The Journal of Systems and Software 161 (2020) 110460

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

CrossRec: Supporting software developers by recommending third-party libraries

Phuong T. Nguyen[a], Juri Di Rocco[a], Davide Di Ruscio[a,*], Massimiliano Di Penta[b]

[a] Università degli studi dell'Aquila, L'Aquila 67100, Italy
[b] Università degli Studi del Sannio, Benevento 82100, Italy

ARTICLE INFO

Article history:
Received 1 July 2019
Revised 27 September 2019
Accepted 10 November 2019
Available online 11 November 2019

Keywords:
Mining software repositories
Recommender systems
Open Source software

ABSTRACT

When creating a new software system, or when evolving an existing one, developers do not reinvent the wheel but, rather, seek available libraries that suit their purpose. In such a context, open source software repositories contain rich resources that can provide developers with helpful advice to support their tasks. However, the heterogeneity of resources and the dependencies among them are the main obstacles to the effective mining and exploitation of the available data. In this sense, advanced techniques and tools are needed to mine the metadata to bring in meaningful recommendations. In this paper, we present CrossRec, a recommender system to assist open source software developers in selecting suitable third-party libraries. CrossRec exploits a collaborative filtering technique to recommend libraries to developers by relying on the set of dependencies, which are currently included in the project being developed. We perform an empirical evaluation to compare the proposed approach with three state-of-the-art baselines, i.e., LibRec, LibFinder, and LibCUP on three considerably large datasets. The experimental results show that CrossRec overcomes the limitation of the baselines by recommending also libraries with a specific version. More importantly, it outperforms LibRec and LibCUP with respect to various quality metrics.

1. Introduction

To facilitate the development activities, programmers frequently consult data available in Open Source System (OSS) repositories to look for reusable artifacts such as source code (Thummalapenta and Xie, 2007), documentation (Ponzanelli et al., 2014; 2016) or API elements (Nguyen et al., 2019b). Among others, finding and reusing third-party libraries are activities that programmers regularly perform during the development phase. Third-party libraries are highly valuable for development activities, since they provide programmers with several tailor-made functionalities (Li et al., 2017). Instead of programming from scratch, one only needs to search for libraries that implement the desired pieces of functionality, and embed them into the code being developed (Robillard et al., 2010). For example, third-party libraries represent a crucial element of many Android apps (Ma et al., 2016). A recent study shows that sub-packages from external libraries account for 60% of code in Android software (Wang et al., 2015).

Due to the heterogeneity of resources and their corresponding dependencies, developers need to spend a lot of effort to search for relevant items (Robillard et al., 2010). Despite the need for better supporting developers in such a task, very little work has been conducted concerning the techniques that facilitate the search for suitable libraries from OSS repositories. Most of the related studies address the issue of finding code snippets (Thummalapenta and Xie, 2007) or API function calls (Nguyen et al., 2019b; Thung et al., 2013b). Among the existing approaches, to the best of our knowledge, LibRec (Thung et al., 2013a), LibFinder (Ouni et al., 2017), and LibCUP (Saied et al. 2018) are the most advanced techniques for library recommendation to support OSS developers. LibRec has been demonstrated to be able to recommend project libraries with a high *success rate*.

In this paper we propose CrossRec, a novel approach utilizing a collaborative filtering technique (Schafer et al., 2007) to recommend third-party libraries. The technique being used is inspired to what used in e-commerce systems to predict the missing ratings of prospective items by analyzing the relationships between users and products (Linden et al., 2003). In a similar fashion, the proposed approach recommends third-party libraries based on their co-occurrence with other libraries in similar contexts.

In a nutshell, we work towards the quest for an appropriate answer for the question:

* Corresponding author.
E-mail addresses: phuong.nguyen@univaq.it (P.T. Nguyen), juri.dirocco@univaq.it (J. Di Rocco), davide.diruscio@univaq.it (D. Di Ruscio), dipenta@unisannio.it (M. Di Penta).

https://doi.org/10.1016/j.jss.2019.110460
0164-1212/

Which third-party libraries should I further include in my current project?

Thus, CrossRec aims at supporting software developers who have already included some libraries in the new projects being developed, and expect to get recommendations on which additional libraries should be further incorporated (if any). The typical usage scenario of CrossRec is the one in which a developer has already started to develop a project, or else she is evolving it. Based on the libraries used in a given development context, CrossRec exploits a collaboration-filtering mechanism to recommend further libraries that may be useful in such a context. This is done based on training data from other projects.

While approaches to recommend software components using collaborative filtering have been previously proposed in the literature (McCarey et al., 2008; Tsunoda et al., 2005), CrossRec is novel in that it recommends libraries based on multiple quality factors, i.e., *success rate, accuracy, sales diversity*, and *novelty*. In addition, an empirical evaluation conducted on three datasets curated from GitHub and Maven shows that CrossRec outperforms LibRec (Thung et al., 2013a), LibFinder (Ouni et al., 2017), and LibCUP (Saied et al., 2018) with respect to different quality indicators. Furthermore, CrossRec is more efficient as it generates recommendations in a comparably short period of time. To this end, our paper makes the following contributions:

- Introducing a representation model to describe the relationship among projects and third-party libraries in OSS repositories and to compute similarities;
- Building a collaborative filtering recommender system (CrossRec) to assist software developers with the selection of suitable third-party libraries;
- Employing a set of metrics for the measurement of quality of suggestions provided by a library recommender system, i.e., *success rate, accuracy, sales diversity*, and *novelty*. Such metrics have been chosen as they have been widely utilized in evaluating recommender systems (Robillard et al., 2010); and
- Performing an empirical study on the performance of CrossRec in comparison with three baselines using three GitHub datasets, exploiting various quality metrics.
- Last, but not least, we contribute with the CrossRec tool itself, as well as with all artifacts used in the evaluation, by publishing them online (Nguyen et al., 2019).

The rest of the paper is structured as follows. Section 2 discusses existing approaches related to the work presented in this paper. Section 3 introduces CrossRec, the proposed approach to suggest third-party libraries using a collaborative filtering recommender system. An empirical evaluation on real datasets is described in Section 4 with the experimental results being clarified in Section 5. Finally, Section 6 concludes the paper and outlines directions for future work.

2. Related work

This section discusses related work about recommender systems for software developers and, in particular, recommender systems aimed at suggesting software components or libraries.

Robillard et al. (2010) identify many emerging issues related to mining software repositories with the help of recommendation techniques. It has been shown that, besides sharing common characteristics with conventional recommender systems, the main challenge specific to recommender systems for Software Engineering is the mining of the highly technical metadata in software repositories. Furthermore, the authors also introduce various quality metrics for evaluating recommendation outcomes in Software Engineering, e.g., *precision* and *recall, confidence*, and *serendipity*. Thus, in this paper we exploited different quality metrics to evaluate the considered systems' performance.

One of the main challenges that a recommender system has to tackle is mining OSS repositories to identify similar projects, which can be evaluated and eventually reused by developers. MUDABlue (Kawaguchi et al., 2006) and CLAN (McMillan et al., 2012) are comparably similar in the way they represent software and identifiers/API in a term-document matrix to compute similarities. CLAN includes API calls for computing similarity, whereas MUDABlue integrates every word in the source code into the term-document matrix. However, CLAN has been claimed to help obtain a higher precision than that of MUDABlue as it considers only API calls to represent software systems (McMillan et al., 2012).

The problem of API learning has attracted considerable attention from the research community recently. Acharya et al. (2007) propose a framework to extract API patterns as partial orders from client code. MAPO (Mining API usage Pattern from Open source repositories) is an approach that learns API usage patterns from client projects (Zhong et al., 2009). MAPO analyzes source files to collect API usage information and groups API methods into clusters. It then mines API usage patterns from the clusters, ranks them according to their similarity with the current development context, and recommends code snippets to developers. Similarly, UP-Miner (Wang et al., 2013) recommends API usage patterns by relying on *SeqSim*, a clustering strategy that reduces patterns redundancy and improves coverage. UP-Miner exploits the BIDE algorithm (Wang and Han, 2004) to learn API frequent closed call sequences.

The current work is related to our previous research using collaborative filtering for the purpose of building recommender systems for software developers. In particular, FOCUS is a recommender system to mine API function calls and usage patterns (Nguyen et al., 2019b; 2019a). FOCUS represents mutual relationships among projects using a tensor, and exploits a collaborative filtering technique to mine API calls (Sarwar et al., 2001). The main advantage of the system is that it does not depend on any specific set of libraries to generate API calls. Furthermore, it scales well with large datasets exploiting the context-aware collaborative filtering technique that helps effectively remove irrelevant API function calls. In a recent work (Nguyen et al., 2019c), we exploited user-based and item-based collaborative filtering techniques to build a book recommender system. The experimental results led us to some interesting conclusions. In contrast to many existing studies which state that the item-based collaborative filtering technique outperforms the user-based collaborative filtering technique (Cacheda et al., 2011; Karypis, 2001; Papagelis and Plexousakis, 2005), we found out that there is no distinct winner between them. Moreover, we confirmed that the performance of a CF recommender system may be good with regards to some quality metrics, but not to some others.

Teyton *et al.* introduce LibTic, an approach to identify relevant experts of Java libraries among GitHub developers by automatically mining software repositories (Teyton et al., 2013). LibTic can also be used to identify experts to be contacted to ask for questions concerning the usage of libraries. LibFinder (Ouni et al., 2017) developed by Ouni et al. based on a multi-objective search-based algorithm is a solution to support developers in searching for "useful" libraries when they implement a new software system. Ouni *et al.* suggest that embedding library semantics in third-party library recommendation can introduce efficiency (Ouni et al., 2017). In this paper, we compare CrossRec with this approach by performing evaluation on the same dataset exploited to evaluate LibFinder.

The problem of third-party library recommendation has been firstly formulated by Thung et al. (2013a). They proposed LibRec, an approach to help developers leverage existing libraries that may be useful for a given project using a combination of rule mining

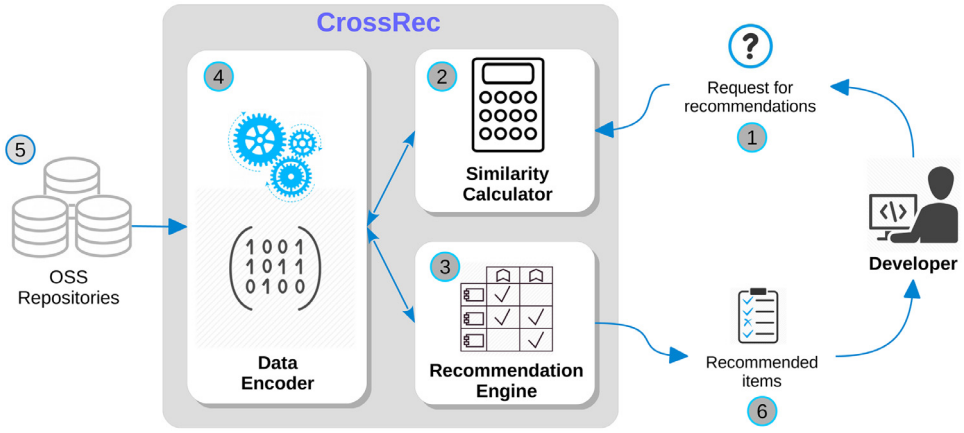


Fig. 1. Overview of the CrossRec Architecture.

and collaborative filtering techniques (Schafer et al., 2007). To compute similarity, each project is characterized by a vector using the set of libraries and the similarity between two projects is the cosine similarity between their corresponding vectors.

