org.apache.hadoop:hadoop-auth* repository.

Client version	**lib$_1$**	**lib$_2$**	**lib$_3$**	**lib$_4$**	**Timestamp**
2.0.2-alpha	1.2.17	1.6.1	0	1.4	2012-10-02T00:44:04
2.3.0	1.2.17	1.7.5	4.2.5	1.4	2014-02-11T13:55:58
2.4.1	0	1.7.5	4.2.5	1.4	2014-06-21T06:08:34
2.5.1	0	0	4.2.5	0	2014-09-05T23:05:15
2.6.0	1.2.17	0	4.3.1	0	2014-11-13T22:35:37
2.7.2	1.2.17	1.7.10	4.2.5	1.4	2016-01-14T21:32:14
3.0.0-alpha3	1.2.17	1.7.10	4.5.2	1.4	2017-05-26T20:39:35
*	?	?	?	?	

the table, we can see that such a heuristic does not work in every case. In particular, there are two additional possible changes that developers can perform on library dependencies: *(i)* removal of a library; and *(ii)* downgrade migration, as we explain as follows.

▷ **Removal of a library.** In the table, the versions of the repository are listed in chronological order, i.e., using their timestamp, that means when moving down the table, from the top to the bottom, we encounter newer versions of the repository. A cell with 0 implies that the library in the column is not included by the repository version represented in the row. It is worth noting thatthe presence of a library is subject to change from version to version. For instance, lib$_1$ has been used by version *2.0.0-alpha*, *2.0.2-alpha*, and *2.3.0*. When the repository is upgraded from *2.3.0* to *2.4.1*, lib$_1$ is removed. However, the library is then re-introduced when moving from *2.5.1* to *2.6.0*. In this respect, we see that the ability to recommend a 0 is also useful.

▷ **Downgrade migrations.** We can see that the upgrading is not always done upward, i.e., moving the library to a higher version, since there are also backward migrations. For instance, when the client moves from *2.6.0* to *2.7.2*, lib$_3$ is downgraded from version *4.3.1* to *4.2.5*. However, the library is then updated to version *4.5.2* by client *3.0.0-alpha3*. This motivates us to perform an investigation on more libraries to see if downgrade migrations are just a special case, or they are commonplace. We crawled the migration history of 26 libraries from the Maven Central Dependency Graph [126]. For each library, the migration history of all of its clients was analyzed. Afterwards, we counted the number of clients that, at certain point in their history updates, migrate to an older version of the library under investigation. Table 6.10 shows the result of our study where we report: $\mathbf{C}_1$: The number of clients that migrate downward (sorted in descending order); $\mathbf{C}_2$: The most downgrading version; and $\mathbf{C}_3$: The number of clients that migrated downward with the version in Column $\mathbf{C}_2$.

Among the mined libraries, *org.slf4j:slf4j-api* is the library with the largest number of clients with downward migrations. In particular, 5,717 clients contain libraries being upgraded back to an older version. For the *org.slf4j:slf4j-api* library, *1.7.25* is the version with

Table 6.10: Libraries with clients performing downgrade migrations.

Library	C_1	C_2	C_3
org.slf4j:slf4j-api	5,717	1.7.25	2,034
com.fasterxml.jackson.core:jackson-databind	2,939	2.9.5	521
com.google.guava	2,802	21	541
org.apache.commons:commons-lang3	2,485	3:3.6	861
org.scala-lang:scala-library	2,153	2.11.12	321
org.slf4j:slf4j-log4j12	1,728	1.7.12	220
commons-io:commons-io	1,599	2.6	877
org.apache.httpcomponents:httpclient	1,151	4.5.5	426
commons-codec:commons-codec	946	1.11	640
ch.qos.logback:logback-classic	945	1.2.3	185
log4j:log4j	910	1.2.17	667
joda-time:joda-time	873	2.9.9	469
junit:junit	580	4.12	305
commons-logging:commons-logging	499	1.2	310
org.clojure:clojure	349	1.3.0	244
commons-lang:commons-lang	322	2.6	230
com.google.code.gson	303	2.8.2	51
com.google.code.findbugs:jsr305	290	3.0.2	125
org.springframework:spring-test	289	3.2.17	25
org.projectlombok:lombok	261	1.16.20	86
org.mockito:mockito-core	99	2.15.0	19
javax.servlet:javax.servlet-api	94	3.1.0	63
org.assertj:assertj-core	80	3.9.1	23
org.testng:testng	53	6.9.10	9
org.scalatest:scalatest	42	3.0.4	7
javax.servlet:servlet-api	30	2.5	17

most downgrading upgrades, i.e., 2,034 clients. Considering the other libraries in Table 6.10 as a whole, it is evident that backward migration is considerably popular. In other words, simply migrating a library to the next version, or the latest one is not always a solution. In practice, we need to perform both backward and downward upgrading.

In fact, to select the right version of each library, developers need to read the documentations very carefully and be informed of the internal changes. They also seek help in Q&A forums like Stack Overflow for probable solutions. Take as an example, with respect to the *org.apache.httpcomponents:httpclient* library (cf. Table 6.9), a developer creates a post[15] in Stack Overflow to ask for support concerning the upgrading of the current version *4.2.5*. Unfortunately, there is no proper solution to the raised issue.

According to an empirical work [73], systems are less likely to upgrade their library dependencies, with 81.5% of systems remaining with a popular older versions. In fact, developers cite upgrading as a practice that requires extra effort and added responsibility.

[15] https://bit.ly/3A8RN2s

Therefore, an automatic mechanism to recommend the future version of a library, or even for the whole set of libraries, is highly desirable, aiming to reduce the burden related to library upgrade.

Motivation 1. The upgrade of a library version is a complex task and it cannot be done just by moving to the next, or the latest version of the library.

6.2.1.2 GitHub Dependabot

This section gives an overview of the GitHub Dependabot approach [159], which aims at addressing the problem of upgrading TPLs used by existing GitHub repositories. In particular, Dependabot focuses on automating security updates of vulnerable dependencies. On a regular basis, Dependabot checks if new dependency versions are available. If yes, it informs developers and directly sends pull requests to the repository under investigation to update the corresponding dependency manifest with the new versions.

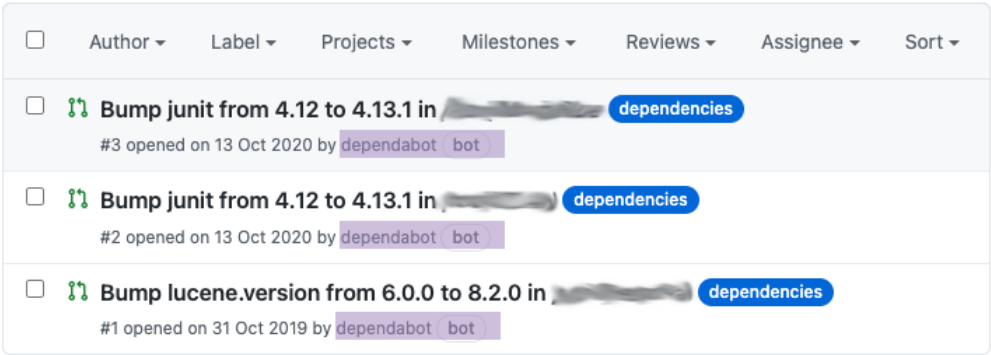


Figure 6.7: GitHub Dependabot pull requests (names are blurred due to privacy).

For instance, Figure 6.7 shows Dependabot in action. It suggests possible upgrades to solve some vulnerabilities of the used *lucene-core*, *log4j*, and *junit* libraries. However, it is important to remark that GitHub Dependabot tends to recommend the closest non-vulnerable version for each dependency used in the project. Thus, libraries are singularly analyzed and recommendations are given without considering the set of used dependencies as a whole. Therefore, the cases represented with the 0 values in Table 6.9 cannot be explicitly managed. It is worth notingthat the upgrades suggested by Dependabot are mainly based on vulnerability databases, e.g., the WhiteSource Vulnerability Database,[16] without taking into account ripple effects that might occur due to the co-existence of some additional dependencies in the given project.

[16] https://www.whitesourcesoftware.com/vulnerability-database/

Motivation 2. GitHub Dependabot is an initial attempt to recommend library upgrades. Nevertheless, it mainly analyzes libraries singularly and provides recommendations without considering the set of used dependencies as a whole. Thus, it deals with limited upgrading scenarios.

Altogether, we conclude that though there exists a tool to recommend upgrading of libraries, there is still a need for an automatic mechanism to support developers with migration, i.e., proper machinery to automatically choose a suitable version for their libraries considered as a whole. In this section, we present an approach being able to leverage the knowledge of already updated projects to assist the developer in choosing a suitable version for the used libraries. In the next subsections, we review two neural networks dealing with time series data as a base for further presentations.

6.2.1.3 Long short-term memory neural networks

Long short-term memory recurrent neural networks (LSTMs) have been developed to work with time-series and sequence data [142]. LSTMs can remove or add information thanks to their internal design, thereby retaining worthy/valuable information and forgetting useless information. Figure 6.8 depicts an LSTM cell, whose main modules are explained as follows.

In the figure, c_t and h_t are cell state and hidden state, respectively, which are propagated to the next cell. Given a cell, the output of the previous unit and the current input are fed as the input data. Considering $i_t = [h_{t-1}, x_t]$ as the concatenation of h_{t-1} (the hidden state vector from the previous time step) and x_t (the current input vector) then the following formulas are derived:

$$f_t = \sigma(W_f \cdot i_t + b_f) \tag{6.4}$$

$$u_t = \sigma(W_u \cdot i_t + b_u) \tag{6.5}$$

$$c_t = tanh(W_c \cdot i_t + b_c) \tag{6.6}$$

$$W_t = c_{t-1} \cdot f_t + N_t \cdot u_t \tag{6.7}$$

Where the *sigmoid* and *tanh* are defined as: $\sigma(x) = (1 + exp(-x))^{-1}$ and $\tanh(x) = 2 \cdot \sigma(2x) - 1$; $W_\times$ and $b_\times$ are the weight and bias matrices for different network entry, hidden state matrix. The sigmoid function is used to discard useless and retain useful information; Equations 6.4 and 6.5 are used to compute the forget and the update values, respectively. The same output h_t is ported as the hidden state to the next cell, and output of the current step [160]. The size of c_t is the number of hidden units in the LSTM cell.

The Softmax function is used as the activation function, rendering a set of real numbers to probabilities which sum to 1.0 [161]. Given C classes, and y_k is the output of the k^{th}

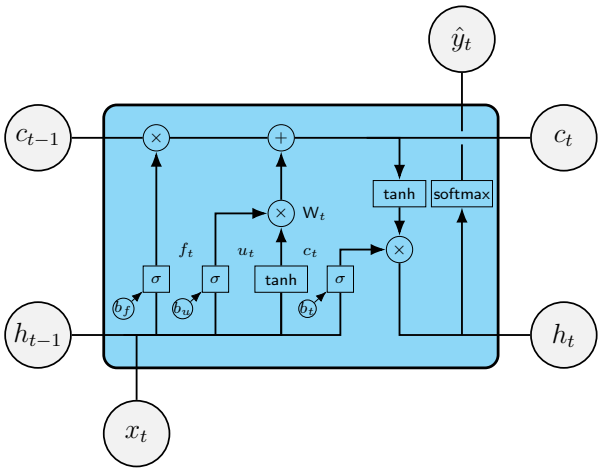


Figure 6.8: An LSTM cell (Reproduced [1]).

neuron, the final prediction is the class that gets the maximum probability, i.e., $\hat{y} = argmax\, p_k$, $k \in \{1,2,..,C\}$, where p_k is computed as: $p_k = \frac{exp(y_k)}{\sum_{k=1}^{C} exp(y_k)}$.

6.2.1.4 Sequence-to-Sequence learning

Encoder-Decoder LSTMs [143] are used to deal with sequence-to-sequence (seq2seq) prediction problems such as text summarization. An Encoder-Decoder LSTM transforms an input sequence $X = (x_1, x_2, \ldots, x_I)$ into an output sequence $Y = (y_1, y_2, \ldots, y_J)$.

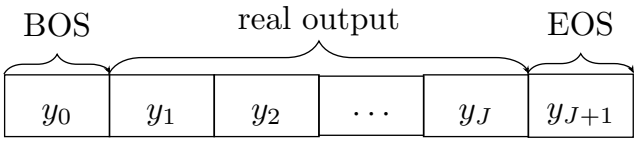


Figure 6.9: Output sequence.

Figure 6.9 depicts an output sequence, where y_0 and y_{J+1} represent the beginning (BOS) and the end of sequence (EOS), respectively. In this respect, generating Y when X is given as input boils down to computing the conditional probability $P_\theta(Y|X)$ below:

$$P_\theta(Y|X) = \prod_{j=1}^{J+1} P_\theta(y_j|Y_{<j}, X) \tag{6.8}$$

where $P_\theta(y_j|Y_{<j}, X)$ is the probability of generating the j^{th} of the output y_j, given $Y_{<j}$ and X. A seq2seq process involves two phases: *(i)* generating a fixed size vector z from X, i.e., $z = f(X)$; and *(ii)* generating Y from z.

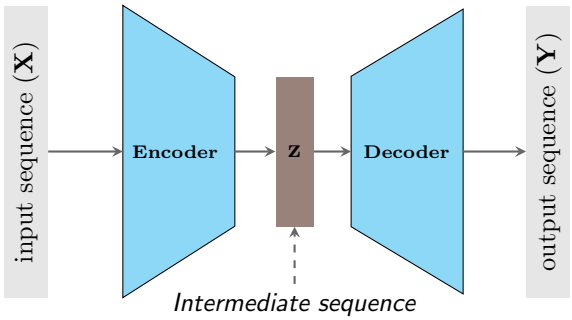


Figure 6.10: The Encoder-Decoder architecture (Reproduced [2]).

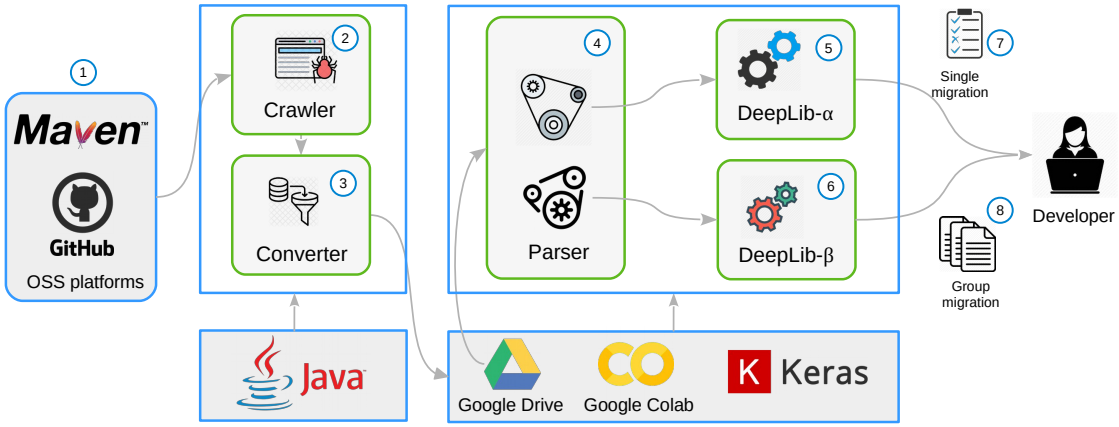


Figure 6.11: The DeepLib architecture [3].

Correspondingly, as shown in Figure 6.10 an Encoder-Decoder LSTM consists of the following components:

- *Encoder*: This is a stack of LSTM cells and it accepts X as input to generate a fixed size vector z.
- *Intermediate sequence*: z is the resulting vector obtained by encoding information contained in X.
- *Decoder*: The component consists of LSTM cells to produce the output sequence using the information fed by the encoder, i.e., z.

Encoder-Decoder LSTMs have gained big succeed in various domains [143] as they are highly suitable to generate sequences from sequences.

6.2.2 Architecture

In practical use, there are two levels of upgrade: *(i)* the *library level*; and *(ii)* the *source code level*. By the former, developers need to replace a library with a suitable version. In contrast,

by the latter, they have to change the affected source code to make it conform with the new library versions and the related APIs. In the scope of this work, we deal with the first type of upgrade, i.e., recommending a suitable version for libraries. The upgrade at the source code level is left as our future work. The conceived architecture is depicted in Figure 6.11. DeepLib has been implemented on top of the Keras framework[17] and trained using Google Colab.[18] Data is fetched from OSS platforms ①, e.g., GitHub and Maven with CRAWLER ②. The collected data is then aligned, sorted, and transformed into a suitable format to store in CSV files by CONVERTER ③. It is necessary to upload the data to Google Drive for further processing. The PARSER component ④ builds migration matrices for DeepLib-α ⑤ and DeepLib-β ⑥ to provide updates for a single library ⑦ and multiple libraries ⑧. The succeeding subsections explain the two modules in detail.

6.2.2.1 DeepLib-α: Recommending the next version for a single library

DeepLib-α has been developed using an LSTM to recommend a version for a third-party library. We mine development history of software projects to build a migration matrix, whose rows represent clients and columns represents their versions. To populate such a matrix, we analyze each software client and fill the correct version for all libraries, one by one. Starting from a set of OSS projects, we parse each client to build a migration matrix for DeepLib-α as shown in Figure 6.13(a). From the resulting matrix, we insert one more column on the right side. For each client, the last cell is filled with the version of the library by the next client.

To illustrate the transformation process, Figure 6.13(a) depicts the migration matrix for lib_1 for the *org.apache.hadoop:hadoop-auth* repository in Table 6.9. On the left, there is the original matrix, whose the corresponding migration matrix is depicted on the right. For instance, the first row contains the versions of the four libraries, i.e., *(1.2.15, 1.6.1, 0, 1.4)* while the last column is the future version of lib_1, i.e., *1.2.17*, which is actually the version of lib_1 by the next client (Version *2.0.2*). This can be interpreted as follows:

> "*Given that in the current client we use version* 1.2.15 *for* lib_1, 1.6.1 *for* lib_2, *no* lib_3, *and version* 1.4 *for* lib_4, *then in the next version of the client we should adopt* 1.2.17 *for* lib_1."

By repeating the same process, we can populate the migration matrices for other libraries. For the sake of clarity, only the matrix for lib_1 is shown in this section.

