Development of recommendation systems for software engineering: the CROSSMINER experience

Juri Di Rocco[1] · Davide Di Ruscio[1] (ID) **· Claudio Di Sipio[1] · Phuong T. Nguyen[1] · Riccardo Rubei[1]**

Abstract

To perform their daily tasks, developers intensively make use of existing resources by consulting open source software (OSS) repositories. Such platforms contain rich data sources, e.g., code snippets, documentations, and user discussions, that can be useful for supporting development activities. Over the last decades, several techniques and tools have been promoted to provide developers with innovative features, aiming to bring in improvements in terms of development effort, cost savings, and productivity. In the context of the EU H2020 CROSSMINER project, a set of recommendation systems has been conceived to assist software programmers in different phases of the development process. The systems provide developers with various artifacts, such as third-party libraries, documentation about how to use the APIs being adopted, or relevant API function calls. To develop such recommendations, various technical choices have been made to overcome issues related to several aspects including the lack of baselines, limited data availability, decisions about the performance measures, and evaluation approaches. This paper is an experience report to present the knowledge pertinent to the set of recommendation systems developed through the CROSS-MINER project. We explain in detail the challenges we had to deal with, together with the related lessons learned when developing and evaluating these systems. Our aim is to provide the research community with concrete takeaway messages that are expected to be useful for those who want to develop or customize their own recommendation systems. The reported experiences can facilitate interesting discussions and research work, which in the end contribute to the advancement of recommendation systems applied to solve different issues in Software Engineering.

Keywords Recommendation systems · Empirical evaluation · Experience report

1 Introduction

Open-source software (OSS) forges, such as GitHub or Maven, offer many software projects that deliver stable and well-documented products. Most OSS forges typically sustain vibrant user and expert communities which in turn provide decent support, both for answering user questions and repairing reported software bugs. Moreover, OSS platforms are also an essential source of consultation for developers in their daily development tasks (Cosentino et al. 2017). Code reusing is an intrinsic feature of OSS, and developing new software by leveraging existing open source components allows one to considerably reduce their development effort. The benefits resulting from the reuse of properly selected open-source projects are manifold including the fact that the system being implemented relies on open source code, *"which is of higher quality than the custom-developed code's first incarnation"* (Spinellis and Szyperski 2004). In addition to source code, also metadata available from different related sources, e.g., communication channels and bug tracking systems, can be beneficial to the development life cycle if being properly mined (Ponzanelli et al. 2016). Nevertheless, given a plethora of data sources, developers would struggle to look for and approach the sources that meet their need without being equipped with suitable machinery. Such a process is time-consuming since the problem is not a lack, but in contrast, an overload of information coming from heterogeneous and rapidly evolving sources. In particular, when developers join a new project, they have to typically master a considerable number of information sources (often at a short time) (Dagenais et al. 2010). In this respect, the deployment of systems that use existing data to improve developers' experience is of paramount importance.

Recommendation systems are a crucial component of several online shopping systems, allowing business owners to offer personalized products to customers (Linden et al. 2003). The development of such systems has culminated in well-defined recommendation algorithms, which in turn prove their usefulness in other fields, such as entertainment industry (Gomez-Uribe and Hunt 2015), or employment-oriented service (Wu et al. 2014). Recommendation systems in software engineering (Robillard et al. 2014) (RSSE hereafter) have been conceptualized on a comparable basis, i.e., they assist developers in navigating large information spaces and getting instant recommendations that are helpful to solve a particular development task (Nguyen et al. 2019; Ponzanelli et al. 2016). In this sense, RSSE provide developers with useful recommendations, which may consist of different items, such as code examples (Fowkes and Sutton 2016; Moreno et al. 2015; Nguyen et al. 2019), topics (Di Rocco et al. 2020; Di Sipio et al. 2020), third-party components (Nguyen et al. 2019; Thung et al. 2013), documentation (Ponzanelli et al. 2016; Rubei et al. 2020), to name a few.