Our work differs from LibRec since it employs a well-defined collaborative filtering technique, which has demonstrated effectiveness in solving similar issues in other domains (Linden et al., 2003; Wu et al., 2014), to particularly address the problem of recommending third-party libraries. Meanwhile, the mechanism applied in LibRec, though referred as collaborative filtering, is actually a filtering technique that undermines the level of similarity among relevant projects. As a result, LibRec can recommend only popular libraries. In contrast, thanks to the similarity-oriented filtering technique, our approach is able to recommend highly relevant items, regardless of their popularity. Thus, as shown later on in this paper, compared to LibRec, CrossRec substantially improves the recommendation performance, e.g., not only in terms of accuracy and diversity but also in terms of novelty.

Saied et al. (2018) propose LibCUP, a recommender system that suggests libraries based on the extent to which they are used together. Specifically, LibCUP uses a clustering approach (based on the DBSCAN algorithm Ester et al., 1996) to identify and recommend library co-usage patterns. Later on in this paper, we demonstrate through the evaluation that CrossRec outperforms LibCUP both in terms of performance and the ability to recommend library versions. Also, a number of approaches have also proposed the use of collaborative filtering to support software development. In particular, Tsunoda et al. (2005) proposed Javawock, a recommender system which suggests Java classes for an unfinished program based on what is being used by similar programs. McCarey et al. (2008) also leveraged collaborative filtering mechanisms to recommend software components. Also in this case, a key difference between our proposed solution and the aforementioned approaches is that CrossRec is able to provide recommendations using a collaborative filtering approach that takes into account different quality factors, i.e., *success rate, accuracy, sales diversity*, and *novelty*.

Recently, work has also been performed with the aim of providing a crowd-sourced assessment of the quality of a library. In particular, Uddin and Khomh proposed an approach named Opiner able to identify the polarity of library-related sentences mined on Stack Overflow along different dimensions, e.g., performance, portability security, usability, or bug-proneness (Uddin and Khomh, 2017). A different approach was followed by de la Mora and Nadi (2018), which used metrics rather than opinions with the aim of comparing similar libraries.

Both approaches of Uddin and Khomh (2017) and de la Mora and Nadi (2018) can be considered as complementary to our work. Indeed, approaches like CrossRec or LibRec aim at recommending a third-party library to be used in a given context, whereas the work of Uddin and Khomh (2017) and de la Mora and Nadi (2018) can be used to better drive the choice among similar libraries.

3. Proposed approach

In this section we describe CrossRec, a system for providing developers with third-party library recommendations. More specifically, CrossRec is a *recommender system* (Aggarwal, 2016) that encodes the relationships among various OSS artifacts by means of a semantic graph and utilizes a collaborative filtering technique (Schafer et al., 2007) to recommend third-party libraries. Such a technique has been originally developed for e-commerce systems to exploit the relationships among users and products to predict the missing ratings of prospective items (Linden et al., 2003). The technique is based on the premise that *"if users agree about the quality or relevance of some items, then they will likely agree about other items"* (Schafer et al., 2007). Our approach exploits this premise to solve the problem of library recommendation (Robillard et al., 2010). Instead of recommending goods or services to customers, we recommend third-party libraries to projects using an analogous mechanism: *"if projects share some libraries in common, then they will probably share other common libraries."*

To this end, the architecture of CrossRec is shown in Fig. 1, and consists of the software components supporting the following activities:

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

In a typical usage scenario of CrossRec, we assume that a developer is creating a new system, in which she has already included some libraries, or otherwise she is evolving an existing system. As shown in Fig. 1, the developer interacts with the system by sending a request for recommendations ①. Such a request contains a list of libraries that are already included in the project the developer is working on. The `Data Encoder` ④ collects *background data* from OSS repositories ⑤, represents them in a graph, which is then used as a base for other components of CrossRec.

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

Fig. 2. An example of a user-item ratings matrix to model the inclusion of third-party libraries in OSS projects.

The `Similarity Calculator` module ② computes similarities among projects to find the most similar ones to the given project. The `Recommendation Engine` ③ implements a *collaborative-filtering* technique (Aggarwal, 2016; Zhao and Shang, 2010), it selects top-*k* similar projects, and performs computation to generate a ranked list of *top-N* libraries. Finally, the recommendations ⑥ are sent back to the developer.

Background data can be collected from different OSS platforms like GitHub (0000), Eclipse (0000), Bitbucket (0000). The current version of CrossRec (Nguyen et al., 2019) supports data extraction from GitHub, even though the support for additional platforms is under development.

In the following, the components `Data Encoder`, `Similarity Calculator`, and `Recommendation Engine` are singularly described.

3.1. Data encoder

A recommender system for online services is based on three key components, namely *users, items*, and *ratings* (Sarwar et al., 2001; Noia and Ostuni, 2015). A *user-item ratings matrix* is built to represent the mutual relationships among the components. Specifically, in the matrix a user is represented by a row, an item is represented by a column and each cell in the matrix corresponds to a rating given by a user for an item (Noia and Ostuni, 2015). For library recommendation, instead of users and items, there are projects and third-party libraries, and a project may include various libraries to implement desired functionalities. We derive an analogous user-item ratings matrix to represent the *project-library inclusion* relationships, denoted as ∋. In this matrix, each row represents a project and each column represents a library. A cell in the matrix is set to 1 if the library in the column is included in the project specified by the row, it is set to 0 otherwise. For the sake of clarity and conformance, we still denote this as a user-item ratings matrix throughout this paper.

For explanatory purposes, we consider a set of four projects $P = \{p_1, p_2, p_3, p_4\}$ together with a set of libraries $L = \{lib_1$=*junit:junit; *lib_2=*commons-io:commons-io; *lib_3=*log4j:log4j; *lib_4=*org.slf4j:slf4j-api; *lib_5=*org.slf4j:slf4j-log4j12*$\}$. By observing the *pom.xml* files of the projects in *P*, we discovered the following inclusions: $p_1 \ni lib_1$, lib_2; $p_2 \ni lib_1$, lib_3; $p_3 \ni lib_1$, lib_3, lib_4, lib_5; $p_4 \ni lib_1$, lib_2, lib_4, lib_5. Accordingly, the user-item ratings matrix built to model the occurrence of the libraries is depicted in Fig. 2.

3.2. Similarity calculator

The `Recommendation Engine` of CrossRec works by relying on an analogous user-item ratings matrix. To provide inputs for this module, the first task of CrossRec is to perform similarity computation on its input data to find the most similar projects to a given project. In this respect, the ability to compute the similarities among projects has an effect on the recommendation outcomes. Nonetheless, computing similarities among software systems is considered to be a difficult task (McMillan et al., 2012). In addition, the diversity of artifacts in OSS repositories as well as their cross relationship makes the similarity computation become even more complicated. In OSS repositories, both humans (i.e., developers and users) and non-human actors (such as repositories, and libraries) have mutual dependency and implication on the others. The interactions among these components, such as developers commit to or star repositories, or projects include libraries, create a tie between them and should be included in similarity computation.

We assume that a representation model that addresses the semantic relationships among miscellaneous factors in the OSS community is beneficial to project similarity computation. To this end, we consider the community of developers together with OSS projects, libraries, and their mutual interactions as an *ecosystem*. We derive a *graph-based* model to represent different kinds of relationships in the OSS ecosystem, and eventually to calculate similarities. In the context of mining OSS repositories, the graph model is a convenient approach since it allows for flexible data integration and numerous computation techniques. By applying this representation, we are able to transform the set of projects and libraries shown in Fig. 2 into a directed graph as in Fig. 3. We adopted our proposed CrossSim approach

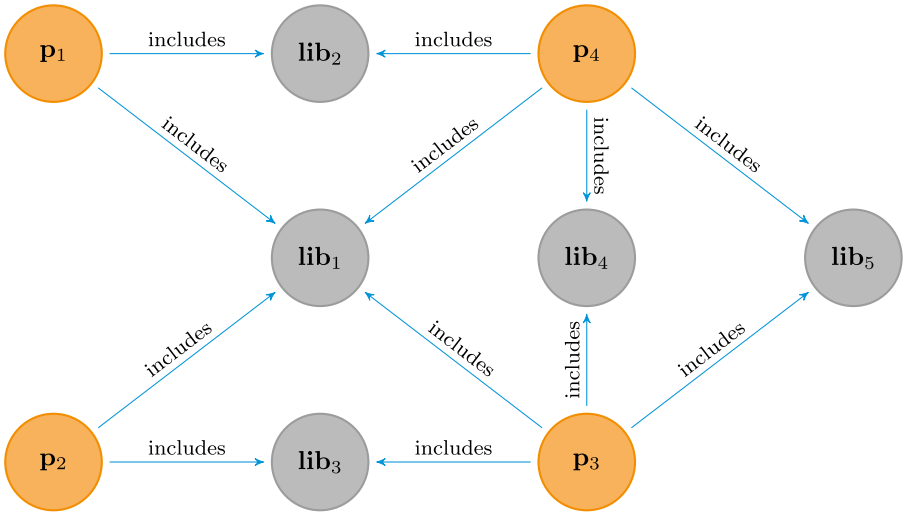


Fig. 3. Graph representation for projects and libraries.

(Nguyen et al., 2019b; 2018) to compute the similarities among OSS graph nodes. It relies on techniques successfully exploited by many studies to do the same task (Di Noia et al., 2012; Briguez et al., 2014). Among other relationships, two nodes are deemed to be similar if they point to the same node with the same edge. By looking at the graph in Fig. 3, we can notice that p_3 and p_4 are highly similar since they both point to three nodes lib_1, lib_4, lib_5. This reflects what also suggested in a previous work by McMillan et al. (2012), i.e., similar projects implement common pieces of functionality by using a shared set of libraries.

Using this metric, the similarity between two project nodes p and q in an OSS graph is computed by considering their feature sets (Di Noia et al., 2012). Given that p has a set of neighbor nodes $(lib_1, lib_2, .., lib_l)$, the features of p are represented by a vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_i being the weight of node lib_i. It is computed as the *term-frequency inverse document frequency* value as follows:

$$\phi_i = f_{lib_i} \times \log\left(\frac{|P|}{a_{lib_i}}\right) \tag{1}$$

where f_{lib_i} is the number of occurrence of lib_i with respect to p, it can be either 0 and 1 since there is a maximum of one lib_i connected to p by the edge *includes*; $|P|$ is the total number of considered projects; a_{lib_i} is the number of projects connecting to lib_i via the edge *includes*. Eventually, the similarity between p and q with their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_i\}_{i=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$ is computed as given below:

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

where n is the cardinality of the set of libraries that p and q share in common (Di Noia et al., 2012). Intuitively, p and q are characterized by using vectors in an n-dimensional space, and Eq. (2) measures the cosine of the angle between the two vectors.

3.3. Recommendation engine

The representation using a user-item ratings matrix allows for the computation of missing ratings (Aggarwal, 2016; Noia and Ostuni, 2015). Depending on the availability of data, there are two main ways to compute the unknown ratings, namely *content-based* (Pazzani and Billsus, 2007) and *collaborative-filtering* (Miranda and Jorge, 2008) recommendation techniques. The former exploits the relationships among items to predict the most similar items. The latter computes the ratings by taking into account the set of items rated by similar customers. There are two main types of collaborative-filtering recommendation: *user-based* (Zhao and Shang, 2010) and *item-based* (Sarwar et al., 2001) techniques. As their names suggest, the user-based technique computes missing ratings by considering the ratings collected from similar users. Instead, the item-based technique performs the same task by using the similarities among items (Cremonesi et al., 2008).