Since LSTMs only work with numbers, it is necessary to encode each library version using a unique number. Moreover, the σ and tanh functions (see Section 6.2.1) accept values

[17] https://keras.io/
[18] https://colab.research.google.com/

in the [0..1] range, we also need to normalize all the numbers to meet this requirement. The right most part of Figure 6.13(a) depicts the migration matrix after the encoding and normalizing phases. [19]

Figure 6.12 explains how DeepLib-α works, with respect to the example in Figure 6.13(a). The data to feed the system is a tuple of the form x_t=<lib_1^{v1}, lib_2^{v1}, lib_3^{v1}, lib_4^{v1}> and y_t=<lib_1^{v2}>, which captures the migration path of a client. DeepLib-α uses input features from recent events, i.e., X = $\{x_t\}, t \in T^p$ to forecast the future version of each libraries, Y = $\{y_t\}, t \in T^f$, where T^p and T^f are time in the past and the future, respectively. By each time step t, only one vector x_t is fed to the LSTM cell. For illustration purposes, we consider only four time steps, i.e., t_0, t_1, t_2, and t_3 and the input data is given below.

- At t_0: $x_0 = (0.500, 0.333, 0.000, 1.000)$, $y_0 = (1.000)$.
- At t_1: $x_1 = (1.000, 0.333, 0.000, 1.000)$, $y_1 = (1.000)$.
- At t_2: $x_2 = (1.000, 1.000, 0.333, 1.000)$, $y_2 = (0.000)$.
- At t_3: $x_3 = (0.000, 1.000, 0.333, 1.000)$, $y_3 = (0.000)$.

The same procedure can be done for other input entries to train DeepLib-α. Being based on the technique presented in Section 6.2.1, the tool uses the trained weights and biases to perform predictions for unknown input data.

[19] Matrix encoding and normalizing is conveniently done with the *LabelEncoder()* and *MinMaxScaler()* utilities embedded in Python.

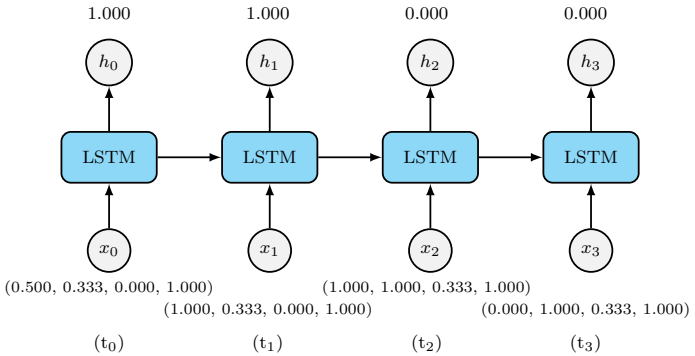
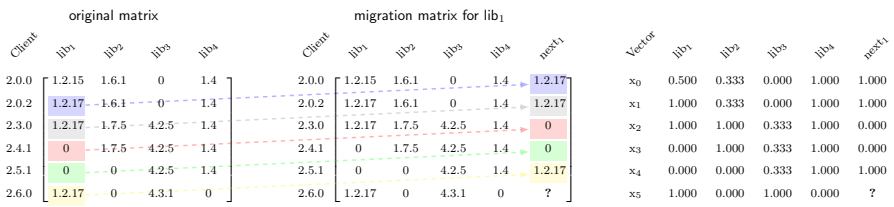


Figure 6.12: Architecture of DeepLib-α.



(a) Migration matrices for lib_1 in the example in Table 6.9

(b) Data parsed from the matrix

Figure 6.13: Migration matrices and input data for DeepLib-α.

Figure 6.14: Migration matrices and input data for DeepLib-β.

current versions (input features)

Client	lib_1	lib_2	lib_3	lib_4
2.0.0	1.2.15	1.6.1	0	1.4
2.0.2	1.2.17	1.6.1	0	1.4
2.3.0	1.2.17	1.7.5	4.2.5	1.4
2.4.1	0	1.7.5	4.2.5	1.4
2.5.1	0	0	4.2.5	1.4
2.6.0	1.2.17	0	4.3.1	0

future versions (labels)

Client	lib_1	lib_2	lib_3	lib_4
2.0.2	1.2.17	1.6.1	0	1.4
2.3.0	1.2.17	1.7.5	4.2.5	1.4
2.4.1	0	1.7.5	4.2.5	1.4
2.5.1	0	0	4.2.5	1.4
2.6.0	1.2.17	0	4.3.1	0
*	?	?	?	?

(a) Migration matrices for a set of libraries

Char.	Vector	1	.	2	5	[sp]	6	0	4	7	3
1	x_0	1	0	0	0	0	0	0	0	0	0
.	x_1	0	1	0	0	0	0	0	0	0	0
2	x_2	0	0	1	0	0	0	0	0	0	0
.	x_3	0	1	0	0	0	0	0	0	0	0
1	x_4	1	0	0	0	0	0	0	0	0	0
7	x_5	0	0	0	0	0	0	0	0	1	0
[sp]	x_6	0	0	0	0	1	0	0	0	0	0

(b) Data parsed from the "*1.2.17*[*sp*]" input string

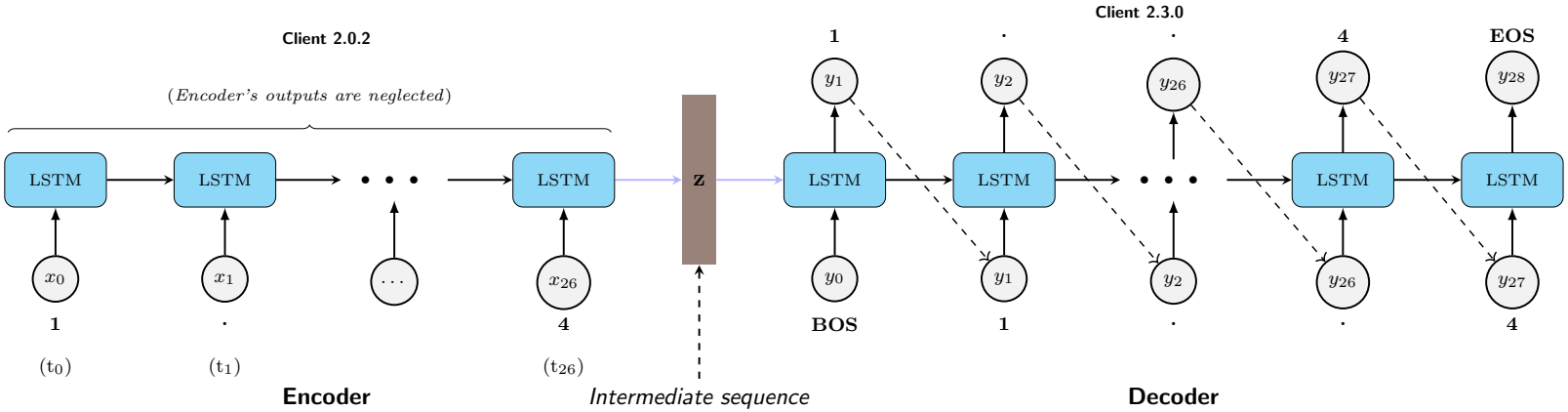


Figure 6.15: DeepLib-β: Training with input sequence $(1.2.17, 1.6.1, 0, 1.4)$ and output sequence $(1.2.17, 1.7.5, 4.2.5, 1.4)$.

6.2.2.2 DeepLib-β: Recommending the next version for the whole set of libraries

Though we can exploit DeepLib-α to recommend the next version for a set of libraries by computing the next version for each of them, one by one, in this section we introduce DeepLib-β to compute the next migration path for a set of libraries as a whole. This is helpful for developers who want to upgrade all libraries at once. In Figure 6.15(a), we illustrate how the migration matrix for the example in Table 6.9 is populated. On the left side, we depict the original matrix, each row corresponds to a version of the considered client (referred as client hereafter for simplicity). On the right-hand side, there is the resulting migration matrix, and each row is filled with all the library versions of the next client of the row in the left matrix. For instance, the future version of client *2.0.2* is *2.3.0*, i.e., marked with the blue frame, and this is expressed as:

> "*Given that in the current client we use* 1.2.17 *for* lib_1, 1.6.1 *for* lib_2, *no* lib_3, *and* 1.4 *for* lib_4, *then in the next version of the client we should adopt* 1.2.17 *for* lib_1, 1.7.5 *for* lib_2, 4.2.5 *for* lib_3, *and* 1.4 *for* lib_4."

The input and output sequences are represented as follows: X="*1.2.17[sp]1.6.1[sp]0[sp]1.4*" and Y="*[BOS]1.2.17[sp]1.7.5[sp]4.2.5[sp]1.4[EOS]*", where $[sp]$ is a space, [BOS] and [EOS] are special characters to signal the beginning and end of an output sequence (see Figure 6.9).

DeepLib-β is built based on the seq2seq learning model [143], converting a sequence into an output sequence of versions. To feed DeepLib-β, we transform the input data into vectors with entries in the [0..1] range. First, a corpus of all the characters used to form the library versions is curated. For instance, "*1.2.17*$[sp]$" contains the following characters: "*1*", ".", "*2*", "*7*", and "$[sp]$". Then each character is encoded using a one-hot vector whose length corresponds to the corpus's number of characters (Ω). For Figure 6.15(b), we get a corpus consisting of $\Omega = 10$ characters, and the maximum sequence length $\Theta = 27$. In this way, a sequence is represented as a 2D matrix of size ($\Theta \times \Omega$), each row corresponds to the one-hot vector of a character. A sequence with length smaller than Θ is padded with $[sp]$ to fill the gap. The same process is done in `Decoder` to form the output sequence.

For the sake of presentation, we illustrate how the parsing is done for the "*1.2.17*$[sp]$" input sequence in Figure 6.15(b). The first and second column represent the sequence and its vectors, respectively, each row corresponds to a one-hot vector. The remaining columns represent the characters.

Figure 6.15 shows how DeepLib-β works with respect to the example in Figure 6.15(a). `Encoder` consists of a stack of LSTM units to encode input sequences, and `Decoder` is also made of LSTM units and it decodes the input sequence. We feed a vector at every time step

to `Encoder`. Similarly, by `Decoder`, we introduce the label vectors singularly, moreover, the output of a time step is fed as input to the next one. It has been shown that, training in the reverse order of the input sentence brings a better prediction performance [143]. Therefore, in the scope of this work, every time there is an input sequence, we reverse it and feed as input to DeepLib-β. Following the paradigm in Figure 6.10, DeepLib-β learns from training data to predict the future versions for an input sequence of versions.

6.2.3 Evaluation

We evaluate DeepLib to study its capability to provide a developer with accurate recommendations featuring suitable migration steps.

6.2.3.1 Research questions

The evaluation was conducted to answer the following research questions:

- **RQ**$_1$: *How well can DeepLib-α recommend the next version for a single library?* We perform experiments to investigate to which extent DeepLib-α is able to recommend the next version for each single library. Such type of recommendation is desired when developers prefer to upgrade libraries one by one.
- **RQ**$_2$: *How well can DeepLib-β recommend the next version for a set of libraries?* Similar to **RQ**$_1$, we extend the evaluation and study if DeepLib-β can recommend the next version for a set of libraries. This is useful in practice, especially for developers who want to upgrade all libraries at once, instead of only one.
- **RQ**$_3$: *What contributes to an improvement in DeepLib's performance?* Our proposed tool is a data-driven approach, and its performance is heavily dependent on the input data. We investigate *when* the system cannot obtain a high prediction accuracy, i.e., it fails, and especially *why*. This aims to find a practical way to avoid the common pitfalls that adversely affect its recommendation capability in the field.

6.2.3.2 Data extraction

The Maven Central Repository[20] consists of a huge number of artifacts,[21] and includes additional features such as statistical reports, the list of most popular libraries, the list of

[20] https://mvnrepository.com/

[21] At the time of writing, there are more than 17 millions of artifacts in the Maven Central Repository.

Table 6.11: Summary of the datasets.

	Library	**Alias**	η_V	η_C	η_M
Dataset $\mathbf{D}_1$	*junit:junit*	L_{01}	29	101,541	2,073
	org.slf4j:slf4j-api	L_{02}	74	44,233	16,187
	org.scala-lang:scala-library	L_{03}	228	25,417	19,508
	com.google.guava:guava	L_{04}	90	24,532	8,921
	org.mockito:mockito-core	L_{05}	259	20,762	855
	com.android.support:appcompat-v7	L_{06}	59	19,772	1,194
	commons-io:commons-io	L_{07}	25	19,198	3,332
	ch.qos.logback:logback-classic	L_{08}	75	18,655	3,100
	org.commons:commons-lang3	L_{09}	18	17,224	3,915
	org.clojure:clojure	L_{10}	67	15,954	234
Dataset $\mathbf{D}_2$	*com.fasterxml.jackson.core*	L_{11}	120	15,891	9,022
	log4j:log4j	L_{12}	20	15,618	1,865
	org.slf4j:slf4j-log4j12	L_{13}	74	13,890	3,607
	org.scalatest:scalatest	L_{14}	18	13,216	10
	javax.servlet:javax.servlet-api	L_{15}	17	13,271	355
	com.google.code.gson:gson	L_{16}	35	13,300	1,562
	javax.servlet:servlet-api	L_{17}	17	13,271	427
	commons-lang:commons-lang	L_{18}	15	10,845	1,848
	org.apache.httpcomponents:httpclient	L_{19}	54	10,136	3,236
	org.slf4j:slf4j-simple	L_{20}	72	9,689	568
	org.springframework:spring-context	L_{21}	155	9,591	8,563
	org.assertj:assertj-core	L_{22}	44	9,628	495
	commons-logging:commons-logging	L_{23}	18	9,423	1,639
	org.projectlombok:lombok	L_{24}	40	9,742	1,021
	commons-codec:commons-codec	L_{25}	15	9.064	1,728
	org.testng:testng	L_{26}	78	8,941	440
	com.google.code.findbugs:jsr305	L_{27}	12	8,355	1,804
	org.springframework:spring-test	L_{28}	107	7,736	1,236
	joda-time:joda-time	L_{29}	38	7,565	3,187

dependencies for each artifact. To evaluate DeepLib, we rely on two datasets, named $\mathbf{D}_1$ and $\mathbf{D}_2$, collected from more than 1,000 public Maven repositories. $\mathbf{D}_1$ and $\mathbf{D}_2$ consist of migration history for the *top ten* and the *next top 19* popular libraries, respectively.

Given a set of libraries, we crawled all of their versions together with the list of clients and their corresponding release date. Moreover, we mined dependency links from a client to the used libraries with Maven Dependency Graph, a graph-based representation of the collected artifacts in Maven and their relationships. To generate a dependency graph from a set of libraries, we made use of an existing dataset [126] using MavenMiner [162].

Afterwards, we performed additional steps to remove unuseful clients by filtering the datasets with the following constraints: A client should *(i)* have more than one version; *(ii)* migrate at least one library among the considered libraries; and *(iii)* use at least four of the given libraries. This allows us to keep the resulting matrices not too sparse.

Compared to the clients in $\mathbf{D}_2$, those in $\mathbf{D}_1$ contain more upgrades from one library version to another. The use of $\mathbf{D}_1$ and $\mathbf{D}_2$ allows us to investigate if DeepLib can work well under different situations of the input data, i.e., if it still provides suitable upgrades even when the data is sparse. Table 6.11 reports the main characteristics of the datasets: each row features an input library with its name, the number of versions (η_V), the number of clients that use at least one version of the library (η_C), the number of clients that migrate from one library version to another (η_M). We obtained $35,300$ rows and $56,230$ rows for $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively.

6.2.3.3 Settings and metrics

▷ **Experimental settings.** The evaluation is done to study if our approach can recommend a future version for a library and a set of libraries. We opted for the ten-fold cross-validation technique [156], widely chosen to evaluate machine learning models. A dataset is split into $k = 10$ equal parts, so-called *folds*. One fold is used as testing data for each validation round, and the remaining $k-1$ folds are merged to create the training data. The testing fold represents projects that need upgrading recommendations, while the training folds correspond to the existing upgrades collected from real clients. In particular, by the D_1 dataset, there are $35,300$ upgrades and thus each testing fold consists of $35,300/10 = 3,530$ rows, while the training data is composed of $(35,300/10) \times 9 = 31,770$ rows (upgrades). By D_2, we have $56,230$ upgrades, and each testing fold has $56,230/10 = 5,623$ rows, while the training data is composed of $(56,230/10) \times 9 = 50,607$ rows (upgrades).

The evaluation simulates a real development scheme where *the system needs to provide the active projects with recommendations using the data from a set of available training projects.* Within the training data, we also used 80% and 20% of the data for training and validation, respectively. While the training phase is used to teach the models, the validation part is done to calibrate their hyperparameters. For each repository, the libraries and their versions corresponding to the older client are fed as input data, while the libraries and their versions corresponding to the newer client are used as label.

We take as an example of a project used for training as follows. Figure 6.16 shows the upgrading of the *com.hubspot:SingularityService*[22] repository from Version 0.4.2 to Version 0.6.1. The repository invokes four libraries, i.e., L_{02}, L_{04}, L_{08}, and L_{09}. On the left, we show the list of versions for the libraries of the older client numbered *0.4.2*. The remaining cells are filled with 0, indicating that the corresponding libraries are not present. On the right, there is the list of versions for the libraries of the newer client numbered *0.6.1*. Following the paradigm presented in Figure 6.12 and Figure 6.15, to train DeepLib, the left part is used

[22] https://bit.ly/2MnvnXn

as input data (query), and the right part is used as label. In practice, this means DeepLib is expected to provide recommendations consisting of versions as shown in the right part, given that it has been fed with the versions on the left part.

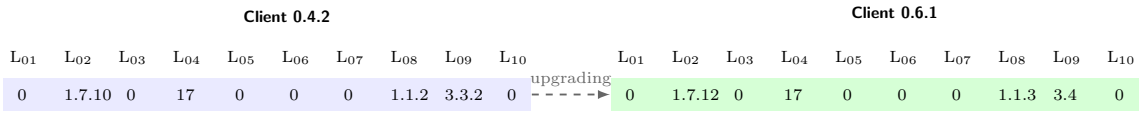


Figure 6.16: Upgrading *com.hubspot:SingularityService* from 0.4.2 to 0.6.1.

▷ **Ground-truth data.** For a testing client $TC_{current}$, all the library versions of its next client TC_{next} are saved as ground truth data. We use $TC_{current}$ to feed DeepLib, which returns a future version for each library of $TC_{current}$. We evaluate if the recommended versions match with the ground-truth data.

By checking the datasets, we see that clients sharing the same set of library versions can be updated with various paths. For example, the following three projects: *(1): org.skinny-framework:skinny-common_2.10*,[23] *(2): org.scalikejdbc:scalikejdbc-interpolation_2.10*, [24] and *(3): org.scalikejdbc:scalikejdbc-config_2.10*[25] depends on L_{02} and L_{03} (see Table 6.11), and their migration is shown in Figure 6.17.

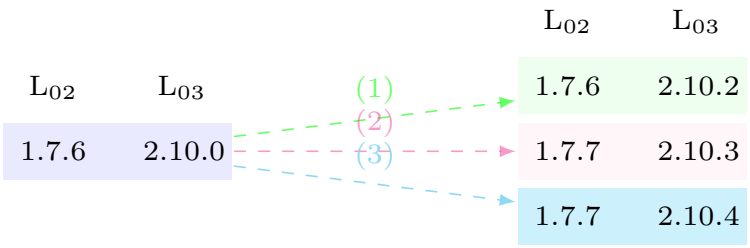


Figure 6.17: Clients with different migration patterns.

All of the starting clients have the same library versions, i.e., L_{02}: *1.7.6* and L_{03}: *2.10.0*. However, by their next client, each of the projects has a different migration pattern. For instance, while L_{02} is kept as *1.7.6* for *(1)*, it is upgraded to *1.7.7* in *(2)* and *(3)*. Similarly, the starting version of L_{03} is *2.10.0* and it is updated to three different versions, i.e., *2.10.2*, *2.10.3*, and *2.10.4* by *(1)*, *(2)* and *(3)*, respectively.

A recent empirical study [138] showed that a large number of libraries can be upgraded by at least one closest version, without causing any code changes. This happens since though

[23] https://bit.ly/3b5RhZn
[24] https://bit.ly/38fr3ln
[25] https://bit.ly/38Xw9lk

TPLs provide several APIs, developers normally make use a small fraction of them. That means for L_{02}, replacing *1.7.6* with *1.7.7*, or for L_{03} substituting *2.10.2* with *2.10.3* or *2.10.4*, does not trigger any incompatibilities. Altogether, in our evaluation, we consider multiple ground-truth paths for a client. With respect to the example in Figure 6.17, all the migrations on the right are ground-truth data for the client on the left.

To validate the performance, we must cover all possible cases concerning the ground-truth data. In fact, by carefully checking the dataset, we see that projects with different updates account for a small fraction. In particular, only 2% and 4% of projects in $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively, have multiple migrations, and the others have single migration. In this respect, the prediction of the next version for a library must be unique, so as to avoid overwhelming developers.

▷ **Metrics.** We evaluate how well DeepLib recommends versions that eventually match with those stored in the ground-truth data. With DeepLib-α, we compute accuracy according to each library (Acc_{lib}): The metric measures the ratio of clients with correct predictions (δ) to the total number of clients (n). While for DeepLib-β, we compute accuracy by each project (Acc_{pro}), i.e., the ratio of the number of correctly predicted versions (Δ) to the total number of libraries ($\mathbb{L}$) as follows.

$$Acc_{lib} = \frac{\delta}{n} \qquad (6.9) \qquad\qquad Acc_{pro} = \frac{\Delta}{\mathbb{L}} \qquad (6.10)$$

Besides accuracy, we compute correlation efficients using the Spearman ρ and the Kendall τ, and measure the effect size with Cliff's delta [134]: the larger the effect size is, the stronger is the relationship between the samples.

6.2.4 Results

We report a recommendation example provided by DeepLib in Section 6.2.4.1. Afterwards, the research questions are answered in Section 6.2.4.2, Section 6.2.4.3, and Section 6.2.4.4.

6.2.4.1 Explanatory example

To illustrate how DeepLib recommends upgrades to third-party libraries, we show in Figure 6.18 the recommendation results for the *com.hubspot:SingularityService*[26] that has been introduced in Figure 6.16. The repository invokes four libraries, i.e., L_{02}, L_{04}, L_{08}, and L_{09}.

The left side depicts the list of versions for the libraries of the older client *0.4.2*, and on the right, there is the list of versions for the libraries of the newer client *0.6.1*. The top row on the right depicts the real versions of all libraries for the next client *0.6.1* (the ground-truth

[26] https://bit.ly/2MnvnXn

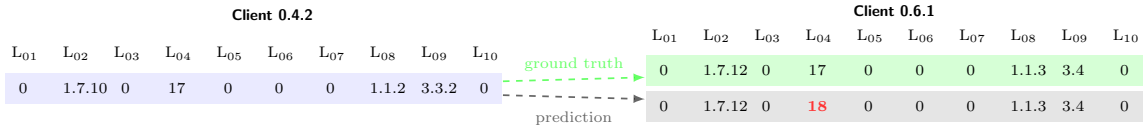


Figure 6.18: Recommendation for the *com.hubspot:SingularityService* repository.

data), and the bottom row represents the recommendations provided by DeepLib. Among the four libraries, three of them are upgraded to a new version. In particular, L_{02}: *1.7.10* $\rightarrow$ *1.7.12*, L_{08}: *1.1.2* $\rightarrow$ *1.1.3*, L_{09}: *3.3.2* $\rightarrow$ *3.4*.

The scenario is challenging as it requires a big upgrading step, i.e., changing almost all the constituent libraries at once. We select the example since it is a typical one in the evaluation. In particular, by carefully checking the given datasets, we realized that the majority of the clients perform big migrations. In this sense, we expect DeepLib to provide proper recommendations to assist developers in migrating their software clients, given that big migrations may make the prediction more challenging.

The second row of Figure 6.18 presents the versions suggested by DeepLib for Client *0.6.1*. As it can be seen, the tool recommends correct upgrading for three libraries, namely L_{02}, L_{08}, and L_{09}. It only mispredicts for L_{04}, by providing *18* instead of *17*, the correct one. Moreover, DeepLib accurately predicts all the zeros, i.e., the libraries that are not invoked. This seems to be trivial at fight sight, however as we pointed out before (see Section 6.2.2), recommending a zero makes sense, also considering the fact that wrongly suggesting a version rather than 0, when a 0 is actually needed, may make developers confused.

Summary. DeepLib provides relevant recommendations to the *explanatory repository*, given that a big step is required, i.e., upgrading at the same time by almost all the constituent libraries.

6.2.4.2 RQ_1: *How well can DeepLib-α recommend the next version for a single library?*

We performed experiments on both datasets and the prediction results for $\mathbf{D}_1$ and $\mathbf{D}_2$ are shown in Table 6.12. For each library, besides the accuracy for each fold from **F01** to **F10**, we also average out the scores to get the final accuracy, which is shown in the last column of the tables. Moreover, the cells with an accuracy smaller than 0.700 are marked using the light red color, signaling an inferior performance.

Table 6.12: RQ_1: $\mathbf{Acc}_{lib}$ obtained by DeepLib-α on $\mathbf{D}_1$ and $\mathbf{D}_2$.

		Cross validation										
	Library	**F01**	**F02**	**F03**	**F04**	**F05**	**F06**	**F07**	**F08**	**F09**	**F10**	**Average**
Dataset $\mathbf{D}_1$	L_{01}	0.975	0.956	0.988	0.991	0.954	0.954	0.962	0.965	0.986	0.970	0.970
	L_{02}	0.558	0.728	0.792	0.667	0.347	0.635	0.656	0.742	0.611	0.515	0.625
	L_{03}	0.748	0.824	0.741	0.856	0.908	0.944	0.902	0.670	0.552	0.669	0.781
	L_{04}	0.767	0.735	0.805	0.776	0.678	0.608	0.766	0.776	0.954	0.857	0.772
	L_{05}	0.961	0.989	0.988	0.972	0.976	0.623	0.952	0.994	0.966	0.974	0.939
	L_{06}	0.997	0.994	1.000	0.999	1.000	1.000	1.000	1.000	1.000	1.000	0.999
	L_{07}	0.952	0.981	0.970	0.969	0.945	0.947	0.952	0.960	0.990	0.961	0.963
	L_{08}	0.889	0.950	0.958	0.922	0.967	0.971	0.921	0.924	0.992	0.967	0.946
	L_{09}	0.937	0.952	0.967	0.966	0.960	0.852	0.966	0.900	0.984	0.952	0.944
	L_{10}	0.994	1.000	0.998	1.000	1.000	0.993	0.987	1.000	1.000	1.000	0.997
Dataset $\mathbf{D}_2$	L_{11}	0.340	0.227	0.524	0.683	0.801	0.109	0.648	0.701	0.641	0.766	0.544
	L_{12}	0.976	0.999	0.970	0.962	0.942	1.000	0.958	0.968	0.979	0.979	0.973
	L_{13}	0.964	0.987	0.845	0.828	0.815	1.000	0.825	0.798	0.989	0.818	0.887
	L_{14}	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.996	1.000	1.000	0.999
	L_{15}	0.996	0.996	0.989	0.993	0.990	1.000	0.976	0.989	0.976	0.991	0.990
	L_{16}	0.966	0.979	0.953	0.956	0.980	1.000	0.977	0.909	0.942	0.977	0.964
	L_{17}	0.998	0.998	0.993	0.995	0.979	1.000	0.986	0.996	0.989	0.995	0.993
	L_{18}	0.976	0.999	0.964	0.953	0.895	0.959	0.936	0.963	0.995	0.984	0.963
	L_{19}	0.878	0.986	0.810	0.848	0.793	0.306	0.804	0.778	0.809	0.757	0.777
	L_{20}	0.994	0.999	0.994	0.971	0.990	1.000	0.989	0.973	1.000	1.000	0.991
	L_{21}	0.822	0.930	0.706	0.822	0.577	0.214	0.580	0.483	0.243	0.365	0.574
	L_{22}	0.898	0.985	0.989	0.985	1.000	1.000	0.992	0.998	0.950	0.992	0.979
	L_{23}	0.976	1.000	0.983	0.963	0.906	0.959	0.969	0.955	0.964	0.951	0.962
	L_{24}	0.992	0.997	0.929	0.944	1.000	1.000	0.998	0.986	0.963	0.954	0.976
	L_{25}	0.959	0.990	0.952	0.942	0.902	0.978	0.936	0.956	0.997	0.978	0.959
	L_{26}	0.975	0.991	0.966	0.983	0.993	1.000	0.983	0.944	0.968	0.997	0.980
	L_{27}	0.866	0.995	0.969	0.959	0.988	1.000	0.974	0.911	1.000	0.999	0.966
	L_{28}	0.962	0.996	0.981	0.904	0.939	0.999	0.863	0.934	0.921	0.921	0.942
	L_{29}	0.902	0.966	0.854	0.917	0.863	0.924	0.848	0.863	0.949	0.947	0.903

For $\mathbf{D}_1$, there are ten libraries in total, and the results obtained by DeepLib-α for the dataset are shown in the upper part of Table 6.12. Overall, the table demonstrates that DeepLib-α can provide accurate predictions for almost all the libraries. For instance, with L_{01}, by all the testing rounds DeepLib always gets an accuracy larger than 0.90, and the average accuracy is 0.970. This also applies to other libraries, such as L_{05} or L_{07}. Especially, by L_{06} we see a maximum accuracy for most of the folds. It is our assumption that the quality of training data is the main contributing factor to the performance gain. We are going to validate this hypothesis in Section 6.2.4.4.

We analyze the results obtained by DeepLib-α on $\mathbf{D}_2$ in the lower part of Table 6.12. The table shows a similar outcome to that when running DeepLib-α on $\mathbf{D}_1$. By most of the libraries, DeepLib-α yields a good prediction performance, i.e., the accuracy is generally larger than 0.90. By L_{14}, DeepLib-α gets the maximum performance by nine among the ten folds.

However, besides the good predictions for most of the libraries of both datasets, it is evident that DeepLib-α suffers a setback by some of them. For instance, by L_{02}, DeepLib-α obtains a low accuracy for most of the folds: by only three among the ten folds, the tool gets an accuracy larger than 0.700, while by the remaining ones, it gains a lower accuracy. A mediocre performance is also seen by L_{11} and L_{21}, compared to the other libraries. This implies that DeepLib-α fails to retrieve accurate predictions for some libraries. We ascertain the cause in Section 6.2.4.4.

Answer to RQ_1. Being fed with proper training data, DeepLib is able to recommend the next version for a single library, obtaining a high prediction accuracy for the majority of the libraries.

6.2.4.3 RQ_2: *How well can DeepLib-β recommend the next version for a set of libraries?*

We report the final results for all the projects for each fold among **F01**-**F10** using boxplots. The accuracies obtained by DeepLib-β for both $\mathbf{D}_1$ and $\mathbf{D}_2$ are shown in Figure 6.19 and Figure 6.20, respectively. By examining the results, we encounter several cases similar to the one in Section 6.2.4.1, i.e., DeepLib-β recommends decent upgradings, also when big migration steps are required.

In Figure 6.19, apart from some outliers, by most of the folds for $\mathbf{D}_1$, we get an accuracy larger than 0.80. By many projects, DeepLib-β earns an accuracy of 1.00, suggesting that the tool correctly predicts all the library versions for these projects.

Figure 6.20 shows a similar outcome for $\mathbf{D}_2$, compared to $\mathbf{D}_1$. It is worth noting that by $\mathbf{D}_2$ there are 19 libraries, resulting in longer input and output sequences, and this should

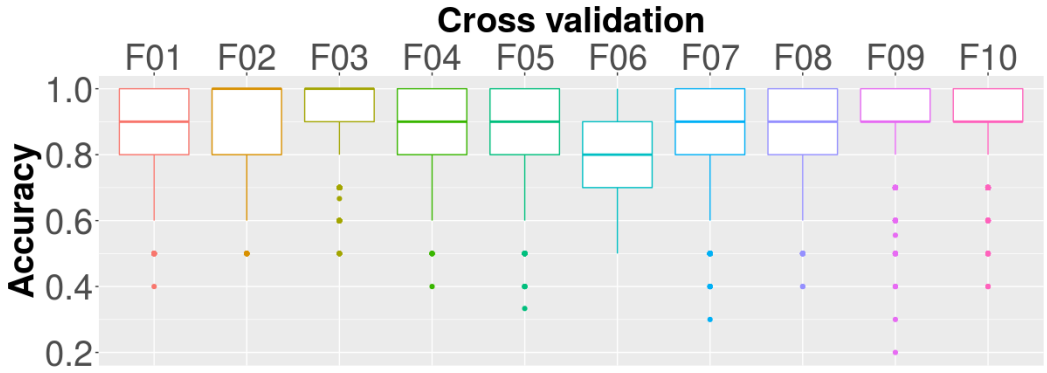


Figure 6.19: RQ$_2$: **Acc**$_{pro}$ obtained by DeepLib-β on $\mathbf{D}_1$.

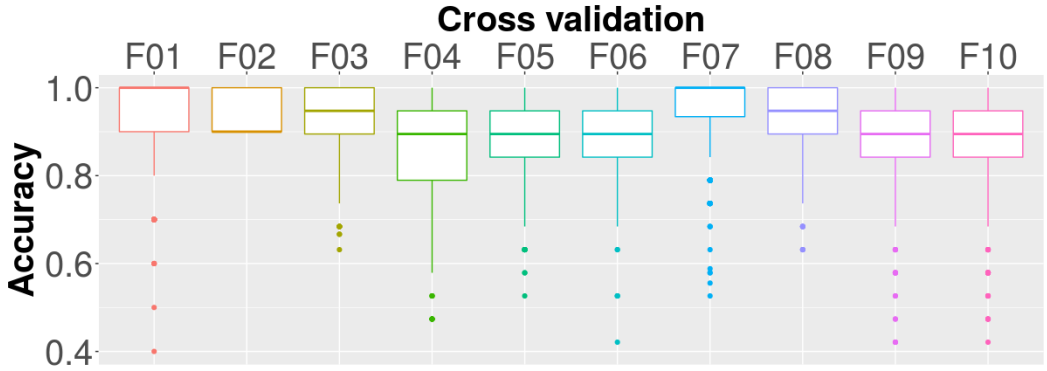


Figure 6.20: RQ$_2$: **Acc**$_{pro}$ obtained by DeepLib-β on $\mathbf{D}_2$.

make the prediction more challenging. It is evident that for almost all the folds, DeepLib-β gets an accuracy close to 1.0, and this is demonstrated by the narrow boxplots converging to the upper bound of the diagram.

Nevertheless, similar to the results in **RQ**$_1$, by both datasets we still witness projects with which DeepLib-β gets a low performance. As seen in the figures, some points lie further down the boxplots, corresponding to a mediocre performance. We attribute such a setback to the quality of the training data, and we find out the reason in the next research question.

Answer to RQ$_2$. On the given datasets, DeepLib-β successfully predicts the next version for libraries by most of the projects.

6.2.4.4 RQ$_3$: *What contributes to an improvement in DeepLib's performance?*

Through **RQ**$_1$ and **RQ**$_2$, we see that DeepLib obtains an encouraging result for most of the libraries as well as clients. Nevertheless, it *fails* in some certain cases. In Table 6.12, we encounter cells marked in red, corresponding to an accuracy lower than 0.700. Similarly, by

Figure 6.19 and Figure 6.20, there are outliers residing near the minimum whiskers, also suggesting a low accuracy. It is necessary to find out the rationale behind such a setback, as this helps reveal the pitfalls that one can avoid when deploying DeepLib.

According to Table 6.11, there are three variables: number of versions (η_V), number of clients (η_C), and number of migrations (η_M). We perform quantitative analyses to study the relationships between these variables and the average accuracy (the last column of Table 6.12). We compute correlation efficients using the Spearman ρ and the Kendall τ, and measure the effect size with Cliff's delta [134]. Table 6.13 reports the results obtained for the outcomes produced by DeepLib-α.

Table 6.13: RQ_3: Correlation coefficients and effect size, DeepLib-α.

Metric	**Score**	**Acc vs. η_V**	**Acc vs. η_C**	**Acc vs. η_M**
Spearman	ρ	-4.84×10^{-1}	-1.29×10^{-1}	-8.48×10^{-1}
	p-value	7.78×10^{-3}	5.04×10^{-1}	5.98×10^{-9}
Kendall	τ	-3.03×10^{-1}	-7.61×10^{-2}	-6.71×10^{-1}
	p-value	2.19×10^{-2}	5.61×10^{-1}	3.32×10^{-7}
Cliff's delta	–	1.0 (large)	1.0 (large)	1.0 (large)

As can be seen, there is a low correlation between accuracy and η_V, and this is enforced by both coefficients, i.e., $\rho = -4.84 \times 10^{-1}$ and $\tau = -3.03 \times 10^{-1}$. Moreover, the difference is statistically significant, i.e., p-value$= 7.78 \times 10^{-3}$ and 5.98×10^{-9}. The table also shows that the effect is large by the considered relationships, i.e., Cliff's delta is 1.0. This essentially means that the more versions a library has, the lower accuracy DeepLib-α obtains. In other words, having a large number of library versions negatively impacts on the prediction performance.