In the context of CrossRec, the term *rating* is understood as the occurrence of a library in a project and computing missing ratings means to predict the inclusion of additional libraries. The project that needs prediction for library inclusion is called the *active project*. By the matrix in Fig. 4, p is the active project and an asterisk (*) represents a known rating, either 0 or 1, whereas a question mark (?) represents an unknown rating and needs to be predicted.

Given the availability of the cross-relationships as well as the possibility to compute similarities among projects using the graph representation, we exploit the user-based collaborative-filtering technique as the engine for recommendation (Linden et al., 2003; Zhao and Shang, 2010). Given an active project p, the inclusion of libraries in p can be deduced from projects that are similar to p. The process is summarized as follows:

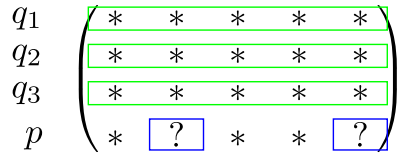


Fig. 4. Computation of missing ratings using the user-based collaborative-filtering technique (Zhao and Shang, 2010).

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

The rectangles in Fig. 4 imply that the row-wise relationships between the active project p and the similar projects q_1, q_2, q_3 are exploited to compute the missing ratings for p. The following formula is used to predict if p should include l, i.e., $p \ni l$ (Noia and Ostuni, 2015):

$$r_{p,l} = \overline{r_p} + \frac{\sum_{q \in topsim(p)} (r_{q,l} - \overline{r_q}) \cdot sim(p,q)}{\sum_{q \in topsim(p)} sim(p,q)} \tag{3}$$

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 Eq. (2).

3.4. CrossRec in action

In the context of the EU H2020 CROSSMINER project[1], we integrated CrossRec into Eclipse as shown in Fig. 5. The figure describes a development of an explanatory scenario where a developer is improving a software project, named *aethereal* hereafter, by replacing some existing source code with features provided by third-party libraries. The goal of such changes is to make the code easier to be understood and evolved. The project is a command line tool written in Java that aims at supporting the automatic generation of cross-projects migration dependencies graph. To this aim *aethereal* distinguishes between clients and libraries: both clients and libraries are Maven artifacts, and clients are defined as artifacts that use a specific library available on the Maven repository.

We explain how CrossRec helps the developer evolve the `build` method of the class `MavenDataset` as follows. Such a method computes a dependency matrix between client versions and libraries. The initial implementation of the `build` method prints the logging information to the console by using Java I/O facilities (i.e., `System.out.println` and `System.err.println`) ①. The CrossRec tool displays the list of third-party libraries currently being included ②, and prompts a list of recommended libraries ③. The list of recommended libraries highlights the suggestion for using a logging library, i.e., `slf4j`[2] or `commons-logging`[3]. Then, a possible migration to `slf4j` is shown ④, where the `System.out.println` and `System.err.println` invocations are replaced by `slf4j` `logger.info` and `logger.debug` calls, respectively.

[1] https://www.crossminer.org.
[2] https://www.slf4j.org/.
[3] https://commons.apache.org/proper/commons-logging/.

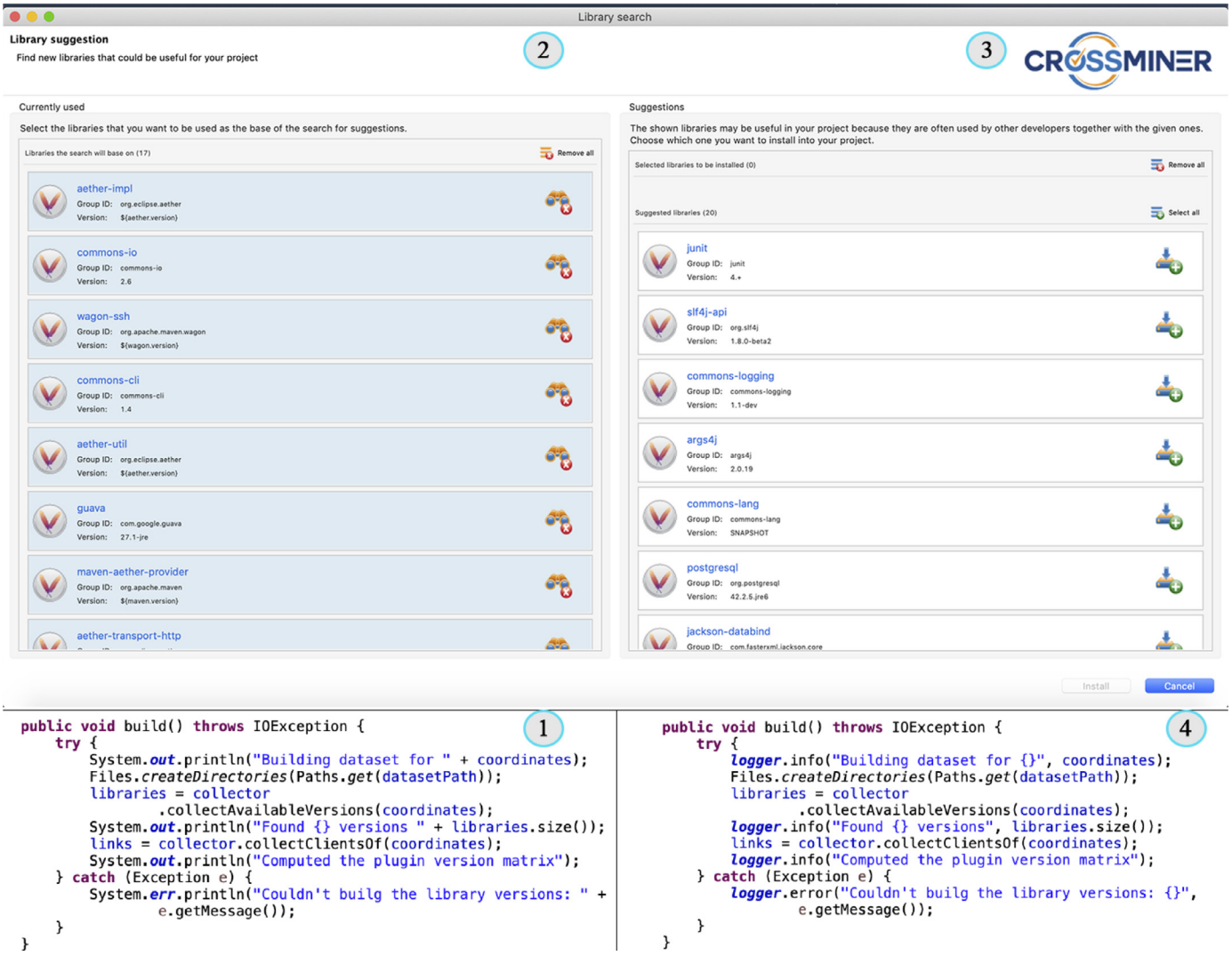


Fig. 5. CrossRec IDE.

In its current version, CrossRec recommends a set of libraries to developers, however it is likely that only some of the suggestions will result as useful for them. While the current version of CrossRec is unable to capture developers' feedback, it is possible to re-train again and obtain new recommendations based on the up-to-date code base.

4. Evaluation

This section describes the planning of our evaluation, having the *goal* of evaluating the performance of CrossRec, and of comparing it with three state-of-the-art approaches, namely LibRec (Thung et al., 2013a), LibFinder (Ouni et al., 2017), and LibCUP (Saied et al., 2018). In Section 4.1, we introduce the datasets exploited in our evaluation. The evaluation methodology and metrics are presented in Section 4.2. Finally, Section 4.3 describes the research questions.

The evaluation process is depicted in Fig. 6 and it consists of three consecutive phases, i.e., *Data Preparation*, *Recommendation*, and *Outcome Evaluation*. We start with the *Data Preparation* phase by creating a dataset from GitHub projects. Such datasets are used to assess the performances of CrossRec, but also to evaluate two of the baselines, i.e., LibFinder and LibCUP. Each dataset is then split into training and testing sets. In the *Recommendation* phase, given a project in the testing set, a portion of its libraries is extracted as ground-truth data, and the remaining is used as query to compute similarity and recommendations. Finally, by the *Outcome Evaluation* phase, we compare the recommendation results with those stored as ground-truth data to compute the quality metrics. All the aforementioned phases are detailed in the rest of this section. The source code implementation of CrossRec, the dataset as well as the experimental results are available for download at https://www.doi.org/10.5281/zenodo.1285620.

4.1. Dataset extraction

To evaluate the systems, we exploited three independent datasets as follows.

Dataset D1 This first dataset is used to assess the performances of CrossRec on a large set of projects, and also, to compare CrossRec with LibRec. As it will be clearer later, other baseline tools are not available, and we had to use different datasets for comparison purposes.

By means of the GitHub API (GitHub REST API v3, 2019) we *randomly* collected a dataset consisting of 1,200 Java projects. Among these projects, only 7 of them have been forked from other projects. Such original projects have been excluded from the dataset as their forked ones share highly similar libraries, and this

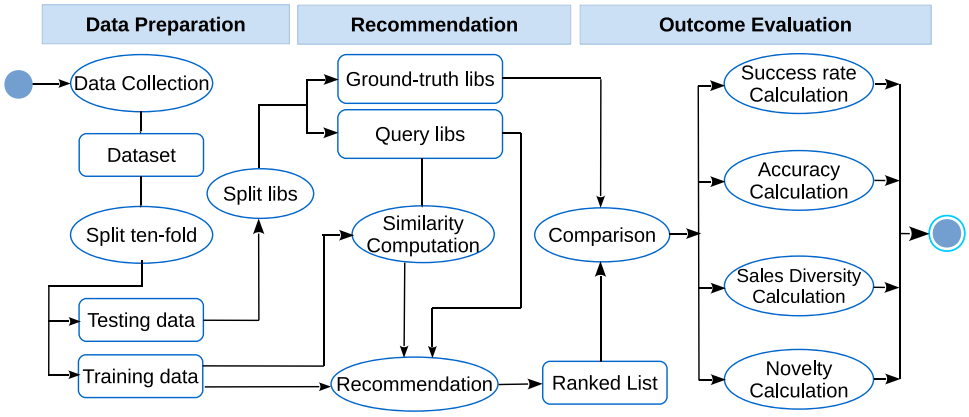
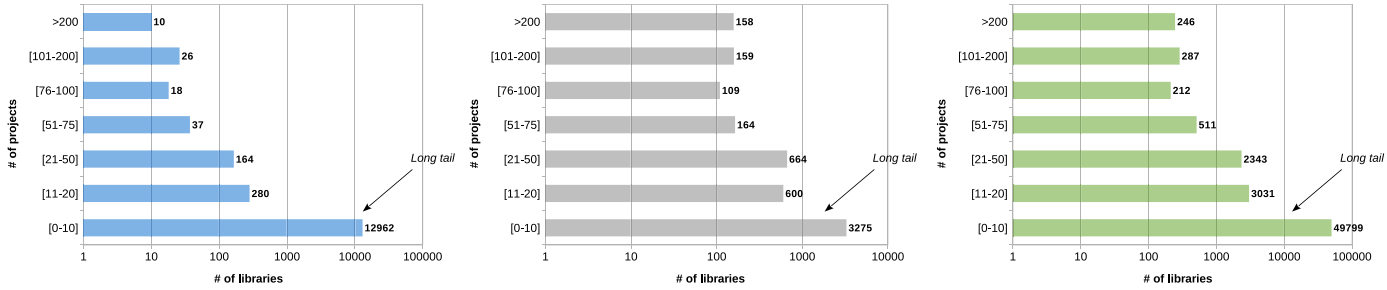


Fig. 6. Evaluation Process.