A similar trend is seen with the relationship between accuracy and η_M. In particular, $\rho = -8.48 \times 10^{-1}$ and $\tau = -6.71 \times 10^{-1}$, which means accuracy is disproportionate to the number of migrations. The difference is statistically significant and the effect size is large. Altogether, this suggests that it is more difficult for DeepLib to provide good recommendations for a library associated with a large number of migrations.

We cannot draw any concrete conclusions about the relationship between accuracy and η_C, as the p-value is larger than 0.05, although both ρ and τ are very small. This means by some libraries, having more clients is beneficial to predictions, while by some others, it is not. This is understandable since in principle having more training data is helpful [163], however, the prediction performance depends also on η_V and η_M, and as shown above, accuracy is greatly affected by these parameters.

We suppose that this happens due to the structure of the networks, i.e., the current weights are sufficient to memorize a certain amount of versions/migrations. However, if there are

more versions or migrations, the network fails to absorb all the patterns. Such a limitation can be overcome with deeper networks [164], i.e., by padding additional hidden units to DeepLib. To validate the hypothesis, we increased the number of units from 40 to 100 and reran the experiments on the libraries with which DeepLib-α gets a low accuracy by most of the folds, i.e., L_{02}, L_{11}, and L_{21}. As expected, we see a gain in accuracy by these libraries. For the sake of clarity, we report the change in accuracy with respect to Table 6.12 as follows: $Acc_{lib}(L_{02})$: $0.625 \rightarrow 0.632$, $Acc_{lib}(L_{11})$: $0.544 \rightarrow 0.559$, $Acc_{lib}(L_{21})$: $0.574 \rightarrow 0.613$. While by L_{02} and L_{11} there is a marginal increase, we can see a substantial gain by L_{21}. Similarly, by running DeepLib-β with more hidden units, compared to the results in Figure 6.19 and Figure 6.20, we got a minimum and maximum increase in Acc_{pro} of 5% and 18%, respectively.

The improvement suggests that one can enhance DeepLib's prediction performance for those libraries having a large number of versions/migrations by means of deeper networks, i.e., with more hidden units. In this way, we suppose that it is possible to further boost up the predictions for any libraries/clients in practical use, by choosing a suitable network configuration according to the input data.

Answer to RQ_3. DeepLib suffers a deficiency in performance on libraries with a large number of versions and/or migrations. However, depending on the input data, the system's performance can be enhanced with deeper networks.

6.2.5 Discussion

We discuss the practicality as well as possible extension of DeepLib in Section 6.2.5.1. The threats that might hamper the validity of our findings are presented in Section 6.2.5.2.

6.2.5.1 Applicability and future developments of DeepLib

A question that might arise at any time is: "*How can DeepLib be deployed in practice?*" As we see from Section 6.2.2, it is necessary to collect TPLs together with a set of clients associated with them. Afterwards, we build migration matrices to feed the recommendation engine. Once the collected data has been used to train the system, it can be removed to give place to new data, i.e., projects coming from OSS platforms. In other words, the knowledge learned from data is embedded in the internal weights and biases of the networks. This makes DeepLib a lightweight framework that can be easily deployed. We plan to integrate DeepLib into the Eclipse IDE, providing instant suggestions to developers while they are coding.

The performance of DeepLib is driven by the availability and quality of the training data. Thus we anticipate that it will fail if some library versions, e.g., newly released versions,

never occurred in the migration history. In this respect, DeepLib is supposed to learn better only when enough movement history is given a specific library version.

Recommendation of library upgrade is a complex problem, and this has been confirmed by various studies [136–138]. In fact, there are two levels of migration: *(i)* the *library level*; and *(ii)* the *source code level*. By the former, a developer needs to be provided with a suitable version of a library, whereas by the latter, she has to adapt the affected source code to make it work with the new library versions and related APIs. In the scope of this work, we tackle the first issue, i.e., recommending library migration. We expect that similar techniques employed to build DeepLib can be exploited to support source code migration. In particular, by collecting related projects, we can build matrices with old APIs as features and new APIs as the label. This enables us to deploy the same techniques used in DeepLib-α and DeepLib-β to predict suitable migration steps. This, however, requires us to parse source code to extract the API functions. Moreover, in the scope of this work, we consider only release time for the recommendation of updates. In practice, there are many artifacts having multiple major versions maintained at the same time. Thus, it would make sense to consider also semantic versioning, especially when upgrading on major versions and minor versions. We are going to tackle these issues in our future work.

According to the empirical evaluation, we see that the performance of DeepLib-α and DeepLib-β depends very much on the availability of the training data, i.e., among the two tools, there is no absolute winner in all cases. In particular, DeepLib-α is good at recommending the next version for a single library which consists of enough training data, i.e., when there are a large number of upgrades for the considered library. Meanwhile DeepLib-β outperforms DeepLib-α when there is a balance between the number of upgrades for all the related libraries. The rationale behind such a difference is as follows. In machine translation, an Encoder-Decoder LSTM can better predict the next sequence of words when there exists a frequent combination of words in the input sequence [2]. Since DeepLib-β is built on top of an Encoder-Decoder LSTM, it inherits the essential qualities of the original machine translation technique. Altogether, this suggests that in practice, the selection of a suitable recommendation strategy for libraries should be made according to the quality of the input data.

6.2.5.2 Threats to validity

Threats to *internal validity* are related to the confounding factors in our approach and evaluation that could have influenced the final results. A possible threat is that the datasets might not fully reflect real-world development scenarios as we were are able to consider only popular libraries. In practice, developers tend to work on a variety of libraries. To mitigate

the threats, we crawled a wide range of clients across several repositories. Still, we believe that considering data from other sources, such as GitHub, can help fully eliminate the threat.

The main threat to *external validity* concerns the generalizability of our findings. DeepLib has been evaluated on projects collected from Maven, since we have suitable software to fetch the data. We anticipate that our tool is also applicable to other platforms, as long as they support versioning. We plan to generalize DeepLib to data from GitHub in our future work. In fact, contributions in Maven come by strictly following a well-defined process, which is not the case for GitHub repositories, where projects are uploaded in an ad hoc manner. In this respect, it is necessary to carefully investigate the performance of DeepLib on projects curated from GitHub.

6.3 Conclusion

In this chapter we tackled the problem of supporting developers in updating their TPLs. Software systems heavily rely on third-party libraries (TPLs), which provide a wide range of functionalities that can be reused without the need to re-implement them from scratch. Even though TPLs evolve, e.g., to fix security holes or to increase the provided capabilities, most systems rarely update their dependencies. Developers consider TPL migration as a practice that can introduce extra efforts and responsibility.

We proposed EvoPlan, a novel approach to support the upgrading of TPLs by considering miscellaneous software artifacts. By envisioning different components, our tool is capable of extracting relevant migration data and encoding it in a flexible graph-based representation. Such a migration graph is used to retrieve multiple upgrade plans considering the popularity as the main rationale. They are eventually ranked by exploiting the GitHub issues data to possibly minimize the effort that is required by the developer to select one of the candidate upgrade plans. A feasibility study shows that the results are promising, with respect to both effectiveness and efficiency.

To reduce the burden related to the identification of the upgrades that need to be operated on the current system we proposed DeepLib, a novel approach to recommendation of the next version for the used TPLs by considering migration histories of several OSS projects. Our proposed tool is able to extract relevant migration data and encode it in matrices. Then, deep learning techniques are employed to provide recommendations that are relevant for the current configuration. Once being deployed, DeepLib allows developers to quickly select a suitable migration, by relying on the experience of other projects with similar sets of TPLs and thus, helping to eliminate any possible complexity concerning the technical details.

Chapter 7

Automated recommendation of GitHub topics

The open source software community makes a daily usage of open source repositories to contribute their work as well as to access to projects coming from other developers. GitHub is one of the most well-known platforms that aggregate these projects and render possible the exchange of knowledge among the users.

In order to aid information discovery and help developers identify projects that can be of their interest, GitHub introduced *topics*. They are words used to characterize projects, which thus can be annotated by means of lists of words that summarize projects' features. Thanks to the availability of *topics*, several applications are enabled, including the automated cataloguing of GitHub repositories [165], further than allowing developers to explore projects by type, technology, to name a few.

To the best of our knowledge, assigning the right topics to GitHub repositories is a crucial step that, if not properly done, may hamper their discoverability. In 2017, GitHub presented *repo-topix*, a topic suggestion tool essentially based on information retrieval techniques [74]. Although the mechanism works well so far and it has been fully integrated into GitHub, in our opinion there is still some room for improvement, e.g., in terms of the variety of the suggested topics, novel data analysis techniques, and the investigation of new recommendation strategies.

In this chapter, we conceive an approach to recommend topics to GitHub repositories. The tool extends the recommendation capabilities of MNBN [16], a state-of-the-art approach to non-featured topics by exploiting a collaborative filtering technique, which is widely used in the recommender system domain [93].

Figure 7.1: An example of GitHub repository and corresponding topics.

Structure of the chapter. Section 7.1 provides an overview of GitHub topics and the main open challenges in the given domain. The proposed approach is described in 7.2. Section 7.3 and Section 7.4 present the evaluation process, the results obtained, respectively. The chapter is finally concluded in Section 7.5.

7.1 Motivation and background

When using OSS repositories, users can be interested in acquiring knowledge from existing developed software projects [17]. However, especially in the case of large source code repositories, the potential benefits related to the availability of reusable projects might be missed if they cannot be suitably discovered. To mitigate such problems, in 2017 GitHub introduced the possibility of assigning tags[1] to projects with the final aim of increasing their discoverability. By means of such a feature, users can find and contribute to software projects by searching the topics of interest, affinity, and other relevant elements.

Figure 7.1 shows an example of repository stored in GitHub. It is a related to the *bootstrap* project,[2] and according to the given tags, it is a `css-framework`, which involves `javascript`, `html`, and `css` artifacts among others. GitHub maintains a curated list of projects, which are organized with respect to the list of *featured topics*,[3] which are meant to be the most popular and active topics. Thus, users can monitor the community's trend by consulting such a list.

Manually assigning topics can be an error-prone activity that can lead to wrongly specified tags. Over the last years, several attempts have been made to *classify* GitHub projects by automatically inferring appropriate topics. In the context of data mining, *classification* is one of the critical operations that are used to dig deep into available data for gaining knowledge and for identifying repetitive patterns [166].

Sharma *et al.* [165] present an approach based on *topic modeling* techniques to create categories of GitHub projects. Manual interventions are needed to refine initial sets of

[1] For the sake of presentation, the terms "tags" and "topics" are used interchangeably throughout the chapter
[2] https://getbootstrap.com/
[3] https://github.com/topics

categories, which are identified by an LDA-GA technique, that combines two algorithms: Latent Dirichlet Allocation (LDA) and Genetic Algorithm (GA) [167]. The approach proposed [165] is unsupervised, meaning that the categories of the catalogue being identified are not known ex-ante.

In a GitHub blog post [74], the author presents *repo-topix*, a tool to recommend topics for GitHub repositories. Such a tool combines standard NLP techniques to find an initial set of topics, by parsing the README files and the textual content of a repository e.g., the repository's description. Then, the results are weighted with the TF-IDF scheme and "bad" topics are removed exploiting a regression model. Using this refined list, repo-topix computes a custom version of the Jaccard distance to identify additional similar topics. To assess the quality of the framework, a rough evaluation was conducted based on ROUGE-1 metrics, an n-gram overlap metric that counts the number of overlapping units between the suggested topics and the repository description. Nevertheless, neither the tool nor the dataset is made available, rendering a comparison with the approach impossible.

GitHub is one of the most used development services that includes version control systems (i.e., git) plus social and collaborative features (e.g., bug tracking, contribution requests, task management, and Wikis). In 2020, GitHub counted more than 40 million users and over 100 million repositories. Because of this enormous amount of data, the availability of reusable projects might be compromised if they cannot be suitably discovered. In recent years, GitHub introduced a mechanism based on topics to explore repositories. The managed GitHub repositories are being continuously monitored and assigned with topics to improve their organization. Moreover, repositories are periodically analyzed to extract the most popular and active topics, i.e., *featured topics*.[4] Thus, users can observe the community's trend by consulting such a public list. In the beginning, this activity was entirely done by humans (i.e., project contributors) who label the repository according to their knowledge, feeling and belief. In the literature, there are a plenty of approaches that mine and exploit available data to analyze repositories. Nevertheless, a few of them cope with the topic recommendation task, which can be crucial in the project's development initial phase. Figure 7.2 shows an explanatory repository with related topics. By this simple snapshot, a GitHub user can figure out that the *SpaceshipGenerator*[5] repository makes use of *Blender-scripts* (i.e., a *python* 3d modeling library) for *procedural-generation* of *3d* spaceships from a random seed.

As previously mentioned, the MNBN approach takes as input README file(s) of a given repository to recommend the related featured topics, which can be assigned to it. To conceive MNBN [16], the authors have adopted standard techniques employed in the ML domain, i.e.,

[4] https://github.com/topics
[5] https://github.com/a1studmuffin/SpaceshipGenerator

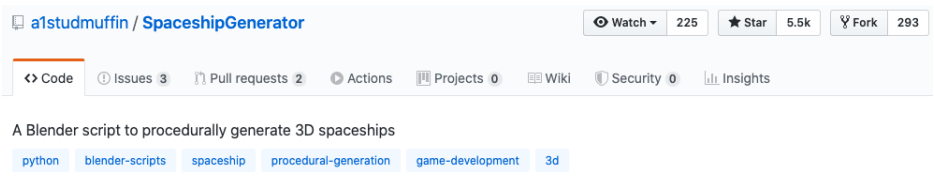


Figure 7.2: A GitHub repository with different topics.

textual engineering, feature extraction, and training phase. By relying on the multinomial probability distribution, the approach can extract relevant information from README file(s) and suggest a set of topics. Table 7.1 shows an example of the outcomes obtained by MNBN given the list of the actual repository topics.

Table 7.1: Example of the MNBN outcomes for the *SpaceshipGenerator* repository.

Actual Topics	**Recommended topics**
python, blender-scripts, spaceship, procedural-generation, game-development, 3d	shell, terminal, **3d**, opengl, **python**

Though MNBN works in practice, it suffers some limitations. First, the underlying model can recommend only featured topics that represent only a small set of all possible terms that can be potentially assigned to the analyzed repository.

For instance, *blender-scripts*[6] as well as *game-development* (which are not featured topics) could be recommended as possible topics because the project includes both *3d* and *python* topics. Thus, MNBN does not express all the concepts covered by a GitHub repository. As shown in the table, only two of the predicted topics match with the real ones. Moreover, in the case the repository already includes all suggested topics, MNBN is not able to recommend new ones. The second major limitation is the underlying structure needed for the training phase. MNBN requires a *balanced* dataset to deliver relevant items, i.e., each topic must have a similar number of README files. It is indeed challenging to satisfy such a constraint in practice, as topics are generally heterogeneous. Furthermore, repositories in GitHub are regularly updated with new topics, and thus, the training phase must take place several times to avoid outdated recommendations.

In recent years, many techniques have been conceived to predict users' interests by relying on the preferences collected from other users. Such techniques can be classified as *content-based* [168] where the relationships among items have been exploited to predict the most similar items, or *collaborative-filtering* [169] that calculates the missing ratings by taking into account the set of items rated by similar customers. There are two main types of collaborative-filtering recommendation: *user-based* [170] and *item-based* [171] techniques.

[6]blender is the most used python library to manipulate 3d objects. https://www.blender.org/

The former computes missing ratings by considering the ratings collected from similar users, whereas, the latter performs the same task by using the similarities among items [172].

In the following section, we show that the proposed collaborative-filtering approach can recommend missing topics for GitHub repositories. Moreover, we also demonstrate that it is possible to increase the accuracy of MNBN by combining it with the proposed technique.

7.2 The proposed approach

This section describes TopFilter, a recommender system that models the relationships among OSS projects using a graph representation, and exploits a collaborative filtering technique [93] to recommend relevant GitHub topics for the software repository under development. Collaborative filtering techniques have been conceived in the e-commerce domain to recommend products [8], based on the assumption that *"if users agree about the quality or relevance of some items, then they will likely agree about other items"* [93]. TopFilter works following the same line of reasoning to mine GitHub topics: *"if projects have some tags in common, then they will probably contain other relevant tags"* [17].

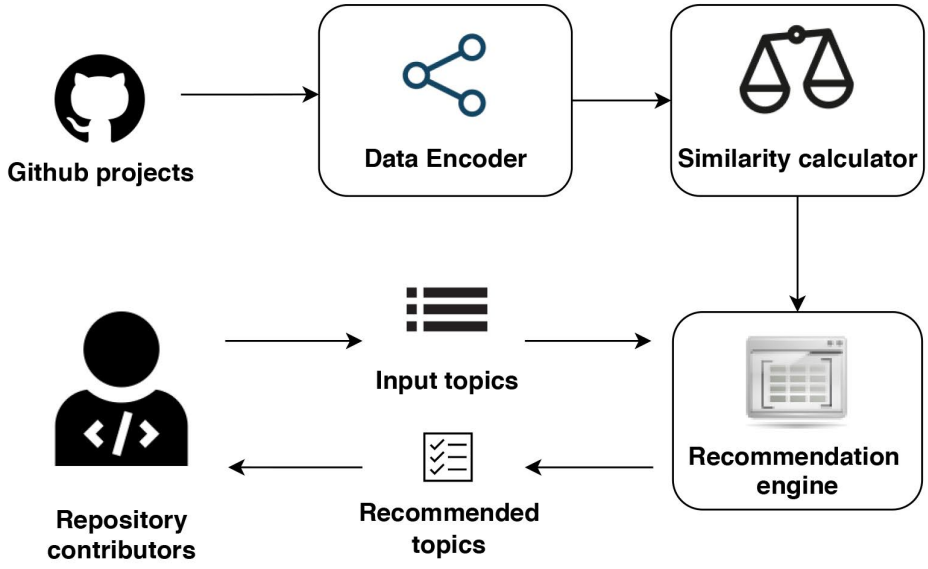


Figure 7.3: Overview of the TopFilter architecture.

In the following subsections, we describe two TopFilter configurations to recommend relevant topics by incorporating different types of input data. The first configuration exploits a collaborative-filtering technique, while the second one is a combination of both MNBN and TopFilter, aiming to improve the prediction performance of the original MNBN approach as initially proposed by its authors in their paper [16].

7.2.1 Recommending topics using a collaborative-filtering technique

Figure 7.3 depicts the architecture conceived to realize the TopFilter prototype. First, the `Data Encoder` component encodes GitHub repositories in a graph-based representation, and then `Similarity Calculator` computes similarities among all the managed projects. The `Recommendation Engine` component implements a *collaborative-filtering* technique to generate a ranked list of *top-N* topics which is eventually suggested to the developer to complement an initial list of topics given as input. We explain in detail the functionalities of each component as follows.

7.2.2 Data Encoder

The mutual relationships among GitHub repositories and topics are encoded using a *project-topic matrix* [171]: Each row represents a project, and each column corresponds to a topic. In this sense, a cell in the matrix is set to 1 if the project in the corresponding row is tagged with the topic in the corresponding column, otherwise the cell is set to 0.

To build the project-topic matrix, raw topics are first pre-processed using various Natural Language Processing (NLP) techniques, such as stemming, lemmatization, and stop words removal. This aims to remove possible syntactical duplicated terms, e.g., *document* and *documents*, which are frequent in GitHub. Afterwards, the final matrix is constructed by means of the topics obtained through the pre-processing phase.

For explanatory purposes, we consider a set of four projects $P = \{p_1, p_2, p_3, p_4\}$ together with a set of topics $L = \{t_1 = \textit{machine-learning}; t_2 = \textit{javascript}; t_3 = \textit{database}; t_4 = \textit{web}; t_5 = \textit{algorithm}, t_6 = \textit{algorithms}\}$. Moreover, the *project-topic inclusion* relationships is denoted as $\ni$. By parsing the projects, we discover the following inclusions: $p_1 \ni t_1, t_2, t_6$; $p_2 \ni t_1, t_3$; $p_3 \ni t_1, t_3, t_4, t_5$; $p_4 \ni t_1, t_2, t_4, t_5$. After the NLP normalization steps, the topics t_5 and t_6 collapse on the same term which is named as t_5. The final project-topic matrix is shown in Table 7.2.

Table 7.2: The *project-topic matrix* for the example.

	t_1	t_2	t_3	t_4	t_5
p_1	1	1	0	0	1
p_2	1	0	1	0	0
p_3	1	0	1	1	1
p_4	1	1	0	0	1

7.2.3 Similarity Calculator

This component relies on the previously encoded data to assess the similarity of given repositories. For explanatory reasons, we represent a set of projects and their topics in a graph, so as to calculate the similarities among the projects. For instance, Figure 7.4 depicts the graph-based representation of the project-topic matrix in Table 7.2. Two nodes in a graph are considered to be similar if they share the same neighbours by considering their edges. Such a technique has been successfully exploited by many studies to do the same task [173] in different domains.

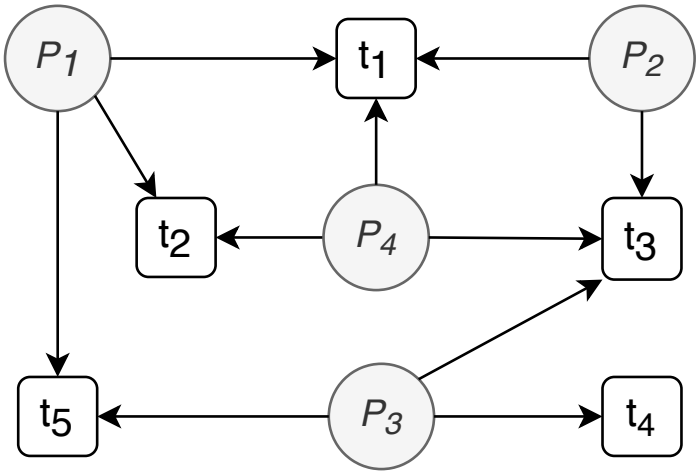


Figure 7.4: Graph representation for projects and topics.

Considering a project p that has a set of neighbor nodes (t_1, t_2,.., t_l), the features of p are represented by a vector $\phi = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_i being the weight of node t_i computed as the *term-frequency inverse document frequency* function as follows: $\phi_i = f_{t_i} \times log(|P| \times a_{t_i}^{-1})$, where f_{t_i} is the number of occurrences of t_i with respect to p, it can be either 0 and 1. $|P|$ is the total number of considered projects; a_{t_i} is the number of projects connecting to t_i via corresponding edges. Intuitively, the similarity between two projects p and q with their corresponding feature vectors $\phi = \{\phi_i\}_{i=1,..,l}$ and $\omega = \{\omega_j\}_{j=1,..,m}$ is computed as the cosine of the angle between the two vectors as given below.

$$sim(p,q) = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n} (\phi_t)^2} \times \sqrt{\sum_{t=1}^{n} (\omega_t)^2}} \tag{7.1}$$

where n is the cardinality of the union of topics by p and q.

7.2.4 Recommendation Engine

Given an input project p, and an initial set of related topics decided by the developer, the inclusion of additional topics can be predicted from the projects that are similar to p. In other words, TopFilter predicts topics' presence by means of those collected from the *top-k* similar projects using the following formula [17]:

$$r_{p,t} = \overline{r_p} + \frac{\sum_{q \in topsim(p)} (r_{q,t} - \overline{r_q}) \cdot sim(p,q)}{\sum_{q \in topsim(p)} sim(p,q)} \tag{7.2}$$

where $\overline{r_p}$ and $\overline{r_q}$ are the mean of the ratings of p and q, respectively; q belongs to the set of top-k most similar projects to p, denoted as $topsim(p)$; $sim(p,q)$ is the similarity between the active project and a similar project q, and it is computed using Equation 7.1.

7.2.5 Combined use of MNBN and TopFilter

Though MNBN works in practice, it suffers from some limitations. First, it can recommend only featured topics which account for a small fraction of all possible terms. Second, given that a repository already includes all suggested topics, MNBN is not able to recommend new ones. Moreover, the tool requires a *balanced* dataset, i.e., each topic must have a similar number of README files, and this is hard to come by in practice as topics are generally heterogeneous. With TopFilter, we attempt to improve MNBN by combining it with the collaborative-filtering technique presented in the previous subsection. The set of featured topics predicted by the MNBN model is used as input to feed TopFilter, which then runs the filtering process to deduce the inclusion of new topics.

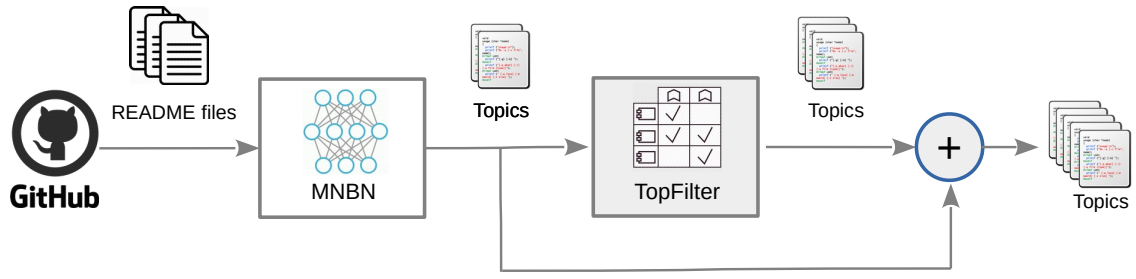


Figure 7.5: Overview of the combined approach.

Figure 7.5 depicts an overview of the combined use of MNBN and TopFilter: a list of featured topics computed by MNBN using README files is fed as input for TopFilter, which computes a list or ranked topics, including also non-featured ones. Finally, a list is generated by combining the topics computed by TopFilter with the top-N featured ones computed by MNBN, and recommended to developers.

We assume that TopFilter is beneficial in the following aspects:

- It is able to recommend non-featured topics, which are selected from similar repositories, independently from their nature;
- TopFilter recommends topics by iterating over refining steps: once we select some topics from the recommended ones, TopFilter can discover new topics using the selected ones as new input;
- MNBN does not provide additional recommendations given that the suggested topics are already included. As discussed in Section 7.4.1, TopFilter considerably improves the performance when more topics are incorporated as input.

In the following sections, we introduce the experiments conducted to evaluate the performance of TopFilter and its combined use with MNBN by means of different evaluation metrics.

7.3 Evaluation

In this section, we report on the evaluation process conducted to study the performance of TopFilter. Section 7.3.1 presents the research questions we want to answer by means of the performed experiments. Section 7.3.2 gives an informative description of the datasets exploited in the evaluation. Section 7.3.3 and Section 7.3.4 describe the evaluation metrics and process, respectively. To facilitate future research, we made available the TopFilter tool together with the related data in a GitHub repository.[7]

7.3.1 Research questions

The following research questions are addressed to study the performance of the proposed approach:

- RQ_1: *Which TopFilter configuration yields the best performance?* We investigate different configurations of TopFilter i.e., the number of input topics as well as the number of neighbour projects is varied, to find the best configuration.
- RQ_2: *To what extent can the accuracy of MNBN be improved by means of TopFilter?* We are interested in understanding if our proposed approach can be used to improve the accuracy of the original MNBN.

[7] https://github.com/MDEGroup/TopFilter

7.3.2 Data extraction

To study our proposed approach, we reused the same dataset employed to evaluate MNBN and made available by the authors of that technique in a replication package.[8]

In particular, to investigate TopFilter's prediction performances, we populated five different datasets starting from the original MNBN corpus by varying the *cut-off* value t [16], i.e., the maximum frequency of the topic distribution (it is detailed in Section 7.3.4). In this way, we removed infrequent elements from the dataset to analyze their impacts on the overall recommendation phase. We firstly filtered the initial set of topics using their frequencies counted on the entire GitHub dataset. Afterwards, we removed irrelevant topics to reduce probable noise during the prediction phase.

We developed a filter by means of tailored Python scripts and applied it to the initial dataset. As a GitHub user can manually specify a topic list for a repository, many of them can contain infrequent or improper terms, i.e., the name of the author, duplicated values, or terms that rarely appear, to name a few. On one hand, imposing such a preprocessing phase reduces the number of repositories to analyze as well as topics to recommend. On the other hand, we improve the overall quality of recommendation by removing *"bad"* terms. This pruning phase can be done offline and does not affect the time required for the recommendation process.

Table 7.3 summarizes the main features of the datasets, i.e., D_1, D_5, D_{10}, D_{15}, and D_{20}, corresponding to different cut-off values $t = \{1, 5, 10, 15, 20\}$. *Avg. topics* is the average number of topics that the repositories include; *Avg. freq. topics* is the average frequency of the topics in the dataset.

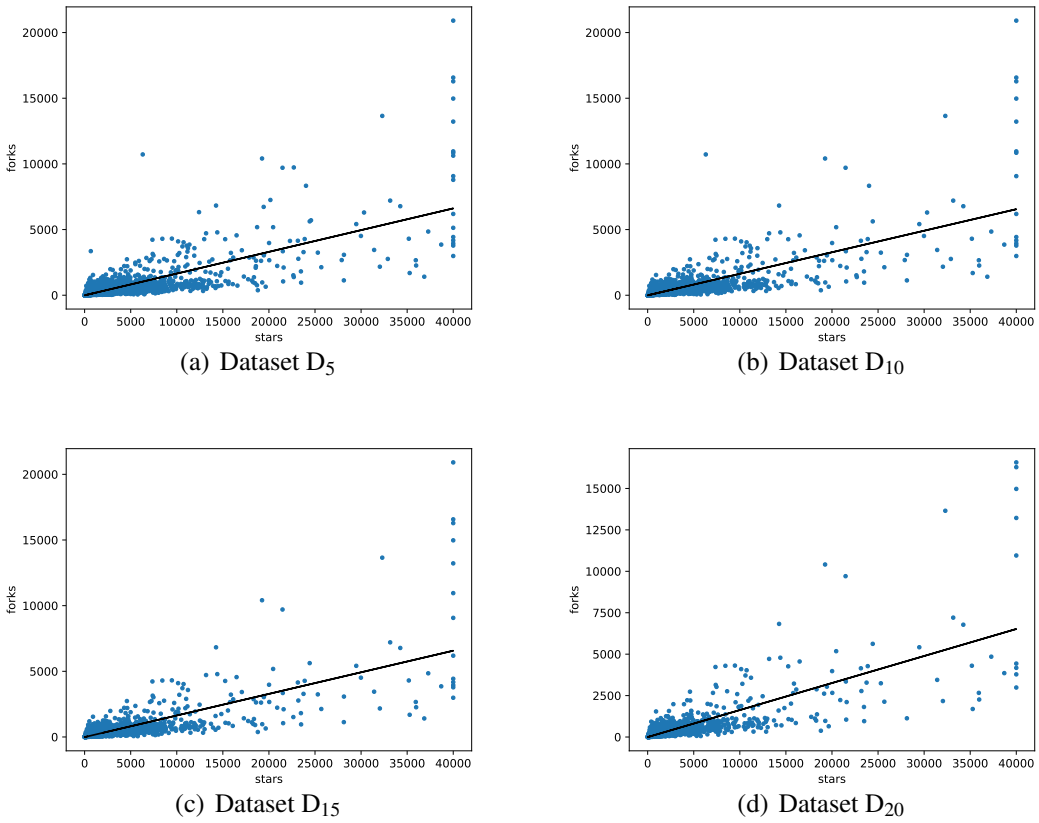
Table 7.3: Datasets.

	D_1	D_5	D_{10}	D_{15}	D_{20}
Number of repos	6,253	3,884	2,897	2,273	1,806
Number of topics	15,743	1,989	964	634	456
Avg. topics	9.9	8.4	8.0	7.8	7.7
Avg. freq. topics	3.9	16.5	24.1	28.1	30.5

As discussed in Section 7.4, removing infrequent topics improves the overall quality of the considered datasets: the collaborative filtering provides better prediction performance when there is enough data, i.e., topics in the training set. Once infrequent topics have been removed, all the repositories that contain less then five topics are filtered out from the dataset, as they contain little information to enable the collaborative filtering prediction. In particular,

[8] https://github.com/MDEGroup/MNB_TopicRecommendation/

Figure 7.6: A summary of the number of forks and stars for the datasets.



(a) Dataset D_5 (b) Dataset D_{10}

(c) Dataset D_{15} (d) Dataset D_{20}

we remove around 2,300 repositories by increasing the cut-off value from 1 to 5. It means that the excluded repositories in Dataset D_5 are tagged with topics that rarely appear in the considered repositories. This finding is strengthened by the number of topics, which dramatically decreases to 1,989 when $t = 5$. We stop at $t = 20$ and consider the D_{20} dataset as the best one according to our metrics. Additionally, we observe that repositories are tagged by 9.9 and 7.7 topics on average for $t = 1$ and $t = 20$, respectively. This demonstrates that a large number of topics are not beneficial to the discoverability of a project.

From the list of projects under analysis, we exploited the GitHub API to retrieve their number of stars and forks. A summary of the retrieved data is plotted in Fig. 7.6. Forking is a means to contribute to the original repositories, furthermore, there is a strong correlation between forks and stars [108], as it can be further observed in Fig. 7.6. A project with a high number of forks means that it garners attention from the developers community. A repository with a large number of forks can be considered as a well-maintained and well-received project. Whereas, since commits also have an influence on source code [109], the number of commits is also a good indicator of how a project has been developed. As can be seen, most

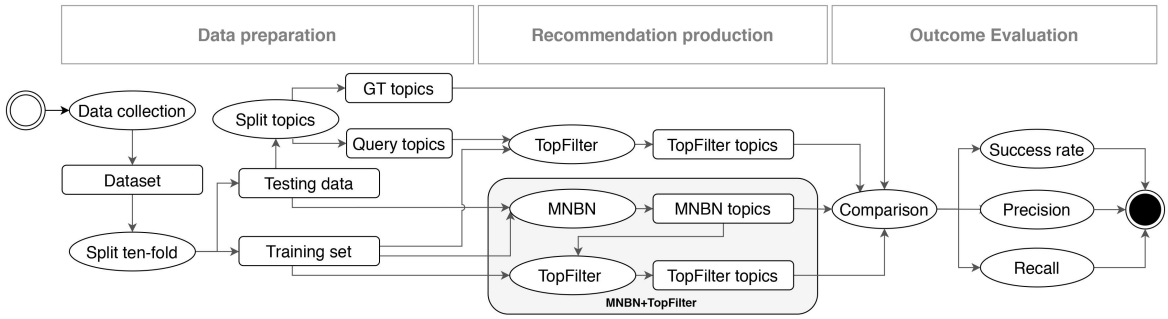


Figure 7.7: Evaluation Process.

of the repositories possess a number of stars and forks less than 5,000. A small fraction of them have more than 30,000 forks and 10,000 stars. In this respect, we see that the datasets exhibit a wide variety of quality in terms of the number of forks and stars.

7.3.3 Metrics

In the scope of this work, *success rate accuracy*, and *catalog coverage* are used to study the TopFilter performance as they have been widely exploited by related research [174]. First, we introduce the following notations as a base for further presentation:

- t is the frequency cut-off value of input topics, i.e., all topics that occur less than t times are removed from the dataset;
- τ is the number of topics that TopFilter takes as input;
- N is the cut-off value for the recommended ranked list of topic;
- k corresponds to the number of top-similar neighbor projects TopFilter considers to predict suggested topics;
- *GT(p)* is defined as a half of the extracted topics for a testing project p using as ground-truth data;
- $REC_N(p)$ is the top-N suggested topics sorted in a descending order;
- a recommended topic rt to a repository p is marked as a *match* if $rt \in REC(p)$;
- $match_N(p)$ is the set of items in $REC_N(p)$ that match with those in *GT(p)* for p;
- T is the set of all the available topics.

By means of the notations, the success rate, accuracy and coverage metrics are defined as follows.

Success rate@N. Given a set of testing projects P, success rate is defined as the ratio of queries that have at least a matched topic among the total number of queries.

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} \tag{7.3}$$

Accuracy. Given a list of *top-N* libraries, *precision* and *recall* are utilized to measure the *accuracy* of the recommendation results. In particular, *precision* is the ratio of the *top-N* recommended topics found in the ground-truth data, whereas *recall* is the ratio of the ground-truth topics belonging to the *N* recommended items [175]:

$$precision@N = \frac{|match_N(p)|}{N} \tag{7.4}$$

$$recall@N = \frac{|match_N(p)|}{|GT(p)|} \tag{7.5}$$

Catalog coverage. Given the set of projects, we compare the number of recommended topics with the global number of the available ones. This metric measures the suitability of the delivered topics considering all the possible set of values.