(a) Dataset D1 (b) Dataset D2 (c) Dataset D3

Fig. 7. The distribution of libraries.

may introduce bias in the recommendation outcomes. We represent the distribution of projects with respect to the number of forks, commits and pull requests in Fig. 8(a). Most projects have a low number of pull requests, i.e., lower than 100, however many of them have a large number of forks and commits. Forking is a means to contribute to the original repositories Jiang et al. (2017). Furthermore, there is a strong correlation between forks and stars Borges et al. (2016), as it is further witnessed in Fig. 8(b). A project with a high number of forks means that it gets attention from the OSS community. In this sense, having many forks can be considered as a sign of a well-maintained and well-received project. Meanwhile, as commits have an impact on the source code Behnamghader et al. (2017), the number of commits is also a good indicator of how a project has been developed.

We mined dependency specification by means of `code.xml` or `.gradle` files.[4] Fig. 7(a) depicts the distribution of libraries across the projects. Most of the libraries in D1 (12,962) are used by a small number of projects, and only 10 libraries are extremely popular by being included in more than 200 projects. By carefully investigating the dataset, we also see that most projects contain a small number of dependencies, i.e., 48% of the projects include less than 20 libraries and just 15% of them include more than 100 libraries.

Dataset D2

To compare CrossRec with LibFinder, we could neither follow the same process utilized to evaluate LibRec, nor use the same D1 dataset, because the LibFinder tool was not available. Therefore, we analyzed the dataset from the original LibFinder paper Ouni et al. (2017) and named it D2. The original dataset consists of 29,653 Github projects with 5,129 Maven libraries.[5] Fig. 7(b) gives an overview on the distribution of dependencies among projects. A large number of libraries, i.e., 3,275 items are used by only 10 projects, whereas 158 libraries are more popular by being included in more than 200 projects. Besides using D2 to compare CrossRec with LibFinder, we use it to investigate the extent to which CrossRec can recommend specific versions of libraries.

Dataset D3 As for LibFinder, also LibCUP was not available, therefore we had to compare CrossRec and LibCUP using the original dataset[6] by Saied et al. (2018). Such a dataset features 90,475 GitHub projects and 56,435 libraries. The distribution of dependencies among projects is depicted in Fig. 7(c). Similar to D1 and D2, most of the libraries in D3 are included in a very small number of projects, and just only 246 libraries (0.4%) are found in more than 200 projects.

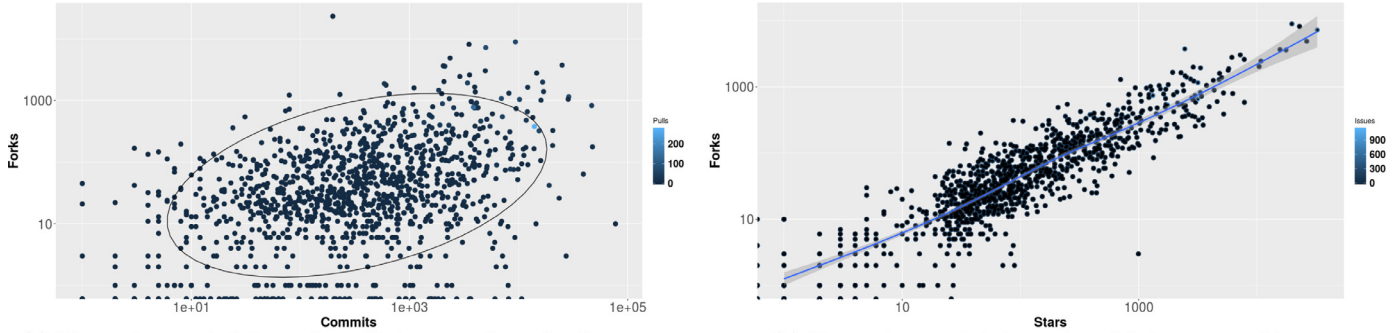
4.2. Evaluation methodology and Metrics' definition

To assess the performance of the four systems, i.e., LibRec, LibFinder, LibCUP, and CrossRec, we applied ten-fold cross-validation, considering every time 9 folds (each one contains 120 projects) for training and the remaining one for testing. For every testing project p, a half of its libraries are *randomly* taken out and saved as ground truth data, let us call them $GT(p)$, which will be used to validate the recommendation outcomes. The other half are used as testing libraries or query, which are called ***te***, and serve as input for `Similarity Computation` and `Recommendation`. Though the ratio of query to ground-truth data can be arbitrarily set, we chose 50% for two reasons: *(i)* this value was used in

[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.

[5] http://sel.ist.osaka-u.ac.jp/people/ali/libRecommendation/dependencyFactSet.csv.

[6] https://github.com/saiedmoh/LibCUP/blob/master/githubsnapshot.zip.



(a) The projects and their number of forks, commits and pull requests

(b) The projects and their number of forks, stars and issues

Fig. 8. Statistics for Dataset D1.

the evaluation of the baselines in the original work (Thung et al., 2013a; Ouni et al., 2017; Saied et al., 2018); and *(ii)* no matter how big a query is, the most important thing is to use exactly the same amount of data when evaluating the systems. In a separate experiment discussed later in this paper, the ratio of query to ground-truth has been varied to investigate whether an increase in the amount of testing data helps improve CrossRec's overall performance. The splitting mimics a real development process where a developer has already included some libraries in the current project, i.e., ***te*** and waits for recommendations, that means additional libraries to be incorporated. A recommender system is expected to provide her with the other half, i.e., *GT(p)*. To ensure a reliable comparison between LibRec and CrossRec, we performed cross-validation for both using exactly the same folds.

There are several metrics available to evaluate a ranked list of recommended items (Noia and Ostuni, 2015). In the scope of this paper, *success rate, accuracy, sales diversity*, and *novelty* have been used to study the systems' performance as already proposed by Robillard et al. (2010) and other studies (Thung et al., 2013a; Nguyen et al., 2015). For a clear presentation of the metrics considered during the outcome evaluation, let us introduce the following notations:

- N is the cut-off value for the ranked list;
- k is the number of neighbor projects exploited for the recommendation process;
- For a testing project p, a half of its libraries are extracted and used as the ground-truth data named as $GT(p)$;
- $REC(p)$ is the *top-N* libraries recommended to p. It is a ranked list in descending order of real scores;
- If a recommended library $l \in REC(p)$ for a testing project p is found in the ground truth of p (i.e., $GT(p)$), hereafter we call this as a library *match* or *hit*.

If $REC_N(p)$ is the set of top-N items and $match_N(p) = GT(p) \bigcap REC_N(p)$ is the set of items in the *top-N* list that match with those in the ground-truth data, then the metrics are defined as follows.

Success rate@N Given a set of testing projects P, this metric measures the rate at which a recommender system returns at least a library match among *top-N* items for every project $p \in P$ (Thung et al., 2013a):

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} \tag{4}$$

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

Accuracy Accuracy is considered as one of the most preferred *quality indicators* for Information Retrieval applications (Saracevic, 1995). However, *success rate@N* does not reflect how accurate the outcome of a recommender system is. For instance, given only one testing project, there is no difference between a system that returns 1 library match out of 5 and another system that returns all 5 library matches, since *success rate@5* is 100% for both cases (see Eq. (4)). Thus, given a list of *top-N* libraries, *precision@N* and *recall@N* are utilized to measure the *accuracy* of the recommendation results. *precision@N* is the ratio of the *top-N* recommended libraries belonging to the ground-truth dataset, whereas *recall@N* is the ratio of the ground-truth libraries appearing in the N recommended items (Nguyen et al., 2019b; Di Noia et al., 2012; Davis and Goadrich, 2006):

$$precision@N = \frac{|match_N(p)|}{N} \tag{5}$$

$$recall@N = \frac{|match_N(p)|}{|GT(p)|} \tag{6}$$

Sales diversity This term originates from the business domain where it is important to improve the coverage as also the distribution of products across customers, thereby increasing the chance that products will get sold by being introduced (Vargas and Castells, 2014). Similarly, in the context of library recommendation, *sales diversity* indicates the ability of the system to suggest to projects as much libraries as possible, as well as to disperse the concentration among all items, instead of focusing only on a specific set of libraries (Robillard et al., 2010). In the scope of this paper, *catalog coverage* and *entropy* are utilized to gauge *sales diversity*. Let L be the set of all libraries available for recommendation, $\#rec(l)$ denote the number of projects getting library l as a recommendation, i.e., $\#rec(l) = count_{p \in P}(|REC_N(p) \ni l|)$, $l \in L$, and *total* denote the number of recommended items across all projects.

Catalog coverage measures the percentage of libraries being recommended to projects:

$$coverage@N = \frac{\left|\cup_{p \in P} REC_N(p)\right|}{|L|} \tag{7}$$

Entropy evaluates if the recommendations are concentrated on only a small set or spread across a wide range of libraries:

$$entropy = -\sum_{l \in L}\left(\frac{\#rec(l)}{total}\right) ln\left(\frac{\#rec(l)}{total}\right) \tag{8}$$

Novelty In business, the *long tail effect* is the fact that a few of the most popular products are extremely popular, while the rest, so-called the long tail, is obscure to customers (Anderson, 2006). Recommending products in the long tail is beneficial not only to customers but also to business owners (Vargas and Castells, 2014). Given that the logarithmic scale is used instead of the decimal one on the x-Axis of Fig. 7(a)–7(c), the long tail effect can be spotted there as a few libraries are very popular by being included in several projects, whereas most of the libraries appear in fewer projects. Specifically, in D1 just 10 libraries are used by more than

200 projects, whereas 12,962 libraries are used by no more than 10 projects. In D2 there are 3,275 libraries, accounting for 63.85% of the total number, being used by no more than 10 projects. Similarly, in D3, 88.24% of the dependencies corresponding to 49,799 libraries are used by no more than 10 projects. When recommending a library, *novelty* measures if a system is able to pluck libraries from the "long tail" to expose them to projects. This might help developers come across *serendipitous* libraries (Ge et al., 2010), e.g., those that are seen by chance but turn to be useful for their current project. To quantify *novelty*, we utilize *expected popularity complement* (EPC) which is defined in the following equation (Vargas and Castells, 2014):

$$EPC@N = \frac{\sum_{p \in P} \sum_{r=1}^{N} \frac{rel(p,r) * [1 - pop(REC_r(p))]}{log_2(r+1)}}{\sum_{p \in P} \sum_{r=1}^{N} \frac{rel(p,r)}{log_2(r+1)}} \quad (9)$$

where $rel(p, r) = |GT(p) \bigcap REC_r(p)|$ represents the relevance of the library at the r position of the *top-N* list to project p, $rel(p, r)$ can either be 0 or 1; $pop(REC_r(p))$ is the popularity of the library at the position r in the *top-N* recommended list. It is computed as the ratio between the number of projects that receive $REC_r(p)$ as a recommendation and the number of projects that receive the most ever recommended library as a recommendation. Eq. (9) implies that the more unpopular libraries a system recommends, the higher the EPC value it obtains and vice versa.

When comparing results of LibRec and CrossRec, we rely on suitable statistical procedures. Specifically, we use the Wilcoxon ranked sum test (considering one data point for each fold of the cross-validation) with a significance level of $\alpha = 5\%$, and the Cliff's delta (d) effect size measure (Grissom and Kim, 2005). Due to multiple tests being performed, we adjust p-values using the Holm's correction (Holm, 1979).

4.3. Research questions

As mentioned in Section 4.1, a direct comparison between CrossRec and LibFinder or LibCUP is difficult to perform, owing to the fact that there is no source code available for public use, as well as their re-implementation might not be compliant with the original work. Thus, we evaluate the recommendations by LibFinder, LibCUP and CrossRec by experimenting CrossRec on the corresponding datasets. Furthermore, we opt for *success rate* as it is the common metric used in the original work (Thung et al., 2013a; Ouni et al., 2017; Saied et al., 2018). In contrast, the full access to its original implementation[7] allows us to painstakingly compare LibRec with CrossRec, making use of all the metrics presented in this paper, i.e., *success rate, accuracy, sales diversity* and *novelty*.

In this sense, our empirical evaluation aims at addressing the following research questions:

- **RQ$_1$**: *How does CrossRec compare with the state-of-the-art approaches in terms of success rate?*We evaluate CrossRec by considering three state-of-the-art approaches, i.e., LibRec, LibFinder, and LibCUP, exploiting the datasets presented in Section 4.1. Furthermore, we study CrossRec to see if it can recommend a specific version of a library, a requirement that all the baselines considered in this paper cannot satisfy.
- **RQ$_2$**: *How well can LibRec and CrossRec recommend third-party libraries with respect to accuracy, sales diversity, and novelty?*Apart from *success rate*, we investigate if the approaches obtain a good performance in terms of *accuracy, sales diversity* and *novelty* (Robillard et al., 2010; Nguyen et al., 2015).
- **RQ$_3$**: *What are the reasons for the performance difference between LibRec and CrossRec?*We are interested in understanding the rationale behind the performance differences between the two systems.
- **RQ$_4$**: *Does an increase in the amount of input data help improve CrossRec's overall performance?*We suppose that feeding CrossRec with more data as query enhances the overall performance. Thus, rather than using $r = 50\%$ as the ratio of query to testing data, we varied r using incremental values, i.e., 20%, 40%, 60%, and 80% to analyze the change in performance.

We study the experimental results in the next section by referring to these research questions.

5. Results

This section reports and discusses the results of our study by addressing the research questions formulated in Section 4. The section is structured into the following subsections. Section 5.1 introduces an example of the recommendation results by LibRec and CrossRec. In Section 5.2, we analyze the obtained results to study the systems' performance. Finally, Section 5.3 discusses the probable threats to validity of our findings.

5.1. Explanatory example

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

Both LibRec and CrossRec obtain a *success rate@10=1.0*. However, CrossRec has a better *recall@10* compared to LibRec as it returns more relevant items (see Eq. (6)). Furthermore, among the matches by CrossRec, 4 items appear in the top rows of the ranked list, indicating that CrossRec recommends with a high *precision@N* (see Eq. (5)). LibRec returns only 1 relevant item, which means that both *precision@N* and *recall@N* are considerably lower compared to those of CrossRec. Furthermore, LibRec tends to suggest very popular libraries: 6 out of 10 items recommended by LibRec are used by more than 200 projects. For instance, besides *junit:junit*, the second highest frequency item is *org.slf4j:slf4j-api* (473/1, 200). By performing an investigation on the outcome of all queries, we realized that LibRec usually recommends very popular items. The reasons for such differences are explained in Section 5.2.

Four out of five items recommended by CrossRec have a low frequency of occurrence. For instance, the first item in the ranked list is *com.badlogicgames. gdx:gdx-platform* and this library is included in only 3/1,200 projects. Referring to Fig. 7(c), it is evident that the top 3 items belong to the long tail, i.e., they are extremely unpopular since each is used by only 3 projects. However, they turn out to be useful as all of them match those stored

[7] We gratefully acknowledge the LibRec source code implementation provided by Ferdian Thung and David Lo at the School of Information Systems, Singapore Management University.

Table 1
Recommendation results (matching with ground truth in bold face) for *peakgames/libgdx-stagebuilder*.

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

as ground-truth. In contrast to some existing studies which choose to recommend only popular items to developers (Ponzanelli et al., 2014; Moreno et al., 2015) we see that popularity is not a good indicator for selecting a library. This implies that the novelty of a ranked list is important: a system should be able to recommend libraries that are *novel* (Castells et al., 2011), i.e., those that have been rarely seen. In this sense, we expect that CrossRec can produce good outcomes, not only in terms of success rate and accuracy, but also sales diversity and novelty.

In summary, for the explanatory example, CrossRec obtains a comparable success rate, but better accuracy and novelty than LibRec. This also confirms that success rate is not sufficient for evaluating the recommendation outcomes. A good recommender system is the one that can maintain a trade-off by improving diversity, novelty but still retaining a good accuracy (Ragone et al., 2017). Consequently, it is necessary to investigate if this trade-off is guaranteed by LibRec and CrossRec, and this is done in the next subsections by considering the whole dataset discussed in the previous section.

5.2. Empirical study results

In this section, we report the results by addressing the research questions **RQ_1, RQ_2, RQ_3, and RQ_4**.

RQ_1: *How does CrossRec compare with the state-of-the-art approaches in terms of success rate?*

Comparison between LibRec and CrossRec We performed a series of experiments on D1 using different combinations of number of recommended libraries (i.e., N), and number of neighbor projects exploited in the recommendation phase (i.e., k). Varying N means changing the length of the recommendation list, whereas increasing k means considering more neighbor projects for recommendation.

Fig. 9 (a) shows the *success rate@5* for k={5, 10, 15, 20, 25}. As it can be seen there, the success rate values obtained by CrossRec are always superior to those of LibRec. The maximum *success rate@5* of LibRec is 0.876, whereas CrossRec obtains success rates being greater than 0.903 for all configurations, with 0.931 being the maximum value. Fig. 9(b) shows the *success rate@10*, for this setting, LibRec gains a comparable performance for $N = 5$. Meanwhile, CrossRec achieves a slight improvement in its performance compared to the case with $N = 5$. It is evident that CrossRec outperforms LibRec in all test configurations. Fig. 9(a) and 9(b) imply that changing the number of neighbor projects k does not make a substantial difference in their match rate, as *success rate@N* is stable towards k for both systems.

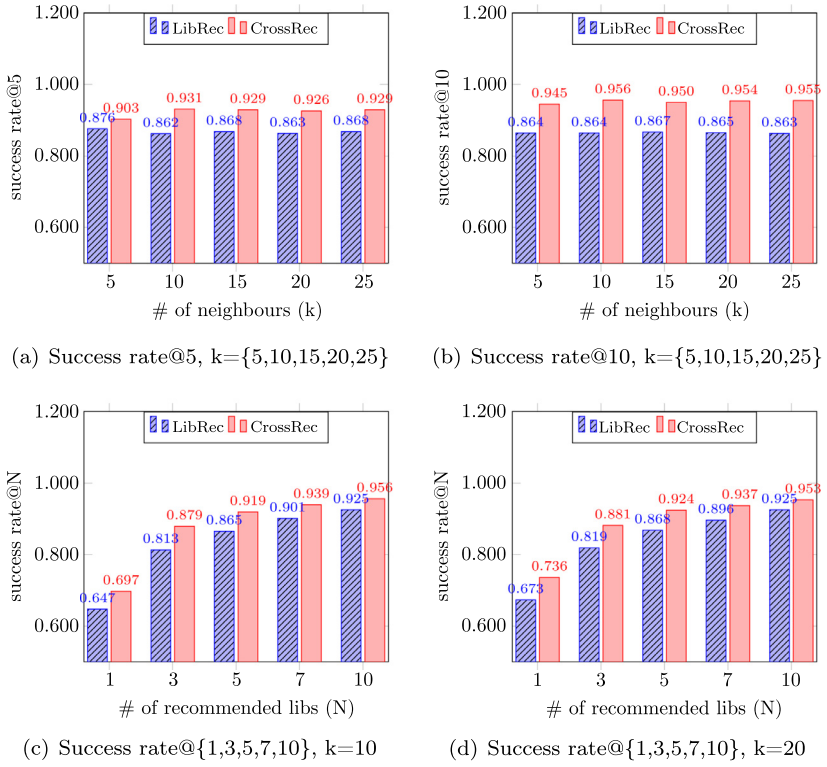
Table 2
Wilcoxon rank sum test adjusted p-values and Cliff's d results for N={3,5,10,15}, k=10.

Test	Cut-off value (N)			
	3	5	7	10
Wilcoxon r.s.t. adjusted p-values	0.02	0.02	0.17	0.002
Cliff's d results	0.70 (l)	0.74 (l)	0.93 (l)	0.58 (l)

Next, we investigate the success rate with regards to N. We consider a small number of recommended items, i.e., $N = \{1, 3, 5, 7, 10\}$. In practice, this means that the developer wants to see a short list of recommended libraries. In the first experiment, k is fixed to 10 and the outcomes are depicted in Fig. 9(c). For $N = 1$, LibRec achieves a success rate of 0.647 which is lower than the corresponding value 0.697 produced by CrossRec. A success rate of 0.697 implies that CrossRec can supply relevant recommendations to the developer at an encouraging match rate, even when she expects only an extremely brief list. Once k is changed from 10 to 20, both systems have a slight increase in *success rate@1* as depicted in Fig. 9(d). However, for other values of N, there are almost no changes in success rate. To further observe this behavior, we conducted more experiments with an increasing k, e.g., $k = \{50, 60, 100\}$. As far as we can see, there are no subtle differences between the conclusions obtained from the new experiments with those previously presented in the paper. Thus, for the sake of clarity, the outcomes of these experiments are omitted from the paper.

Table 2 reports Wilcoxon rank sum test adjusted p-values and Cliff's d, respectively for the comparison of LibRec and CrossRec in terms of *success rate*, using $k = 10$ and $N = \{3, 5, 10, 15\}$; the labels in parentheses indicate the magnitude (n:negligible, s:small, l:large). As the table shows, the differences are always statistically significant and in favor of CrossRec (effect size is positive), with a large effect size.

In summary, CrossRec significantly outperforms LibRec in all considered test configurations concerning *success rate*, with a large effect size. The recommendation time for a fold (120



(a) Success rate@5, k={5,10,15,20,25} (b) Success rate@10, k={5,10,15,20,25}

(c) Success rate@{1,3,5,7,10}, k=10 (d) Success rate@{1,3,5,7,10}, k=20

Fig. 9. Success rate of CrossRec and LibRec on D1.

Table 3
Success rate of LibFinder and CrossRec on D2.

System	Cut-off value (N)					
	1	2	4	6	8	10
LibFinder	0.633	0.698	0.813	**0.876**	**0.904**	**0.918**
CrossRec	**0.771**	**0.816**	**0.851**	0.860	0.863	0.864

Table 4
Success rate of CrossRec on D2 with different number of libraries (*L*).

		Cut-off value (N)					
	# of libraries	1	2	4	6	8	10
L	4	0.811	0.853	0.894	0.906	0.911	0.914
	6	0.890	0.926	0.948	0.958	0.961	0.964
	8	0.930	0.958	0.972	0.977	0.982	0.986
	10	0.938	0.964	0.976	0.981	0.984	0.987

projects) is relatively faster for CrossRec (3s) than for LibRec (20s).