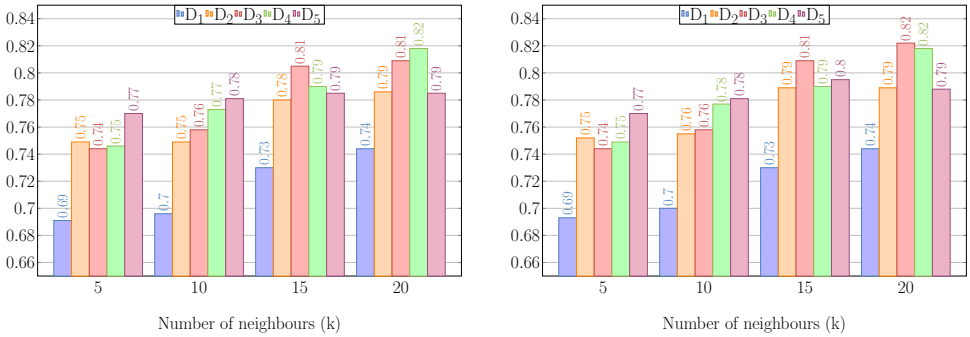
$$coverage@N = \frac{\left|\cup_{p \in P} REC_N(p)\right|}{|T|} \tag{7.6}$$

7.3.4 Evaluation process

The *ten-fold cross-validation* technique [156] has been used to assess the performance of TopFilter and its combined use with MNBN. Figure 7.7 depicts the evaluation process consisting of three consecutive steps, i.e., *Data Preparation*, *Recommendation production*, and *Outcome Evaluation* and they are going to be explained as follows.

Data Preparation. This phase is conducted to collect repositories that match the requirements defined in previous section from GitHub during the *Data collection* step. The dataset is then split into a training and testing set, i.e., `Split ten-fold`. Due to the different nature of the recommender systems (i.e., MNBN requires *README* files as input and training data, whereas TopFilter uses a set of assigned topics as input and for training) testing and training data needs to be specifically prepared for each approach. The `Split topic` activity resembles a real development process where a developer has already included some topics in the repository, i.e., *Query topics* and waits for recommendations.

Recommendation production. To enable the evaluation of TopFilter, we extracted a portion of topics from a given testing project, i.e., the ground-truth part. The remaining part is used as a query to produce recommendations. Since MNBN uses only README file(s) to predict a set of topics, it does not require any topics as input. The approach parses and encodes text files in vectors using the TF-IDF weighting scheme. Finally, the combined approach uses TopFilter as the recommendation engine which is fed by topics generated by MNBN. In this



(a) N=5 (b) N=10

Figure 7.8: Success rate with 5 and 10 input topics.

respect, both *Testing data* and *Training set* boxes are simplified to provide the needed data, i.e., README files and assigned topics to the recommender systems.

Outcome Evaluation. It is worth noting that we cannot directly compare TopFilter with MNBN since they rely on different input data. To be concrete, TopFilter is a supervised learning system that requires an initial set of assigned topics for the training, whereas MNBN is unsupervised learning and it uses only information mined from README files to recommend, without making use of any topics as input. Thus, we evaluate the performance of TopFilter by analysing the recommendation results that are compared with those stored as ground-truth data to compute the quality metrics (i.e., *Success rate*, *Precision*, *Recall*, and *Catalog coverage*). The same metrics are used to evaluate the performance of the combined use of TopFilter and MNBN.

7.4 Results

This section analyzes the performance of TopFilter as well as the combination of MNBN with TopFilter by addressing the two research questions as discussed in Section 7.4.1 and Section 7.4.1, respectively.

7.4.1 Result analysis

▷ ***RQ*$_1$*: Which TopFilter configuration yields the best performance?***

As presented in Section 7.3, given a testing project p, a certain number of topics is used as input, i.e., τ, and the remaining ones are saved as ground truth data, i.e., *GT(p)*. To find the configuration obtaining the best prediction performances, we experimented by varying the available parameters, i.e., the number of top-similar neighbour projects k, the number of recommended items N. To be more concrete, we chose two different values of N, i.e., N

= $\{5, 10\}$, and five values of k, i.e., $k = \{5, 10, 15, 20, 25\}$. Furthermore, we also considered all the five datasets defined in Section 7.3.2, i.e., D_1, D_5, D_{10}, D_{15}, and D_{20}. The value of τ is always considered as half of the number of topics already assigned to the project under analysis. In particular, given a testing project, the first half of the related topics are used to construct the query. The average success rates obtained by running the ten-fold cross-validation technique with TopFilter are depicted in Fig. 7.8(a) and Fig. 7.8(b).

Overall, it is evident that infrequent topics negatively affect the prediction outcomes. With D_1, i.e., all projects are considered, TopFilter obtains a low success rate by both configurations N=5 and N=10, compared to the results of other cut-off values t. For instance, with k=5, TopFilter gets 0.54 as success rate for D_1, meanwhile it gets 0.62 as success rate for D_5 and D_{10}, and 0.63 for D_{15} and D_{20}. The same trend can also be witnessed with other values of k, i.e., $k = \{10, 15, 20, 25\}$: TopFilter achieves a better performance when we consider a dataset with more topics for each project. This is understandable since TopFilter relies on the availability of training data to function: the more topics it has, the better it can compute the similarities among projects (cf. Fig. 7.4 and Eq. 7.1), and thus it is able to find more relevant topics (cf. Eq. 7.2). Similarly, for N=10, i.e., we consider a longer list of recommendations, TopFilter obtains a better success rate when a dataset with more detailed training data is exploited.

Next, we examine the influence of the number of neighbors k on the prediction performance. For N=5, from Fig. 7.8(a) it is clear that incorporating more neighbor projects for recommendation helps improve the performance considerably. Take as an example, for k=10 the best success rate is 0.69 obtained for D_{20}, and the corresponding score achieved with k=25 is 0.73 for D_{20}. For N=10, we see a clear gain in performance when more neighbors are considered for the computation of recommendations. The best obtained success rate is 0.83 when k=25 for D_{20}. As a whole, we come to the conclusion that increasing the number of neighbors used for computing missing ratings in the project-topic matrix is beneficial to the final recommendations. On the other hand, this also increases the computational complexity as comprehended in Eq. 7.2. Therefore, it is necessary to maintain a trade-off between accuracy and efficiency when deploying TopFilter by choosing a suitable value of k.

To further study TopFilter's performance, we computed and depicted in Fig. 7.9 the precision/recall curves (PRCs) for all the considered datasets. For this setting, the number of recommended items N was varied from 1 to 20, aiming to study the performance for a long recommendation list. Each dot in a curve corresponds to precision and recall obtained for a specific value of N. Furthermore, we fixed k=25 since this number of neighbors brings the best prediction outcomes among others, while it still maintains a reasonable execution time. As a PRC close to the upper right corner represents a better precision/call [17], Fig. 7.9

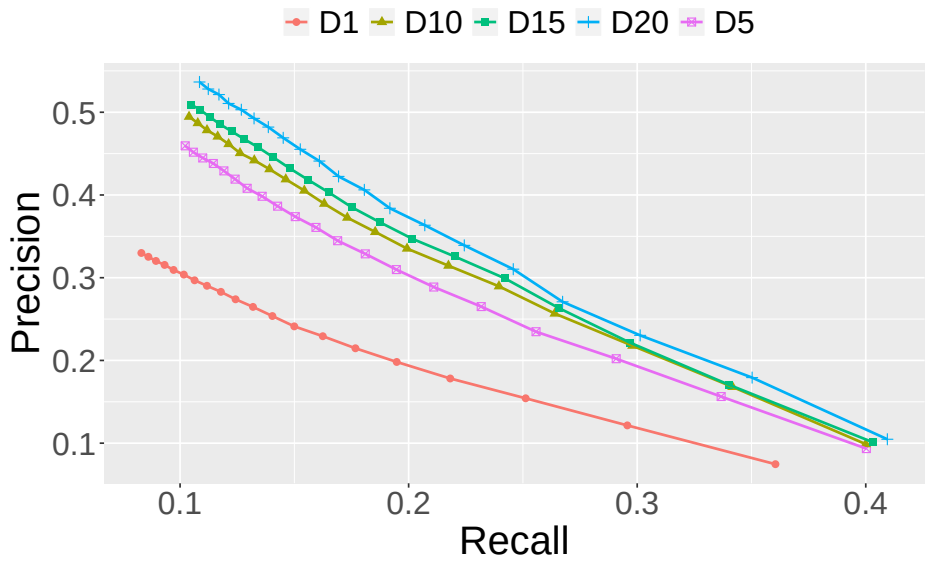


Figure 7.9: Precision/recall curves.

demonstrates that by considering a dataset with more topics for each repository, TopFilter yields a better prediction performance. In particular, the worst precision/recall relationship is seen by D_1, while the best one is obtained by D_{20}. Overall, these results are consistent with those presented in Fig. 7.8(a) and Fig. 7.8(b): using projects consisting of more input topics helps TopFilter enhance its performance substantially.

We investigate if TopFilter can recommend a wide range of topics to repositories by considering catalog coverage. The metric measures the percentage of recommended topic in the training data that the model is able to suggest to a test set, and a higher value corresponds to a better coverage.

Table 7.4 reports the average coverage value for the considered datasets, i.e., D_1, D_5, D_{10}, D_{15}, D_{20}.

Table 7.4: Catalog Coverage.

N	**D$_1$**	**D$_5$**	**D$_{10}$**	**D$_{15}$**	**D$_{20}$**
2	2.313	1.433	1.075	0.886	0.715
4	3.925	2.362	1.753	1.440	1.143
6	5.494	3.232	2.346	1.858	1.478
8	7.075	4.035	2.835	2.185	1.737
10	8.720	4.788	3.239	2.458	1.920
12	10.385	5.472	3.615	2.702	2.082
14	12.073	6.120	3.934	2.915	2.223
16	13.872	6.729	4.216	3.088	2.339
18	15.753	7.252	4.475	3.244	2.442
20	17.746	7.770	4.699	3.369	2.521

From the table, we see that by considering a longer list of items, i.e., increasing N, a better coverage is gained. Furthermore, using a higher cut-off value t has a negative impact on the global catalog coverage. For instance, with D_1 and N=2, we obtain 2.313 as coverage, and this score gradually decreases along t, and shrinks to 0.715 with D_{20}. Similarly, by other values of N, catalog coverage is large for a low t and small for a high t. This can be explained as follows: setting a high value of t means that a large amount of training data is discarded, and thus removing also topics. Altogether, we see that using a denser dataset for training, i.e., projects with more topics, is beneficial to success rate, accuracy, but not to catalog coverage.

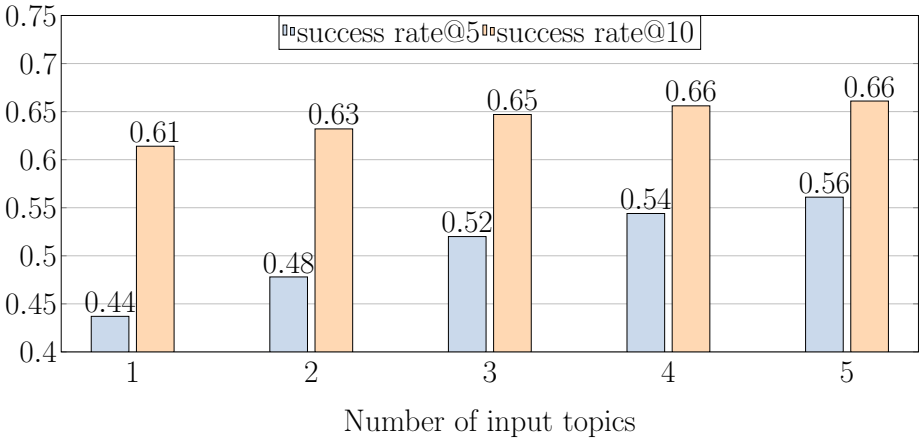


Figure 7.10: Success rates for τ={1,2,3,4,5} on D_{20}.

We are also interested to understanding how τ impacts on the prediction performance by performing a cross validation with τ={1,2,3,4,5}. Moreover, to simplify the evaluation, we fixed the number neighbors k to 25 and the number of topics t to 20 and applied TopFilter on the dataset D_{20}. Figure 7.10 reports the average success rate obtained for two values of N, i.e., N={5,10}. The figure demonstrates an evident outcome: by using more input topics as input data, we get a better prediction performance. Take as an example, for τ=1, we get a success rate of 0.44 and 0.61 for N=5 and N=10, respectively. When we use 5 topics to feed TopFilter, the corresponding success rate is 0.56 and 0.66 for N=5 and N=10, respectively. This happens due to the fact that by considering more input topics, TopFilter is able to better determine the similarity between the testing project and projects in the training data, as shown in Eq. 7.2, which eventually allows it to mine more relevant topics.

Answer to RQ$_1$. Given a project, TopFilter achieves a better success rate and accuracy when more similar projects are considered for recommendation. A higher cut-off value t negatively impacts on the coverage; and using more topics as input data helps the system improve its performance.

▷ ***RQ$_2$: To what extent can the accuracy of MNBN be improved by means of TopFilter?***

Table 7.5: Comparison between MNBN and *MNBN+TopFilter* using Dataset D_1.

	Success rate		Recall		Precision		Catalog coverage	
N	MNBN	*MNBN+TopFilter*	MNBN	*MNBN+TopFilter*	MNBN	*MNBN+TopFilter*	MNBN	*MNBN+TopFilter*
2	0.240	0.148	0.026	0.017	0.137	0.077	0.175	0.411
4	0.383	0.287	0.042	0.035	0.123	0.081	0.314	0.617
6	0.441	0.449	0.049	0.064	0.104	0.098	0.398	0.919
8	0.499	0.531	0.055	0.085	0.093	0.098	0.474	1.272
10	0.550	0.572	0.061	0.100	0.087	0.092	0.557	1.626
12	0.584	0.603	0.066	0.111	0.081	0.086	0.620	2.008
14	0.601	0.624	0.067	0.120	0.074	0.080	0.658	2.431
16	0.616	0.643	0.068	0.128	0.067	0.074	0.687	2.825
18	0.632	0.660	0.069	0.136	0.062	0.070	0.718	3.169
20	0.662	0.675	0.073	0.143	0.060	0.066	0.768	3.544

Table 7.6: Comparison between MNBN and *MNBN+TopFilter* using Dataset D_{20}.

	Success rate		Recall		Precision		Catalog coverage	
N	MNBN	*MNBN+TopFilter*	MNBN	*MNBN+TopFilter*	MNBN	*MNBN+TopFilter*	MNBN	*MNBN+TopFilter*
2	0.363	0.217	0.035	0.031	0.206	0.118	0.263	0.249
4	0.600	0.389	0.075	0.063	0.221	0.119	0.562	0.444
6	0.635	0.601	0.094	0.121	0.187	0.153	0.715	0.612
8	0.680	0.704	0.106	0.171	0.159	0.162	0.810	0.848
10	0.701	0.754	0.116	0.204	0.140	0.156	0.890	1.022
12	0.719	0.788	0.124	0.230	0.124	0.146	0.950	1.178
14	0.733	0.808	0.130	0.254	0.111	0.138	0.994	1.330
16	0.745	0.829	0.135	0.274	0.101	0.131	1.035	1.463
18	0.759	0.840	0.143	0.290	0.095	0.123	1.090	1.582
20	0.772	0.855	0.150	0.306	0.090	0.117	1.148	1.701

As already reasoned in Section 7.3.4, it is not feasible to directly compare TopFilter with MNBN, as they are based on different recommendation mechanisms. TopFilter relies on a supervised learning technique, requiring an initial set of assigned topics for the training. Meanwhile, MNBN works on the basis of an unsupervised learning system, which needs only data mined from README files to recommend, without being fed with any input topics.

In this research question, we aim to show that combining MNBN with TopFilter (referred with *MNBN+TopFilter* hereafter) helps boost up the recommendation outcomes provided by MNBN. In particular, we run TopFilter on the outputs produced by MNBN, attempting to generate a more relevant list of items.

We conducted experiments on two of the selected datasets, i.e., D_1 and D_{20}, as they correspond to two distinct levels of data completeness: D_1 has more projects but with a lower quality, while D_{20} has fewer data but with a higher quality. In the experiment, τ was set to 5 since through RQ_1, we realized that this value fosters the best performance, among others (see Fig. 7.10). The success rate, precision, recall and coverage scores obtained for D_1 and D_{20} are reported in Table 7.5 and Table 7.6, respectively.

The results in Table 7.5 demonstrate that compared to TopFilter, MNBN obtains a better prediction performance in terms of success rate, recall and precision for a ranked list with

a low number of items, i.e., N $\leq$ 4. In particular, with N=2, MNBN gets 0.240 as success rate, while the corresponding score by *MNBN+TopFilter* is 0.148. The same trend can be seen with recall and precision. This is understandable as MNBN relies on only README file(s) to function, and its performance is not affected by the number of considered topics. In contrast, TopFilter needs input topics to provide recommendation, that is the reason why for a dataset with a low quality dataset (with respect to the number of topics), TopFilter gets a moderate performance.

Table 7.5 also shows that the combined use of MNBN and TopFilter outperforms MNBN when we consider a longer list of recommended items. To be concrete, starting from N=6 (i.e., the row marked with gray), all the metrics computed with *MNBN+TopFilter* are superior to those computed with MNBN. For example, *MNBN+TopFilter* gets 0.675 as the maximum success rate, while the corresponding score by MNBN is 0.662. More importantly, *MNBN+TopFilter* always achieves a much better 0 coverage than that of MNBN: By the best configuration, *MNBN+TopFilter* reaches a coverage of 3.544, which is much higher than 0.768 obtained by MNBN. In other words, our proposed approach is able to recommend a wider range of topics than MNBN. This can be explained by the fact that TopFilter takes into consideration a set of topics as input, and the more data it has, the larger the set of topics it can recommend.

By examining Table 7.6, we encounter a similar outcome compared to that of Table 7.5: *MNBN+TopFilter* outperforms MNBN by all the quality metrics, i.e., success rate, recall, precision and coverage. Especially, using D_{20} as the training data, *MNBN+TopFilter* improves the overall success rate considerably, with respect to using D_1. This further confirms our findings in RQ_1: A denser dataset facilitates the capability of recommending a more relevant set of topics.

Answer to RQ_2. Compared to MNBN, the combined used of MNBN and TopFilter substantially improves the prediction performance with respect to success rate, precision and recall. Moreover, while MNBN suffers a low catalog coverage, *MNBN+TopFilter* is able to recommend a wide range of topics to repositories.

7.4.2 Threats to Validity

This section discusses the threats that may affect the results of the evaluation. We also list the countermeasures taken to minimize these issues.

The *internal validity* could be compromised by the dataset features, i.e., the number of projects for each topic, the number of results. We tackle this issue by varying the

aforementioned parameters to build datasets with different characteristics. In this way, several settings have been used to evaluate TopFilter's overall performances.

External validity concerns the rationale behind the selection of the GitHub repositories used in the assessment. As stated in Section 7.3, we randomly downloaded repositories by imposing a quality filter on the number stars. Nevertheless, some repositories could be tagged with topics that can affect the quality of the graph computed in the data extraction phase. To be concrete, a user can label a repository using terms that are not descriptive enough, i.e., using infrequent or duplicated terms in the topic list. To deal with this issue, we applied the topic filter as described in Section 7.3.2 to reduce any possible noise during the graph construction phase.