Comparison between LibFinder and CrossRec Even though LibFinder is a bit different from CrossRec as it aims at providing *on-the-fly* recommendations while the developer is coding by exploiting the existing semantic, we attempted to compare CrossRec with LibFinder by applying the same experimental settings. A comparison between LibFinder and CrossRec is depicted in Table 3.[8] For small cut-off values, i.e., $N = \{1, 2, 4\}$, CrossRec obtains a better success rate than LibFinder does. However, by the higher values $N = \{6, 8, 10\}$, LibFinder gains an improvement in its performance. This suggests that LibFinder tends to provide matches quite late in the ranked list. On one hand, a system with such a recommendation list helps developers approach relevant libraries. On the other hand, this is not useful for those who prefer to get only a short list of recommendations.

By carefully examining the D2 dataset, we realized that there are several projects containing a small number of dependencies, e.g., 1 or 2 third-party libraries. Such projects are not beneficial to the training of CrossRec since the corresponding metadata is not sufficient to compute similarity and, consequently, to provide a recommendation (see Eqs. (1) and (2)). Thus, to further investigate the performance of CrossRec on D2, we performed a filtering step as follows. Starting from the dataset, we selected projects containing a certain number of libraries, and such a number is called as *L*. The number was then varied to create different configurations. In practice, this means that we consider only mature projects, i.e., those that include a decent number of libraries for training. Table 4 shows the success rate obtained by varying *L*. As it can be seen, CrossRec's performance is proportional to *L*: the more dense (with respect to dependencies) the projects are, the better performance CrossRec achieves. For instance, when we consider only projects with at least 4 libraries, i.e., $L = 4$ the success rate obtained for $N = 10$ is 0.914. However, if *L* is increased to 10, the corresponding success rate improves to reach 0.987. The same trend can be witnessed by other combinations of *L* and *N*.

While the D2 dataset also specifies library versions, LibFinder is unable to deal with this information, resulting in a limitation, as already stated by the authors (Ouni et al., 2017). In practice, it is crucial to provide developers with not only a suitable library, but also a specific version of that library, otherwise there might be some issues related to version compatibility (Mileva et al., 2009) or, in general, recommending obsolete libraries. Thus, we injected also library version into the training and testing data and fed it to CrossRec. The final results for different configurations are reported

[8] The success rates of LibFinder were extracted directly from the paper (Ouni et al., 2017).

Table 5
Success rate of CrossRec on D2 when library version is considered.

	# of libraries	Cut-off value (N) 1	2	4	6	8	10
L	2	0.620	0.676	0.715	0.729	0.735	0.738
	4	0.785	0.834	0.872	0.893	0.903	0.907
	6	0.848	0.885	0.907	0.921	0.931	0.936
	8	0.888	0.919	0.931	0.937	0.945	0.950
	10	0.906	0.930	0.944	0.948	0.951	0.955

Table 6
Success rate of LibCUP and CrossRec on D3.

System	Cut-off value (N) 1	3	5	7	10
LibCUP	0.12	0.14	0.15	0.17	0.22
CrossRec	**0.21**	**0.31**	**0.36**	**0.39**	**0.42**

in Table 5. As it can be seen in the table, given that decent training data is available, CrossRec also recommends libraries with version, achieving a high success rate. For instance, even when projects containing at least 2 libraries are considered, or $L = 2$, CrossRec obtains a success rate of 0.620 for $N = 1$. This quality metric improves linearly alongside L and N. As an example, when $L = 10$ and $N = 10$, the corresponding success rate is 0.955.

On the D2 dataset, CrossRec obtains a comparable performance compared to that of LibFinder. When it comes to a dataset containing projects with more libraries, CrossRec is able to give more precise recommendations. Differently from LibFinder, CrossRec is capable recommending also a specific version of a library.

Comparison between LibCUP and CrossRec For this evaluation, we used the same experimental settings utilized to evaluate LibFinder. In particular, the following cut-off values have been considered: $N = \{1, 3, 5, 7, 10\}$. According to Table 6, the performance obtained by CrossRec is always better than that of LibCUP.[9] For example, when the cut-off value $N = 1$, LibCUP gets 0.12 as success rate while the corresponding value by CrossRec is 0.21. Similarly, for other values of N, our proposed tool gains a superior performance compared to that of LibCUP, in particular when $N = 10$, CrossRec achieves a success rate which is nearly twice what LibCUP achieves, i.e., 0.42 compared to 0.22.

Similarly to D2, D3 also features library versions, however LibCUP cannot provide version-specific recommendations. For each library in the dataset, we integrated its version and fed as input for CrossRec. For this experiment, we selected only projects with a considerably high number of third-party libraries, i.e., $L = \{7, 8, 9, 10\}$ and the final results are shown in Table 7.

Table 7 shows that CrossRec also recommends specific versions of libraries. Among the configurations, the maximum success rate is 0.494, and it is reached when $L = 10$ and $N = 10$. However, in comparison to the outcome by D2, CrossRec achieves a considerably lower success rate. Moreover, there is a marginal difference in performance between different values of L. Especially, the system's performance suffers a setback when $L = 9$ in comparison to $L = 8$. By carefully investigating the dataset, we found out that the projects of D2 are very diverse, and they contain a large number of libraries. Moreover, the similarity among the projects is considerably low. This means that the dataset exhibits a high level of heterogeneity and therefore, given a project, generally we cannot find a good set of similar projects. Under the circumstances, it is more difficult to recommend highly relevant libraries, resulting in a low success rate.

[9] The success rate values for LibCUP were extracted from the paper (Saied et al., 2018).

Table 7
Success rate of CrossRec on D3 when library version is incorporated.

	# of libraries	Cut-off value (N) 1	3	5	7	10
L	7	0.215	0.313	0.361	0.392	0.423
	8	0.219	0.315	0.362	0.392	0.423
	9	0.214	0.311	0.355	0.383	0.415
	10	0.255	0.368	0.422	0.456	0.494

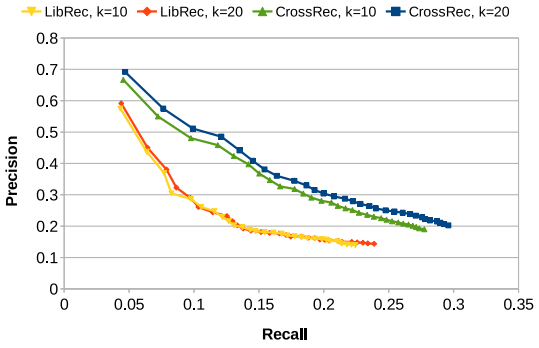


Fig. 10. Precision and Recall.

On the D3 dataset, CrossRec clearly outperforms LibCUP with respect to various configurations. Furthermore, CrossRec can recommend also libraries with a specific version, which LibCUP cannot.

RQ$_2$: *How well can LibRec and CrossRec recommend third-party libraries with respect to accuracy, sales diversity, and novelty?*

Accuracy To represent accuracy, we vary N from 1 to 30 to get *precision@N* and *recall@N*. The rationale behind the selection of 30 as the maximum cut-off value is that LibRec normally produces a short list of recommendations, ranging from 32 to 50 items. From the accuracy scores computed using Eq. (5) and Eq. (6), the Precision-Recall curves (PRCs) for all 10 rounds of validation and different values of k were sketched. However, we noticed that the figures representing the folds share a very similar pattern. Thus, for the sake of clarity, we only show in Fig. 10 results of the most representative fold. Since a PRC close to the upper right corner represents a better accuracy Di Noia et al. (2012), we see that by LibRec, changing the number of neighbor k almost makes no difference in its accuracy. Meanwhile for CrossRec we can notice that an increase of k brings a slightly better accuracy for some testing folds, however the gain is negligible. For all pieces of testing data, CrossRec always produces a superior accuracy compared to that of LibRec.

Sales diversity The catalog coverage scores for LibRec and CrossRec are depicted in Table 8. The maximum coverage values are 4.594 and 5.897 for LibRec and CrossRec, respectively. According to Eq. (7), a higher score means a better coverage. In this sense, the recommendations generated by CrossRec cover a wider spectrum of libraries than those by LibRec for both configurations, i.e., $k = 10$ and $k = 20$, using different cut-off values N. Table 9 shows the *entropy* for LibRec and CrossRec. Eq. (8) suggests that

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

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

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

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

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

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

Table 11
Wilcoxon rank sum test adjusted p-values for N={3,5,10,15}, k=10.

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

a low entropy value represents a better distribution of items, therefore the recommendations by CrossRec have a much better distribution than those obtained by LibRec. For example, for the case $N = 25$ and $k = 20$, CrossRec has an entropy of 0.635, which is much better than 2.751, the corresponding value by LibRec.

Novelty The *EPC@N* scores for LibRec and CrossRec are shown in Table 10. With LibRec, changing k from 10 to 20 decreases *novelty* for all cut-off values. For example, the *novelty* with $N = 25$ and $k = 10$ is 0.349, however once k is changed to 20, it drops to 0.261. For CrossRec, changing the number of neighbours k from 10 to 20 does not bring a rise in *novelty*. As shown in Table 10, CrossRec always obtains scores that are higher than those of LibRec. For example, when $N = 5$ and $k = 20$, CrossRec achieves a value of 0.292 for *novelty*, whereas LibRec gets 0.114. This, together with the example in Section 5.1, confirms that CrossRec recommends libraries that are closer to the long tail than LibRec can do.

Also in this case (see columns 2–6 of Tables 11 and 12), our analyses are supported by statistical procedures. Differences are always statistically significant. The effect size is negligible/small for *accuracy* (precision and recall), whereas it is large for all other indicators *sales diversity* and *novelty*.

Table 12
Cliff's d results for N={3,5,10,15}, k=10. Labels in parenthesis indicate the magnitude (n:negligible, s:small, l:large).

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

The results in Fig. 10 and Tables 8–10 demonstrate that CrossRec significantly outperforms LibRec concerning *accuracy, sales diversity*, and *novelty*, with a small/negligible effect size for *accuracy* and large elsewhere.

RQ$_3$: *What are the reasons for the performance difference between LibRec and CrossRec?*

We attempt to ascertain why CrossRec outperforms LibRec. This task might necessitate further investigations, both qualitative and quantitative research. However, by carefully studying the internal design of LibRec, we found out that the improvement attributes to the following facts. In the first place, CrossRec employs a completely different approach to represent projects and libraries: it encodes the relationships among them into a graph. Second, to compute the similarity between two projects, CrossRec assigns a weight to every library node using tf-idf (see Eq. (1)). In this way, the level of importance of a node is disproportional to its popularity. This is similar to the context of document matching where popular terms are given a low weight (Hliaoutakis et al., 2006). For instance, in Fig. 3, lib_1 is a popular node since it is referred by 4 projects and this makes it have a low weight. As a result, CrossRec is able to better capture the similarity between two projects compared to LibRec, which equally treats all libraries. Third, LibRec employs a very simple collaborative-filtering technique, though it also considers a set of k-nearest neighbor similar projects for finding libraries, it neglects their similarity level by considering all projects in the same way. Also, the technique assigns more weight to popular libraries without considering the degree of similarity between projects, from where the libraries come. This explains why LibRec recommends very popular items as shown in Section 5.1. In contrast, CrossRec improves by assigning a larger weight to libraries that come from highly similar projects (see Eq. (3)). In other words, given a project, CrossRec is able to "mimic" the behavior of highly similar projects, it attempts to suggest a comparable set of libraries. Lastly, LibRec exploits association rule mining which indeed mines items that co-exist. This is why the coverage of the recommended items is low compared to what achieved by CrossRec.

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

RQ$_4$: *Does an increase in the amount of input data help improve CrossRec's overall performance?*

Finally, we studied whether an increase in the input data contributes to an improvement in the performance of CrossRec. As it was already mentioned in Section 4.3, r is the ratio of libraries used as query over the total number of libraries that a project

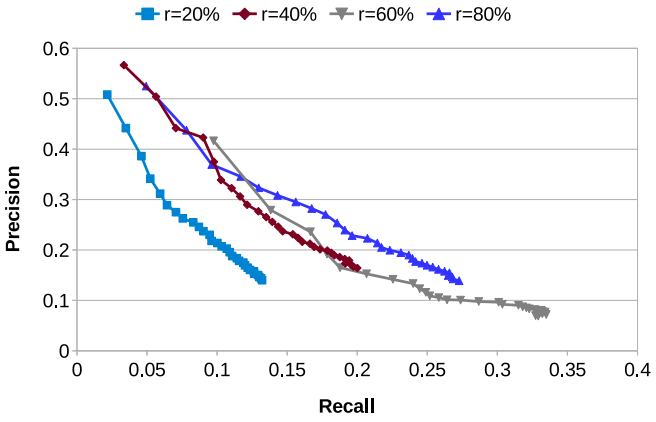


Fig. 11. Precision and recall for different amounts of testing and ground-truth data.

contains. For this evaluation, rather than using $r = 50\%$, we varied it along the following values: 20%, 40%, 60%, and 80%, and measured the achieved level of performance. This simulates different levels of a project's maturity: at the beginning when the developer has just included a small number of libraries, or later on when more libraries have been accumulatively populated alongside the project's lifecycle. This research question aims at studying if CrossRec can assist the developer in choosing the right libraries at different stages of the development process.

Fig. 11 reports the precision and recall curves (PRCs) obtained by the configurations. As a better accuracy is represented by a PRC close to the upper right corner (Di Noia et al., 2012; Davis and Goadrich, 2006), we see that generally there is an improvement in the recommendation performance when r increases. In particular, there is a sharp growth in precision and recall when r is changed from 20% to 40%. However, the gain tends to stagnate when r becomes larger. For instance, the change in performance between $r = 40\%$ and $r = 60\%$ is small and can be considered as negligible. Similarly, when r goes up to 80% from 60%, there is also a marginal change in accuracy. This corresponds to a saturation: at a certain threshold of r, e.g., $r = 50\%$ or $r = 60\%$, a little or almost no performance gain can be obtained. In essence, that means CrossRec can achieve a decent level of accuracy once the developer has included around a half of the total number of libraries.

> For CrossRec, an increase in the amount of input data considerably improves the overall performance when the ratio of query to testing data r is small. However, such the gain is disproportionate to r.

5.3. Threats to validity

We identify the threats that may adversely affect the validity of the experiments as well as the countermeasures taken to mitigate them.

Threats to internal validity are related to any factors internal to our study that can influence our results. The performances achieved by CrossRec can depend on the values of N and k. We showed in the paper results for $k = 10, 20,$ and for $N = 5, 10, 15, 20, 25$ (see more in Section 5.4). Results for other values of N and k are consistent with what we already found.

The comparison with LibFinder and LibCUP has been done in an indirect manner through their corresponding datasets, and this might possibly pose a threat to internal validity. We attempted to mitigate such a threat by exploiting the same experimental settings as well as performing various trials on the same datasets. All the additional experiments yielded comparable outcomes to those presented in the paper.

Threats to external validity concern the generalizability of our findings. In the data collection phase, we tried to cover a wide range of possibilities by mitigating also the fact that many repositories in GitHub are of low quality, which is especially true when they do not have many stars. The set of 1,200 GitHub projects was randomly created by obtaining the following distribution of stars: 14 projects have 0 stars, 135 projects have [1–4] stars, 66 projects have [5–9] stars, 512 projects have [10–99] stars, 300 projects have [100–499] stars, 78 projects have [500–999] stars, and 95 projects have more than 1,000 stars. Moreover, the number of libraries that a project in the considered dataset includes varies considerably from 10 to more than 500.

Threats to construct validity are whether the setup and measurement in the study reflect real-world situations. The threats have been mitigated by applying ten-fold cross validation, attempting to simulate a real scenario of recommending third-party libraries. In the experiments, the dataset is split into two independent parts, namely a *training set* and a *testing set*. In practice, the items in the training data correspond to the OSS projects collected a priori. They are available at developers' disposal, ready to be exploited for any mining purpose. Whereas, an item in the testing data corresponds to the project being developed. In this sense, our evaluation attempts to mimic a real deployment: the recommender systems should produce recommendations for a project based on the data available from a set of existing projects.

5.4. Discussions

CrossRec is capable of recommending a library (either just the library, or also a specific version of that library), depending on the availability of the training data. That is, the CrossRec approach transcends the limitation of the baselines considered in this paper, i.e., LibRec, LibFinder and LibCUP. Such approaches cannot recommend a specific version of a library. Moreover, CrossRec obtains a better performance when the input data is more dense, i.e., more libraries are available for training.

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

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

In contrast to LibRec, CrossRec is able to maintain a trade-off between *accuracy* and *sales diversity*, it gains better precisions and

recalls for all testing folds. Furthermore, CrossRec also achieves an adequate catalog coverage and novelty by recommending more unpopular libraries to projects. Although in this paper we performed an evaluation only on Java projects, it is also possible to apply CrossRec to search for third-party libraries in other languages, e.g., C++, Python, etc., as long as the relationship between projects and libraries can be represented following the model proposed in Sections 3.1 and 3.2.

6. Conclusions and future work

Third-party libraries contain tailored and well-defined functionalities which in turn offer a useful resource to software projects being developed. Making use of such libraries allows developers to leverage an existing infrastructure, without reinventing the wheel. In this way, recommending third-party libraries to developers help them save time as well as increase productivity. We implemented CrossRec, a novel approach to library recommendation that relies on a collaborative-filtering recommender system. The approach has been evaluated by considering different quality metrics and three independent datasets. The evaluation demonstrated that our approach outperforms three well-established baselines in terms of various quality metrics. In the first place, CrossRec recommends a library with a specific version, which cannot be obtained by all the baselines. Furthermore, the system achieves a better performance in comparison to LibRec and LibCUP, two well-known systems for library recommendation with regards to various quality indicators. CrossRec is also more efficient as it produces recommendations in a rather short time. To the best of our knowledge, our work is the first one that employs graphs to represent the relationships among software projects so as to effectively compute similarity and eventually to recommend libraries. Among other characteristics, we found out that the novelty of the outcomes is important in the context of library recommendation, as very unpopular items, i.e., those belong to the long tail, are also useful.

The deployment of various quality metrics to study the systems' performance has shown to be meaningful. Though these metrics have been widely used to evaluate recommender systems (Robillard et al., 2010; Karypis, 2001; Saracevic, 1995), to the best of our knowledge, the current work is the first one that exploits them to examine the performance of a system for recommending third-party libraries. Apart from *success rate*, we also incorporated other metrics to investigate if the approaches obtain a good performance, i.e., *accuracy, sales diversity*, and *novelty*. Each of these metrics reflects a different view on the recommendations, which helps thoroughly study the final outcomes. For future work, among others we will investigate in detail the importance of Novelty and Sales Diversity in the recommendation outcomes.

The CrossRec tool has been successfully integrated into Eclipse, in order to provide developers with an IDE prompting instant library recommendations. Also, we are working to equip CrossRec with the ability to recommend different artifacts, such as API function calls and code snippets, as well as to extract background data from various sources, such as Eclipse. A different improvement direction, as discussed in Section 3.4, would be to account for developers' feedback, or capture the extent to which a suggestion has been actually integrated into the project, to refine further recommendations. Finally, we also plan to apply the proposed approach to other ecosystems based on different languages such as C++ and C#.

Acknowledgments

The research described in this paper has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant 732223. We thank the anonymous reviewers for their valuable comments and suggestions that help us improve the manuscript.

References

Acharya, M., Xie, T., Pei, J., Xu, J., 2007. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ACM, New York, pp. 25–34. doi:10.1145/1287624.1287630.

Aggarwal, C., 2016. Neighborhood-Based Collaborative Filtering. Springer International Publishing, Cham, pp. 29–70. DOI: 10.1007/978-3-319-29659-3_2

Anderson, C., 2006. The Long Tail: Why the Future of Business is Selling Less of More. Hyperion.

Behnamghader, P., Alfayez, R., Srisopha, K., Boehm, B., 2017. Towards better understanding of software quality evolution through commit-impact analysis. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 251–262. doi:10.1109/QRS.2017.36.

Borges, H., Hora, A.C., Valente, M.T., 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2–7, 2016, pp. 334–344. doi:10.1109/ICSME.2016.31.

Briguez, C.E., Budán, M.C.D., Deagustini, C.A.D., Maguitman, A.G., Capobianco, M., Simari, G.R., 2014. Argument-based mixed recommenders and their application to movie suggestion. Expert Syst. Appl. 41 (14), 6467–6482. doi:10.1016/j.eswa.2014.03.046. http://www.sciencedirect.com/science/article/pii/S0957417414001845

Bitbucket. https://www.bitbucket.com. last accessed 16.06.2019.

Cacheda, F., Carneiro, V., Fernández, D., Formoso, V., 2011. Comparison of collaborative filtering algorithms: limitations of current techniques and proposals for scalable, high-performance recommender systems. ACM Trans. Web 5 (1), 2:1–2:33. doi:10.1145/1921591.1921593.

Castells, P., Vargas, S., Wang, J., 2011. Novelty and diversity metrics for recommender systems: Choice, discovery and relevance. In: Proceedings of the the 33rd European Conference on Information Retrieval (ECIR) International Workshop on Diversity in Document Retrieval (DDR 2011). Dublin, Ireland. http://www.ir.ii.uam.es/rim3/publications/ddr11.pdf

Cremonesi, P., Turrin, R., Lentini, E., Matteucci, M., 2008. An evaluation methodology for collaborative recommender systems. In: Proceedings of the International Conference on Automated Solutions for Cross Media Content and Multichannel Distribution. IEEE Computer Society, Washington, DC, USA, pp. 224–231. doi:10.1109/AXMEDIS.2008.13.

Davis, J., Goadrich, M., 2006. The Relationship Between Precision-Recall and ROC Curves. In: Proceedings of the 23rd International Conference on Machine Learning. ACM, New York, NY, USA, pp. 233–240. doi:10.1145/1143844.1143874.

Di Noia, T., Mirizzi, R., Ostuni, V.C., Romito, D., Zanker, M., 2012. Linked Open Data to Support Content-based Recommender Systems. In: Proceedings of the 8th International Conference on Semantic Systems. ACM, New York, NY, USA, pp. 1–8. doi:10.1145/2362499.2362501.

Eclipse. 2019https://projects.eclipse.org. last accessed 16.06.2019.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., 1996. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. AAAI Press, pp. 226–231.

Ge, M., Delgado-Battenfeld, C., Jannach, D., 2010. Beyond accuracy: Evaluating recommender systems by coverage and serendipity. In: Proceedings of the Fourth ACM Conference on Recommender Systems. ACM, New York, NY, USA, pp. 257–260. doi:10.1145/1864708.1864761.

Grissom, R.J., Kim, J.J., 2005. Effect sizes for research: A broad practical approach, 2nd edition Lawrence Earlbaum Associates.

GitHub. 2019https://www.github.com. last accessed 16.06.2019.