Threats to *construction validity* are related to the choice of MNBN as the baseline in the conducted experiments. First of all, the availability of the replication package allows us to perform a comprehensive evaluation. As we claimed before, the two approaches are strongly different from the construction point of view including the recommendation engine and data extraction components. To make the comparison as fair as possible, we ran MNBN on the same datasets by adapting the overall structure for the ten-fold cross-validation evaluations.

7.5 Conclusion

GitHub is nowadays among the most popular platforms to handle and maintain OSS projects. Topics have been introduced in 2017 to promote projects' visibility on the platform. In this work, we presented TopFilter, a collaborative filtering-based recommender system to suggest GitHub topics. By encoding repositories and related topics in a graph-based representation, we built a project-topic matrix and applied a syntactic-based similarity function to predict missing topics. To assess the prediction performances, we combined our approach with a well-founded system based on an ML algorithm, i.e., MNBN. In particular, by taking as input in TopFilter, the output produced by MNBN, we showed that by means of a combined usage of such approaches, it is possible to obtain a significant boost in topic predictions. Nevertheless, the accuracy did not reach higher values in all the experimental settings. To our best knowledge, this depends very much on the similarity function used in the recommendation engine as well as on the heterogeneity of the dataset. Thus, we plan to extend TopFilter by adding different degrees of similarity, e.g., semantic analysis on topics. Moreover, we can enlarge the evaluation by considering other common metrics in the collaborative filtering domain such as sales diversity and novelty. All of this is planned as future work.

Chapter 8

Conclusions

During their daily tasks, developers use a plethora of instruments and sources of information. Nevertheless, the overwhelming amount of available artifacts forces developers to search for the most appropriate one. Therefore, recommender systems for software engineering have been conceived as an aiding mechanism to assist developers by providing them with reusable artifacts like third-party libraries, code snippets, and documentation, to mention but a few. The contributions in the domain of recommender systems that have been presented throughout this dissertation are highlighted in Section 8.1. The related publications and tools that have been produced while working on this work are listed in Section 8.2 and Section 8.3, respectively. Finally, future work is described in Section 8.4.

8.1 Summary of the contributions

Being motivated by the challenges presented in Section 1.1, in this dissertation, we addressed the problem of supporting software developers by providing them with different recommendations relevant to the current development context. In this respect, the main contributions of our work are summarized below.

Recommending similar open-source projects. In Chapter 3, CrossSim (**C**ross project **R**elationships for computing **O**pen **S**ource **S**oftware **Sim**ilarity) was developed as a framework for computing similarities among OSS projects. CrossSim makes use of graphs for modeling different types of relationships in the OSS ecosystems, and computes similarities among the projects therein. We evaluated the approach by computing similarities among OSS projects on a dataset of 580 GitHub Java projects. The performed evaluation demonstrated that CrossSim outperforms the baselines including MudaBlue, CLAN, and RepoPal.

Exploiting user-feedback to recommend third-party libraries. In Chapter 4, we presented an approach to exploiting user feedback to enhance third-party libraries' recommendations. By taking into account three types of feedback i.e., positive, negative, and additive, we demonstrate that it is possible to increase the performance of a well-founded TPLs recommender system. We assume that such a mechanism is helpful also in the context of other types of recommendations in software engineering.

Recommending Stack Overflow posts. Chapter 5 introduced PostFinder, a recommender system, which can retrieve Stack Overflow posts being relevant for the current development context. We first evaluated PostFinder, comparing it with FaCoY, a code-to-code search engine by means of a user study. Moreover, we also performed a light comparison with PROMPTER, a tool with similar functionalities. The experimental results showed that our proposed approach obtains a better prediction performance compared to both baselines. We conclude that PostFinder is capable of helping developers to fetch posts that can be useful for solving their coding tasks.

Recommending upgrade plans for third-party libraries. In Chapter 6, we presented EvoPlan, a recommender system that can provide developers with upgrade plans for TPLs. The approach aims to suggest the most appropriate migration plan by considering two key factors: the popularity of the upgrade plan and the availability of discussions about it. An evaluation of about 11,000 projects was conducted, showing promising results.

Furthermore, we conceived DeepLib, a Deep Learning based recommender system to provide an upgrade plan for TPLs, employing Encoder-Decoder neural network as the prediction engine. By analyzing the migration history of mined projects, we build matrices containing libraries and their versions in chronological order, which are fed to the recommendation engine. We evaluated the tool using a dataset collected from more than 1,000 public Maven repositories.

Recommending topics for GitHub repositories. Chapter 7 presented TopFilter, a recommender system that employs a collaborative filtering technique to suggest topics that can be assigned to new GitHub projects. This approach has been developed to extend the capabilities of an existing work [16]. The ten-fold cross-validation methodology has been used to assess the performance of TopFilter by considering different metrics, i.e., success rate, precision, recall, and catalog coverage. The results show that TopFilter recommends good topics depending on different factors, i.e., collaborative filtering settings, considered datasets, and pre-processing activities. Moreover, TopFilter can be combined with a state-of-the-art topic recommender system, i.e., MNBN, to improve the overall prediction performance. Our

results confirm that collaborative filtering techniques can successfully be used to provide relevant topics for GitHub repositories.

8.2 Publications

The work presented in this dissertation has been published in various peer-reviewed journals and conference/workshop proceedings. Two papers, namely W1 [176] and W2 [177] employ techniques conceived in the dissertation to solve issues in different domains, thus they are not presented as a part of the dissertation. In particular, W1 presents an approach to recommend metamodels by exploiting a general purpose search engine developed for PostFinder. Meanwhile, W2 describes a mechanism to classify and recommend video game genre tags, using the machine learning techniques developed in Chapter 7.

Journal Papers

J1 - Riccardo Rubei, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, Juri Di Rocco, "*Providing Upgrade Plans for Third-party Libraries: A Recommender System using Migration Graphs*," Springer Applied Intelligence (APIN), 2022, DOI: 10.1007/s10489-021-02911-4. ***This paper was presented in Chapter 6***.

J2 - Riccardo Rubei, Claudio Di Sipio, Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, "*PostFinder: Mining Stack Overflow posts to support software developers*," Elsevier Information and Software and Technology (IST), 2020, ISSN: 0950-5849, DOI: 10.1016/j.infsof.2020.106367. ***This paper was presented in Chapter 5.***

J3 - Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Claudio Di Sipio, Davide Di Ruscio, "*DeepLib: Machine Translation Techniques to Recommend Upgrades for Third-party Libraries*," Elsevier Expert Systems with Applications (ESWA), 2022, ISSN: 0957-4174, DOI: 10.1016/j.eswa.2022.117267. ***This paper was presented in Chapter 6***.

J4 - Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, Riccardo Rubei, "*Development of recommendation systems for software engineering: the CROSSMINER experience*," Springer Empirical Software Engineering (EMSE), 2021, ISSN: 1573-7616, DOI: 10.1007/s10664-021-09963-7. ***This paper is a general report for all the chapters in this dissertation***.

J5 - Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Davide Di Ruscio, "*An Automated Approach to Assess the Similarity of GitHub Repositories*," Springer Software Quality

Journal (SQJ), Vol 28, pages 595–631, 2020, ISSN: 0963-9314, DOI: 10.1007/s11219-019-09483-0. ***This paper was presented in Chapter 3***.

J6 - Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, Riccardo Rubei, "*HybridRec: A Recommender System for Tagging GitHub Repositories*," submitted to Springer Applied Intelligence (APIN), ***Under review after revision***.

Conference Papers

C1 - Riccardo Rubei, Davide Di Ruscio, Claudio Di Sipio, Juri Di Rocco, Phuong T. Nguyen, "*Endowing third-party libraries recommender systems with explicit user feedback mechanism*," in Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, DOI: 10.1109/SANER534-32.2022.00099. ***This paper was presented in Chapter 4***.

C2 - Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Davide Di Ruscio, "*CrossSim: exploiting mutual relationships to detect similar OSS projects*," in Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018, ISBN: 978-1-5386-7383-6, DOI: 10.1109/SEAA.2018.00069, ***Distinguished paper award*** (https://bit.ly/3hrPMr1). ***This paper was presented in Chapter 3***.

C3 - Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong Nguyen, Riccardo Rubei, "*TopFilter: An Approach to Recommend Relevant GitHub Topics*," in Proceedings of the 14th International Symposium on Empirical Software Engineering and Measurement, ESEM 2020, ISBN: 978-1-4503-7580-1, DOI: 10.1145/3382494.3410690. ***This paper was presented in Chapter 7***.

C4 - Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, Phuong T. Nguyen, "*A Multinomial Naïve Bayesian (MNB) Network to Automatically Recommend Topics for GitHub Repositories*," in Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering, EASE 2020, ISBN: 978-1-4503-7731-7, DOI: 10.1145/3383219.3383227. ***This paper was presented in Chapter 7***.

Workshop Papers

W1 - Riccardo Rubei, Juri Di Rocco, Davide Di Ruscio, Phuong T. Nguyen, Alfonso Pierantonio, "*A Lightweight Approach for the Automated Classification and Clustering of Metamodels*," in Proceedings of the 2nd International Workshop on Open Model Based

Engineering Environment (OpenMBEE 2021), co-located with MODELS 2021, DOI: 10.1109/MODELS-C53483.2021.00074. ***This paper employs a similar technique proposed in Chapter 5 to classify metamodels, however it is not presented in any chapter of the dissertation***.

W2 - Riccardo Rubei, Claudio Di Sipio, "*AURYGA: A Recommender System for Game Tagging*," in Proceedings of the 11th Italian Information Retrieval Workshop, September 13–15, 2021, Bari, Italy, IIR 2021, URL: http://ceur-ws.org/Vol-2947/paper10.pdf. ***This paper is not presented in any chapter of the dissertation***.

W3 - Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Claudio Di Sipio, Davide Di Ruscio,"*Recommending Third-party Library Updates with LSTM Neural Networks*," in Proceedings of the 11th Italian Information Retrieval Workshop, September 13–15, 2021, Bari, Italy, IIR 2021, URL: http://ceur-ws.org/Vol-2947/paper7.pdf. ***This paper was presented in Chapter 6***.

8.3 Developed tools

The following tools have been produced while working on the challenges addressed by this dissertation:

- CrossSim [178, 92]: it is an Eclipse-based tool able to calculate similarities among open-source projects as presented in Chapter 3;
- PostFinder [20]: it is an Eclipse-based tool that can automatically retrieve Stack Overflow posts that are relevant for the current development context. The tool implements the approach presented in Chapter 5.

8.4 Future work

This dissertation conceptualizes a set of recommender systems to support software developers in their daily tasks. By mining data from various sources, the systems provide developers with different types of recommendations, among others, similar projects to a project under development, upgrading plan for third-party libraries, or Stack Overflow posts relevant to the code being developed, or topics related to a GitHub repository. Altogether, this enables developers to leverage the existing sources of information to improve their work efficiency and effectiveness. We anticipate that the different approaches presented in this dissertation can be further developed as discussed below.

Making the proposed user feedback support generic. The user feedback mechanism proposed in this work has been applied to a specific recommender system to improve the accuracy of recommendations consisting of third-party libraries. We plan to extend the work to make the approach generic and applicable to any recommender systems to push forward the idea of introducing the support for human-in-the loops. In the first place, we suppose that the endowing mechanism presented in this dissertation is also applicable to API recommendations [12, 14, 179].

Enabling countermeasures to avoid adversarial attacks. Recommender system are prone to malicious attacks [180, 181]. It could be possible to steer the recommended items by injecting fake information that will be used by the systems to calculate the similarity among the stored items. It is also possible to inject fake feedbacks in order to modify the final rank of an item. For example, by down voting or by expressing low rating intentionally, the visibility of an item could be compromised. The work presented in Chapter 4 did not take into account security and countermeasures aspects, and this is what we will investigate deeply in the forthcoming research.

Improving the performance of Q&A recommender systems. The PostFinder tool presented in Chapter 5 to recommend useful Stack Overflow posts could be further improved as follows. Two additional search criteria can be added to identify the Q&A posts to be recommended, i.e., *energy consumption* and *sentiment analysis* of the contained code. The sustainability issue has gained attention in the software engineering community. In a recent work [182], the authors measured energy efficiency of Java data structure in a possible similar fashion that we use to rank first the posts that include source code being more efficient. We aim at developing a recommender system which is as much as possible platform agnostic. Moreover, we believe that employing sentiment analysis [183, 184] to analyze textual description and comments of the posts containing source code would help retrieve more relevant posts. Thus, posts can be ranked according to the detected sentiment.

Supporting the migration of source code. Concerning EvoPlan and DeepLib, two approaches conceived to support developers in upgrading their third-party libraries in Chapter 6, we anticipate that there are the following possible developments. Even though suggesting the appropriate library version is utile, developers may face problems in the concrete upgrading phase. The problem of supporting migrations has been intensively studied [67, 70, 71]. However, we noticed that a proper post-migration assistance is still missing. Problems like the lack of easy accessible documentation of new libraries or mandatory manual code changes

still need to be solved. We plan to support developers in two different manners. First, we can enrich the recommendation by providing ad-hoc documentation related to possible code changes. Second, a dedicated mechanism can be deployed to automatically modify the code according to the mandatory modifications.

Bibliography

[1] C. Olah, "Understanding LSTM Networks," May 2020.

[2] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, (Doha, Qatar), pp. 103–111, Association for Computational Linguistics, Oct. 2014.

[3] P. T. Nguyen, J. Di Rocco, R. Rubei, C. Di Sipio, and D. Di Ruscio, "Deeplib: Machine translation techniques to recommend upgrades for third-party libraries," *Expert Systems with Applications*, vol. 202, p. 117267, 2022.

[4] V. Cosentino, J. L. Cánovas Izquierdo, and J. Cabot, "A systematic mapping study of software development with github," *IEEE Access*, vol. 5, pp. 7173–7192, 2017.

[5] D. Spinellis and C. Szyperski, "How is open source affecting software development?," *IEEE Software*, vol. 21, pp. 28–33, Jan 2004.

[6] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza, "Prompter - turning the IDE into a self-confident programming assistant," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2190–2231, 2016.

[7] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. 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, (New York, NY, USA), pp. 275–284, ACM, 2010.

[8] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, pp. 76–80, Jan. 2003.

[9] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, pp. 13:1–13:19, Dec. 2015.

[10] L. Wu, S. Shah, S. Choi, M. Tiwari, and C. Posse, "The browsemaps: Collaborative filtering at linkedin," in *RSWeb@RecSys*, vol. 1271 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014.

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

[12] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns," in *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, (Piscataway, NJ, USA), pp. 1050–1060, IEEE Press, 2019.

[13] J. Fowkes and C. 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, (New York, NY, USA), pp. 254–265, ACM, 2016.

[14] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, (Piscataway, NJ, USA), pp. 880–890, IEEE Press, 2015.

[15] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. Nguyen, and R. Rubei, "TopFilter: An Approach to Recommend Relevant GitHub Topics," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[16] C. Di Sipio, R. Rubei, D. Di Ruscio, and P. T. Nguyen, "Using a Multinomial Naïve Bayesian (MNB) Network to Automatically Recommend Topics for GitHub Repositories," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering, EASE2020, Trondheim, Norway, April 15-17, 2020*, EASE'20, pp. 24–34, ACM, 2020.

[17] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "CrossRec: Supporting Software Developers by Recommending Third-party Libraries," *Journal of Systems and Software*, p. 110460, 2019.

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

[19] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter," in *11th Working Conference on Mining Software Repositories*, (New York), pp. 102–111, ACM, 2014.

[20] R. Rubei, C. Di Sipio, P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, "PostFinder: Mining Stack Overflow posts to support software developers," *Information and Software Technology*, vol. 127, p. 106367, 2020.

[21] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, and R. Rubei, "Development of recommendation systems for software engineering: the CROSSMINER experience," vol. 26, no. 4, p. 69.

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

[23] P. T. Nguyen, P. Tomeo, T. Di Noia, and E. Di Sciascio, "An evaluation of SimRank and Personalized PageRank to Build a Recommender System for the Web of Data," in *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, (New York, NY, USA), pp. 1477–1482, ACM, 2015.

[24] P. K. Garg, S. Kawaguchi, M. Matsushita, and K. Inoue, "MUDABlue: An Automatic Categorization System for Open Source Repositories," *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 184–193, 2004.

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

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

[27] H. Wang, Y. Guo, Z. Ma, and X. 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, (New York, NY, USA), pp. 71–82, ACM, 2015.

[28] P. D. Turney and P. Pantel, "From frequency to meaning: Vector space models of semantics," *J. Artif. Int. Res.*, vol. 37, pp. 141–188, Jan. 2010.

[29] A. Tversky, "Features of similarity," *Psychological Review*, vol. 84, no. 4, pp. 327–352, 1977.

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

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

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

[33] P. T. Nguyen, J. Di Rocco, and D. 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, (New York, NY, USA), pp. 326–331, ACM, 2019.

[34] N. Chen, S. C. Hoi, S. Li, and X. Xiao, "SimApp: A Framework for Detecting Similar Mobile Applications by Online Kernel Learning," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, (New York, NY, USA), pp. 305–314, ACM, 2015.

[35] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable Detection of Semantically Similar Android Applications," in *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pp. 182–199, 2013.

[36] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts," in *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, (New York, NY, USA), pp. 319–330, ACM, 2018.

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

[38] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, Nov. 2011.

[39] T. Landauer, P. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, pp. 259–284, 1998.

[40] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[41] U. Bhandari, K. Sugiyama, A. Datta, and R. Jindal, "Serendipitous recommendation for mobile apps using item-item similarity graph.," in *AIRS* (R. E. Banchs, F. Silvestri, T.-Y. Liu, M. Zhang, S. Gao, and J. Lang, eds.), vol. 8281 of *Lecture Notes in Computer Science*, pp. 440–451, Springer, 2013.

[42] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, pp. 39–41, Nov. 1995.

[43] Y. Zhou, H. Jin, X. Yang, T. Chen, K. Narasimhan, and H. C. Gall, "BRAID: an API recommender supporting implicit user feedback," in *ESEC/FSE*, pp. 1510–1514, 2021.

[44] A. Ghazimatin, S. Pramanik, R. S. Roy, and G. Weikum, "ELIXIR: learning from user feedback on explanations to improve recommender models," WWW, pp. 3850–3860, ACM / IW3C2, 2021.

[45] S. Wang, D. Lo, and L. Jiang, "Active code search: Incorporating user feedback to improve code search relevance," ASE, p. 677–682, 2014.

[46] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "Webapirec: Recommending web apis to software projects via personalized ranking," *IEEE TETCI*, vol. 1, no. 3, pp. 145–156, 2017.

[47] V. W. Anelli, Y. Deldjoo, T. Di Noia, A. Ferrara, and F. Narducci, "Federank: User controlled feedback with federated recommender systems," in *Advances in Information Retrieval*, pp. 32–47, 03 2021.

[48] G. Lin, F. Liang, W. Pan, and Z. Ming, "Fedrec: Federated recommendation with explicit feedback," *IEEE Intelligent Systems*, vol. 36, no. 5, pp. 21–30, 2021.

[49] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[50] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *J. Systems and Software*, vol. 145, pp. 164 – 179, 2018.

[51] 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," ICSE, pp. 335–346, 2017.

[52] 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)*, pp. 38–42, June 2012.

[53] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the ide," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 1295–1298, May 2013.

[54] L. B. L. de Souza, E. C. Campos, and M. d. A. Maia, "Ranking crowd knowledge to assist software development," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, (New York, NY, USA), pp. 72–82, ACM, 2014.

[55] B. Xu, Z. Xing, X. Xia, and D. Lo, "AnswerBot: Automated generation of answer summary to developers' technical questions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (Urbana, IL), pp. 706–716, IEEE, Oct. 2017.

[56] P. C. Rigby and M. 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* (D. Notkin, B. H. C. Cheng, and K. Pohl, eds.), pp. 832–841, IEEE Computer Society, 2013.

[57] M. Liu, X. Peng, Q. Jiang, A. Marcus, J. Yang, and W. Zhao, "Searching StackOverflow Questions with Multi-Faceted Categorization," in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware - Internetware '18*, (Beijing, China), pp. 1–10, ACM Press, 2018.

[58] "searchcode | source code search engine."

[59] "Home | krugle - software development productivity."

[60] "Search Engine for Source Code - PublicWWW.com."

[61] "Java Code Examples."

[62] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "FaCoY: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, eds.), pp. 946–957, ACM, 2018.

[63] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 72–83, 2021.

[64] D. Katsuragawa, A. Ihara, R. G. Kula, and K. Matsumoto, "Maintaining Third-Party Libraries through Domain-Specific Category Recommendations," in *2018 IEEE/ACM 1st International Workshop on Software Health (SoHeal)*, pp. 2–9, May 2018.

[65] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, "Diversified third-party library prediction for mobile app development," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[66] Z. Sun, Y. Liu, Z. Cheng, C. Yang, and P. Che, "Req2Lib: A Semantic Neural Model for Software Library Recommendation," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 542–546, Feb. 2020. ISSN: 1534-5351.

[67] S. Xu, Z. Dong, and N. Meng, "Meditor: Inference and Application of API Migration Edits," in *2019 IEEE/ACM 27th Int. Conf. on Program Comprehension (ICPC)*, pp. 335–346, May 2019.

[68] A. Hora and M. T. Valente, "Apiwave: Keeping track of API popularity and migration," in *2015 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pp. 321–323, Sept. 2015.

[69] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining Library Migration Graphs," in *2012 19th Working Conf. on Reverse Engineering*, pp. 289–298, Oct. 2012.

[70] C. Chen, "SimilarAPI: Mining Analogical APIs for Library Migration," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 37–40, Oct. 2020. ISSN: 2574-1926.

[71] M. Fazzini, Q. Xin, and A. Orso, "APIMigrator: an API-usage migration tool for Android apps," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, (Seoul Republic of Korea), pp. 77–80, ACM, July 2020.

[72] Y. Duan, L. Gao, J. Hu, and H. Yin, "Automatic generation of non-intrusive updates for third-party libraries in android applications," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pp. 277–292, USENIX Association, 2019.

[73] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, Feb. 2018.

[74] Ganesan, "Topic Suggestions for Millions of Repositories - The GitHub Blog," 2019.

[75] N. Orii, "Collaborative topic modeling for recommending github repositories," *Inf. Softw. Technol.*, vol. 83, no. 2, pp. 110–121, 2012.

[76] Z. Liao, T. Song, Y. Wang, X. Fan, and Y. Zhang, "User personalized label set extraction algorithm based on LDA and collaborative filtering in open source software community," in *2018 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pp. 1–5, July 2018.

[77] S. Ajoudanian and M. N. Abadeh, "Recommending human resources to project leaders using a collaborative filtering-based recommender system: Case study of gitHub," *IET Software*, vol. 13, no. 5, pp. 379–385, 2019. Conference Name: IET Software.

[78] W. Xu, X. Sun, J. Hu, and B. Li, "REPERSP: Recommending Personalized Software Projects on GitHub," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 648–652, Sept. 2017.

[79] S. Purushotham and Y. Liu, "Collaborative Topic Regression with Social Matrix Factorization for Recommendation Systems," p. 8.

[80] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to turn the IDE into a self-confident programming prompter," in *Proceedings of MSR 2014*, pp. 102–111, ACM, 2014.

[81] P. T. Nguyen, J. Di Rocco, and D. 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.

[82] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165 – 1199, 2013.

[83] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs," in *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, (New York, NY, USA), pp. 266–270, ACM, 1999.

[84] W. S. Evans, C. W. Fraser, and F. Ma, "Clone detection via structural abstraction," *Software Quality Journal*, vol. 17, pp. 309–330, Dec 2009.

[85] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, pp. 2464–2519, Aug 2018.

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

[87] R. Tiarks, R. Koschke, and R. Falke, "An extended assessment of type-3 clones as detected by state-of-the-art tools," *Software Quality Journal*, vol. 19, pp. 295–331, Jun 2011.

[88] A. M. Leitão, "Detection of redundant code using r2d2," *Software Quality Journal*, vol. 12, pp. 361–382, Dec 2004.

[89] A. E. V. B. Coutinho, E. G. Cartaxo, and P. D. de Lima Machado, "Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing," *Software Quality Journal*, vol. 24, pp. 407–445, 2014.

[90] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, and V. Chang, "A survey on test suite reduction frameworks and tools," *Int. J. Inf. Manag.*, vol. 36, pp. 963–975, Dec. 2016.

[91] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. von Gudenberg, "Similarity in programs," in *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006*, 2006.

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

[93] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen, "The adaptive web," ch. Collaborative Filtering Recommender Systems, pp. 291–324, Berlin, Heidelberg: Springer-Verlag, 2007.

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

[95] T. Di Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker, "Linked Open Data to Support Content-based Recommender Systems," in *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, (New York, NY, USA), pp. 1–8, ACM, 2012.

[96] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 800–813, Aug. 1991.

[97] S. Ugurel, R. Krovetz, and C. L. 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, (New York, NY, USA), pp. 632–638, ACM, 2002.

[98] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), pp. 872–881, ACM, 2006.

[99] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 3, p. 1–22, 2009.

[100] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer, and R. Lee, "Media meets semantic web – how the bbc uses dbpedia and linked data to make connections," in *The Semantic Web: Research and Applications* (L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, R. Mizoguchi, E. Oren, M. Sabou, and E. Simperl, eds.), (Berlin, Heidelberg), pp. 723–737, Springer Berlin Heidelberg, 2009.

[101] C. Stadler, J. Lehmann, K. Höffner, and S. Auer, "LinkedGeoData: A core for a web of spatial open data," *Semantic Web*, vol. 3, pp. 333–354, 2012.

[102] P. T. Nguyen, J. Di Rocco, and D. 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), ACM, 2018.

[103] H. Nassar, N. Veldt, S. Mohammadi, A. Grama, and D. F. Gleich, "Low rank spectral network alignment," in *Proceedings of the 2018 World Wide Web Conference*, WWW '18, (Republic and Canton of Geneva, Switzerland), pp. 619–628, International World Wide Web Conferences Steering Committee, 2018.

[104] M. Wang, C. Wang, J. X. Yu, and J. Zhang, "Community detection in social networks: An in-depth benchmarking study with a procedure-oriented framework," *Proc. VLDB Endow.*, vol. 8, pp. 998–1009, June 2015.

[105] G. Kollias, M. Sathe, O. Schenk, and A. Grama, "Fast parallel algorithms for graph similarity and matching," *J. Parallel Distrib. Comput.*, vol. 74, pp. 2400–2410, May 2014.

[106] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio, "CrossSim tool and evaluation data," 2018. https://github.com/crossminer/CrossSim.

[107] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang, "Why and how developers fork what from whom in github," *Empirical Softw. Engg.*, vol. 22, pp. 547–578, Feb. 2017.

[108] 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)*, pp. 334–344, Oct 2016.

[109] 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)*, pp. 251–262, July 2017.

[110] S. Ghose and O. Lowengart, "Taste tests: Impacts of consumer perceptions and preferences on brand positioning strategies," *Journal of Targeting, Measurement and Analysis for Marketing*, vol. 10, pp. 26–41, Aug 2001.

[111] S. Pettigrew and S. Charters, "Tasting as a projective technique," *Qualitative Market Research: An International Journal*, vol. 11, no. 3, pp. 331–343, 2008.

[112] C. Spearman, "The proof and measurement of association between two things.," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904. Place: US Publisher: Univ of Illinois Press.

[113] M. G. Kendall, "A New Measure of Rank Correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[114] T.-Y. Liu, *Learning to Rank for Information Retrieval.* Springer, 2011.

[115] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," ICML, (Bonn, Germany), pp. 89–96, ACM Press, 2005.

[116] J. Weston, H. Yee, and R. J. Weiss, "Learning to rank recommendations with the k-order statistic loss," RecSys, (New York, NY, USA), p. 245–248, Association for Computing Machinery, 2013.

[117] M. Kula, "Metadata embeddings for user and item cold-start recommendations," in *Procs. the 2nd CBRecSys, co-located with RecSys 2015, Vienna, Austria, September 16-20, 2015.*, vol. 1448, pp. 14–21, CEUR-WS.org, 2015.

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

[119] "Stackoverflow." https://stackoverflow.com/. last access 04.04.2019.

[120] R. Abdalkareem, E. Shihab, and J. Rilling, "On code reuse from stackoverflow: An exploratory study on android apps," *Information and Software Technology*, vol. 88, pp. 148 – 158, 2017.

[121] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. 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, (New York, NY, USA), pp. 83–94, ACM, 2014.

[122] Lucene, "Apache Lucene Core." https://lucene.apache.org/core/. last access 26.04.2019.

[123] R. Rubei, C. Di Sipio, P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, "PostFinder - Online appendix," June 2020.

[124] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *Journal of Systems and Software*, vol. 126, pp. 57 – 84, 2017.

[125] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon, "Augmenting and structuring user queries to support efficient free-form code search," *Empirical Software Engineering*, vol. 23, pp. 2622–2654, Oct 2018.

[126] A. Benelallam, N. Harrand, C. S. Valero, B. Baudry, and O. Barais, "Maven central dependency graph," Nov. 2018.

[127] C. E. Shannon, "A mathematical theory of communication," *Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[128] M. Borg, P. Runeson, J. Johansson, and M. 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, (New York, NY, USA), pp. 8:1–8:4, ACM, 2014.

[129] J. Pérez-Iglesias, J. R. Pérez-Agüera, V. Fresno, and Y. Z. Feinstein, "Integrating the probabilistic models BM25/BM25F into lucene," *CoRR*, vol. abs/0911.5046, 2009.

[130] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: the kappa statistic.," *Family Medicine*, vol. 37.5, pp. 360–363, 2005.

[131] M. Happ, A. Bathke, and E. Brunner, "Optimal sample size planning for the wilcoxon-mann-whitney test," *Statistics in Medicine*, vol. 38, 10 2018.

[132] J. L. Hintze and R. D. Nelson, "Violin plots: A box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998.

[133] F. Wilcoxon, *Individual Comparisons by Ranking Methods*, pp. 196–202. New York, NY: Springer New York, 1992.

[134] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition ed., 2005.

[135] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on java library migrations: Prevalence, trends, and rationales," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, (New York, NY, USA), p. 478–490, Association for Computing Machinery, 2021.

[136] J. Visser, A. van Deursen, and S. Raemaekers, "Measuring software library stability through historical version analysis," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, (USA), p. 378–387, IEEE Computer Society, 2012.

[137] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140 – 158, 2017.

[138] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android.," in *ACM Conference on Computer and Communications Security* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 2187–2200, ACM, 2017.

[139] J. Huang, N. Borges, S. Bugiel, and M. Backes, "Up-to-crash: Evaluating third-party library updatability on android," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 15–30, 2019.

[140] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 35–45, 2020.

[141] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, Mar. 2015.

[142] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, p. 1735–1780, Nov. 1997.

[143] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, (Cambridge, MA, USA), p. 3104–3112, MIT Press, 2014.

[144] Github, "Viewing and updating vulnerable dependencies in your repository - GitHub Docs," Sept. 2020.

[145] H. Alrubaye, M. W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, and J. Mcgoff, "Learning to recommend third-party library migration opportunities at the api level," *Applied Soft Computing*, vol. 90, pp. 106–140, 2020.

[146] Z. Xing and E. Stroulia, "API-Evolution Support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, pp. 818–836, Dec. 2007.

[147] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, and M. O'Boyle, "M3: Semantic API Migrations," *arXiv:2008.12118 [cs]*, Aug. 2020. arXiv: 2008.12118.

[148] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically Assisting Android API Migrations Using Code Examples," *arXiv:1812.04894 [cs]*, Dec. 2018. arXiv: 1812.04894.

[149] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), p. 54–65, Association for Computing Machinery, 2005.

[150] V. Misra, J. S. K. Reddy, and S. Chimalakonda, "Is there a correlation between code comments and issues?: an exploratory study," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, (Brno Czech Republic), pp. 110–117, ACM, Mar. 2020.

[151] L. Neil, S. Mittal, and A. Joshi, "Mining Threat Intelligence about Open-Source Projects and Libraries from Code Repository Issues and Bug Reports," in *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pp. 7–12, Nov. 2018.

[152] R. Kikas, M. Dumas, and D. Pfahl, "Issue dynamics in github projects," in *Proceedings of the 16th International Conference on Product-Focused Software Process Improvement - Volume 9459*, PROFES 2015, (Berlin, Heidelberg), p. 295–310, Springer-Verlag, 2015.

[153] J. Y. Yen and J. Y. YENt, "Finding the k shortest loopless paths in a network," 2007.

[154] Z. Liao, D. He, Z. Chen, X. Fan, Y. Zhang, and S. Liu, "Exploring the Characteristics of Issue-Related Behaviors in GitHub Using Visualization Techniques," *IEEE Access*, vol. 6, pp. 24003–24015, 2018. Conference Name: IEEE Access.

[155] Pandas, "pandas documentation — pandas 1.1.3 documentation," Oct. 2020.

[156] R. Kohavi, "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection," in *14th International Joint Conference on Artificial Intelligence*, (San Francisco), pp. 1137–1143, Morgan Kaufmann Publishers Inc., 1995.

[157] M. E. Magnello, "Karl pearson and the establishment of mathematical statistics," *International Statistical Review / Revue Internationale de Statistique*, vol. 77, no. 1, pp. 3–29, 2009.

[158] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 483–492, 2012.

[159] Dependabot, "Keep all your packages up to date with Dependabot - The GitHub Blog."

[160] X. Shi, X. Shao, Z. Guo, G. Wu, H. Zhang, and R. Shibasaki, "Pedestrian trajectory prediction in extremely crowded scenarios," *Sensors*, vol. 19, p. 1223, 03 2019.

[161] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural Comput.*, vol. 29, p. 2352–2449, Sept. 2017.

[162] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: a temporal graph-based representation of maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 344–348, IEEE, 2019.

[163] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into Imaging*, vol. 9, pp. 611–629, Aug 2018.

[164] P. T. Nguyen, D. Di Ruscio, A. Pierantonio, J. Di Rocco, and L. Iovino, "Convolutional neural networks for enhanced classification mechanisms of metamodels," *Journal of Systems and Software*, vol. 172, p. 110860, 2021.

[165] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo, "Cataloging github repositories," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, (New York, NY, USA), pp. 314–319, ACM, 2017.

[166] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, pp. 3–24, 2007.

[167] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 522–531, May 2013.

[168] M. J. Pazzani and D. Billsus, *Content-Based Recommendation Systems*, pp. 325–341. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[169] C. Miranda and A. 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, (Washington, DC, USA), pp. 389–392, IEEE Computer Society, 2008.

[170] Z.-D. Zhao and M.-s. 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, (Washington, DC, USA), pp. 478–481, IEEE Computer Society, 2010.

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

[172] P. Cremonesi, R. Turrin, E. Lentini, and M. 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, (Washington, DC, USA), pp. 224–231, IEEE Computer Society, 2008.

[173] C. E. Briguez, M. C. Budán, C. A. Deagustini, A. G. Maguitman, M. Capobianco, and G. R. Simari, "Argument-based mixed recommenders and their application to movie suggestion," *Expert Systems with Applications*, vol. 41, no. 14, pp. 6467 – 6482, 2014.

[174] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Softw.*, vol. 27, pp. 80–86, July 2010.

[175] J. Davis and M. Goadrich, "The Relationship Between Precision-Recall and ROC Curves," in *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, (New York, NY, USA), pp. 233–240, ACM, 2006.

[176] R. Rubei, J. Rocco, D. Di Ruscio, P. Nguyen, and A. Pierantonio, "A lightweight approach for the automated classification and clustering of metamodels," pp. 477–482, 10 2021.

[177] R. Rubei and C. Di Sipio, "Auryga: A recommender system for game tagging," 10 2021.

[178] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio, "An automated approach to assess the similarity of github repositories," *Software Quality Journal*, pp. 1–37, 2020.

[179] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, *API Method Recommendation without Worrying about the Task-API Knowledge Gap*, p. 293–304. New York, NY, USA: Association for Computing Machinery, 2018.

[180] P. T. Nguyen, C. Di Sipio, J. Di Rocco, M. Di Penta, and D. Di Ruscio, "Adversarial attacks to api recommender systems: Time to wake up and smell the coffee?," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 253–265, 2021.

[181] P. T. Nguyen, D. Di Ruscio, J. Di Rocco, C. Di Sipio, and M. Di Penta, "Adversarial machine learning: On the resilience of third-party library recommender systems," in *Evaluation and Assessment in Software Engineering*, EASE 2021, (New York, NY, USA), p. 247–253, Association for Computing Machinery, 2021.

[182] W. Oliveira, R. Oliveira, F. Castor, G. Pinto, and J. Fernandes, "Improving energy-efficiency by recommending java collections," *Empirical Software Engineering*, vol. 26, 05 2021.

[183] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 94–104, Association for Computing Machinery, 2018.

[184] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 70–80, 2020.