GitHub REST API v3.2019https://developer.github.com/v3/. last accessed 16.06.2019.

Hliaoutakis, A., Varelas, G., Voutsakis, E., Petrakis, E.G.M., Milios, E.E., 2006. Information retrieval by semantic similarity. Int. J. Semantic Web Inf. Syst. 2 (3), 55–73. doi:10.4018/jswis.2006070104.

Holm, S., 1979. A simple sequentially rejective bonferroni test procedure. Scandinav. J. Stat. 6, 65–70.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P.S., Zhang, L., 2017. Why and how developers fork what from whom in github. Empir. Softw. Eng. 22 (1), 547–578. doi:10.1007/s10664-016-9436-6.

Karypis, G., 2001. Evaluation of Item-Based Top-N Recommendation Algorithms. In: Proceedings of the ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5–10, 2001, pp. 247–254. doi:10.1145/502585.502627.

Kawaguchi, S., Garg, P.K., Matsushita, M., Inoue, K., 2006. MUDABlue: An automatic categorization system for open source repositories. J. Syst. Softw. 79 (7), 939–953. doi:10.1016/j.jss.2005.06.044. Selected papers from the 11th Asia Pacific Software Engineering Conference (APSEC2004). http://www.sciencedirect.com/science/article/pii/S0164121205001822

Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W., 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 335–346. doi:10.1109/ICSE.2017.38.

Linden, G., Smith, B., York, J., 2003. Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Comput. 7 (1), 76–80. doi:10.1109/MIC.2003.1167344.

Ma, Z., Wang, H., Guo, Y., Chen, X., 2016. LibRadar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, New York, NY, USA, pp. 653–656. doi:10.1145/2889160.2889178.

McCarey, F., Mel, O.C., Kushmerick, N., 2008. Knowledge reuse for software reuse. Web Intelli. Agent Syst. 6 (1), 59–81.

McMillan, C., Grechanik, M., Poshyvanyk, D., 2012. Detecting similar software applications. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 364–374.

Mileva, Y.M., Dallmeier, V., Burger, M., Zeller, A., 2009. Mining Trends of Library Usage. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops. ACM, New York, NY, USA, pp. 57–62. doi:10.1145/1595808.1595821.

Miranda, C., Jorge, A.M., 2008. Incremental collaborative filtering for binary ratings. In: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01. IEEE Computer Society, Washington, DC, USA, pp. 389–392. DOI: 10.1109/WIIAT.2008.263

de la Mora, F.L., Nadi, S., 2018. Which Library Should I Use?: A Metric-based Comparison of Software Libraries. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results. ACM, New York, NY, USA, pp. 37–40. doi:10.1145/3183399.3183418.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., 2015. How can i use this method? In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. IEEE Press, Piscataway, NJ, USA, pp. 880–890.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D., 2019a. 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, Copenhagen, Denmark, April 15–17, 2019, pp. 326–331.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Di Penta, M., 2019. CrossRec: Supporting Software Developers by Recommending Third-party Libraries - Online Appendix. https://github.com/crossminer/CrossRec. Last accessed 25.09.2019.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., Di Penta, M., 2019a. FOCUS: a recommender system for mining API function calls and usage patterns. In: Proceedings of the 41st International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 1050–1060. doi:10.1109/ICSE.2019.00109.

Nguyen, P.T., Di Rocco, J., Rubei, R., Di Ruscio, D., 2018. CrossSim: exploiting mutual relationships to detect similar OSS projects. In: Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 388–395. doi:10.1109/SEAA.2018.00069.

Nguyen, P.T., Rocco, J.D., Ruscio, D.D., 2019c. Building information systems using collaborative-filtering recommendation techniques. In: Proceedings of the Advanced Information Systems Engineering Workshops - CAiSE 2019 International Workshops, Rome, Italy, June 3–7, 2019, pp. 214–226. doi:10.1007/978-3-030-20948-3_19.

Nguyen, P.T., Tomeo, P., Di Noia, T., Di Sciascio, E., 2015. 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. ACM, New York, NY, USA, pp. 1477–1482. doi:10.1145/2740908.2742141.

Noia, T.D., Ostuni, V.C., 2015. Recommender systems and linked open data. In: Proceedings of the 11th International Summer School Reasoning Web. Web Logic Rules, Berlin, Germany, July 31, - August 4, 2015, Tutorial Lectures, pp. 88–113. DOI: 10.1007/978-3-319-21768-0_4

Ouni, A., Kula, R.G., Kessentini, M., Ishio, T., German, D.M., Inoue, K., 2017. Search-based software library recommendation using multi-objective optimization. Inf. Softw. Technol. 83 (C), 55–75. doi:10.1016/j.infsof.2016.11.007.

Papagelis, M., Plexousakis, D., 2005. Qualitative analysis of user-based and item-based prediction algorithms for recommendation agents. Eng. Appl. Artif. Intell. 18 (7), 781–789. doi:10.1016/j.engappai.2005.06.010.

Pazzani, M.J., Billsus, D., 2007. Content-Based Recommendation Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 325–341. Doi: 10.1007/978-3-540-72079-9_10

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M., 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 102–111. DOI: 10.1145/2597073.2597077

Ponzanelli, L., Bavota, G., Penta, M.D., Oliveto, R., Lanza, M., 2016. Prompter - Turning the IDE into a self-confident programming assistant. Empir. Softw. Eng. 21 (5), 2190–2231. doi:10.1007/s10664-015-9397-1.

Ragone, A., Tomeo, P., Magarelli, C., Di Noia, T., Palmonari, M., Maurino, A., Di Sciascio, E., 2017. Schema-summarization in linked-data-based feature selection for recommender systems. In: Proceedings of the Symposium on Applied Computing. ACM, New York, NY, USA, pp. 330–335. doi:10.1145/3019612.3019837.

Robillard, M., Walker, R., Zimmermann, T., 2010. Recommendation systems for software engineering. IEEE Softw. 27 (4), 80–86. doi:10.1109/MS.2009.161.

Saied, M.A., Ouni, A., Sahraoui, H., Kula, R.G., Inoue, K., Lo, D., 2018. Improving reusability of software libraries through usage pattern mining. J. Syst. Softw. 145, 164–179. doi:10.1016/j.jss.2018.08.032. http://www.sciencedirect.com/science/article/pii/S0164121218301699

Saracevic, T., 1995. Evaluation of Evaluation in Information Retrieval. In: Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, New York, NY, USA, pp. 138–146. doi:10.1145/215206.215351.

Sarwar, B., Karypis, G., Konstan, J., Riedl, J., 2001. Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th International Conference on World Wide Web. ACM, New York, NY, USA, pp. 285–295. doi:10.1145/371920.372071.

Schafer, J.B., Frankowski, D., Herlocker, J., Shilad, S., 2007. The Adaptive Web. Springer-Verlag, Berlin, Heidelberg, pp. 291–324. chapter Collaborative Filtering Recommender Systems

Teyton, C., Falleri, J.-R., Morandat, F., Blanc, X., 2013. Find your library experts. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), pp. 202–211. doi:10.1109/WCRE.2013.6671295.

Thummalapenta, S., Xie, T., 2007. Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 204–213. doi:10.1145/1321631.1321663.

Thung, F., Lo, D., Lawall, J., 2013a. Automated library recommendation. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), pp. 182–191. doi:10.1109/WCRE.2013.6671293.

Thung, F., Wang, S., Lo, D., Lawall, J., 2013b. Automatic recommendation of API methods from feature requests. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 290–300. doi:10.1109/ASE.2013.6693088.

Tsunoda, M., Kakimoto, T., Ohsugi, N., Monden, A., Matsumoto, K.-i., 2005. Javawock: A java class recommender system based on collaborative filtering. In: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China, July 14–16, 2005, pp. 491–497.

Uddin, G., Khomh, F., 2017. Automatic Summarization of API Reviews. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 159–170.

Vargas, S., Castells, P., 2014. Improving sales diversity by recommending users to items. In: Proceedings of the Eighth ACM Conference on Recommender Systems, RecSys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014, pp. 145–152. doi:10.1145/2645710.2645744.

Wang, H., Guo, Y., Ma, Z., Chen, X., 2015. Wukong: A scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, pp. 71–82. doi:10.1145/2771783.2771795.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D., 2013. Mining Succinct and High-coverage API Usage Patterns from Source Code. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE, Piscataway, pp. 319–328. doi:10.1109/MSR.2013.6624045.

Wang, J., Han, J., 2004. BIDE: Efficient Mining of Frequent Closed Sequences. In: Proceedings of the 20th International Conference on Data Engineering. IEEE, Washington, pp. 79–90. doi:10.1109/ICDE.2004.1319986.

Wu, L., Shah, S., Choi, S., Tiwari, M., Posse, C., 2014. The browsemaps: Collaborative filtering at linkedin. RSWeb@RecSys. CEUR-WS.org.

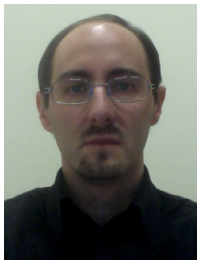
Zhao, Z.-D., Shang, M.-s., 2010. User-based collaborative-filtering recommendation algorithms on hadoop. In: Proceedings of the Third International Conference on Knowledge Discovery and Data Mining. IEEE Computer Society, Washington, DC, USA, pp. 478–481. doi:10.1109/WKDD.2010.54.

Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H., 2009. MAPO: mining and recommending API usage patterns. In: Proceedings of the 23rd European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, pp. 318–343. doi:10.1007/978-3-642-03013-0_15.

Phuong T. Nguyen is a postdoctoral researcher at the University of L'Aquila, Italy. He obtained a Ph.D in Computer Science from the University of Jena, Germany. Since the graduation, he has worked as a university teaching and research assistant in Vietnam and Italy. His research interests include Computer Networks, Semantic Web, Recommender Systems, and Machine Learning. Recently, he has been working to develop recommender systems in Software Engineering for mining open source code repositories.

Juri Di Rocco is a postdoctoral researcher at the Department of Information Engineering Computer Science and Mathematics of the University of L'Aquila. Previously, He obtained a PhD degree from the University of L'Aquila in the MDEGroup research team with Alfonso Pierantonio and Davide Di Ruscio. He is interested in all aspects of software language engineering. Main research interests are related to several aspects of Model Driven Engineering (MDE) including domain specific modelling languages, model transformation, model differencing, modelling repositories and mining techniques.

Davide Di Ruscio is Associate Professor at the DISIM - University of L'Aquila. His main research interests are related to several aspects of Software Engineering, Open Source Software, and Model Driven Engineering (MDE) including domain specific modelling languages, model transformation, model differencing, and model evolution. He has published more than 130 papers in various journals, conferences and workshops on such topics. He is a member of the steering committee of the International Conference on Model Transformation (ICMT), of the Software Language Engineering (SLE) conference, of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE), of the Workshop on Modelling in Software Engineering at ICSE (MiSE) and of the International Workshop on Robotics Software Engineering (RoSE). He is in the editorial board of the International Journal on Software and Systems Modeling (SoSyM), of the Journal of Object Technology, and of the IET Software journal. More information is available at http://people.disim.univaq.it/diruscio.



Massimiliano Di Penta is an associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is an author of over 270 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, ICSME. He is in the editorial board of the Empirical Software Engineering Journal edited by Springer, ACM Transactions on Software Engineering and Methodology, and of the Journal of Software: Evolution and Processes edited by Wiley, and has served the editorial board of the IEEE Transactions on Software Engineering. Web: http://www.ing.unisannio.it/mdipenta/.