While the issue of designing and implementing generic recommendation systems has been carefully addressed by state-of-the-art studies (Proksch et al. 2015; Robillard et al. 2014), there is a lack of proper references for the design of a recommendation system in a concrete context, i.e., satisfying requirements by various industrial partners. By means of a thorough investigation of the related work, we realized that existing studies tackled the issue of designing and implementing a recommendation system for software engineering in general. However, to the best of our knowledge, an experience report extracted from real development projects is still missing. The report presented in this paper would come in handy for those who want to conceive or customize their recommendation systems in a specific context. For example, a developer may be interested in understanding which techniques are suitable for producing recommendations; or how to capture the developer's context; or which is the most feasible way to present recommendation outcomes, to name a few.

In the context of the EU CROSSMINER project[1] we exploited cutting-edge information retrieval techniques to build recommendation systems, providing software developers with practical advice on various tasks through an Eclipse-based IDE and dedicated analytical Web-based dashboards. Based on the project's mining tools, developers can select open-source software and get real-time recommendations while working on their development tasks. This paper presents an experience report pertaining to the implementation of the CROSSMINER recommendation systems, with focus on three main phases, i.e., *Requirements elicitation*, *Development*, and *Evaluation*. We enumerate the challenges that we faced when working with these phases and present the lessons gained by overcoming the challenges. With this work, we aim at providing the research community at large with practical takeaway messages that one can consult when building their recommendation systems.

Outline of the paper　The paper is structured as follows: Section 2 gives an overview of the CROSSMINER project and the underpinning motivations. Sections 3–5 discuss the challenges we had to address while conceiving the CROSSMINER recommendation systems and the corresponding lessons learned that we would like to share with the community as well as with potential developers of new recommendation systems. Section 6 reviews the related work and finally, Section 7 sketches perspective work and concludes the paper.

2　The CROSSMINER project

In recent years, software development activity has reached a high degree of complexity, led by the heterogeneity of the components, data sources and tasks. The adoption of recommendation systems in software engineering (RSSE) aims at supporting developers in navigating large information spaces and getting instant suggestions that might be helpful to solve a particular development task (Robillard et al. 2014).

In the context of open-source software, developing new software systems by reusing existing open-source components raises relevant challenges related to at least the following activities (Karlsson 1995): *(i)* searching for candidate components, *(ii)* evaluating a set of retrieved candidate components to find the most suitable ones, and *(iii)* adapting the selected components to fit some specific requirements. The CROSSMINER project conceived techniques and tools for extracting knowledge from existing open source components and use it to provide developers with real-time recommendations that are relevant to the current development task.

As shown in Fig. 1, the CROSSMINER components are conceptually in between the developer and all the different and heterogeneous data sources (including source code, bug tracking systems, and communication channels) that one needs to interact with when understanding and using existing open-source components. In particular, an Eclipse-based IDE and Web-based dashboards make use of data produced by the mining tools working on the the back-end of the CROSSMINER infrastructure to help developers perform the current development tasks. CROSSMINER is under the umbrella of Eclipse Research Labs with the name Eclipse SCAVA.[2]

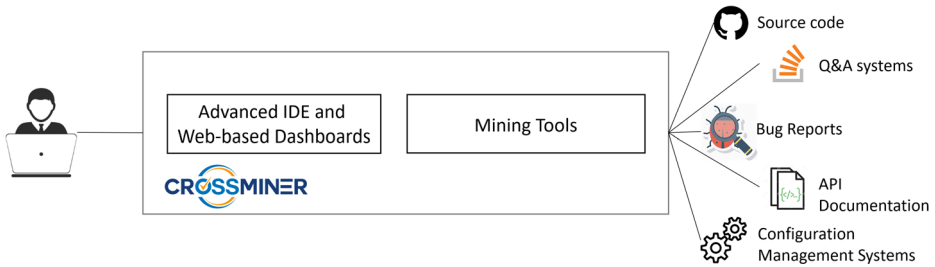
[1]https://www.crossminer.org

[2]https://www.eclipse.org/scava/

Fig. 1 CROSSMINER overview

2.1 CROSSMINER as a set of recommendation systems

Figure 2 shows CROSSMINER from a different perspective. In particular, CROSSMINER can be seen as a set of recommendation systems, each designed to implement the four main activities, which are typically defined for any recommendation systems, i.e., data pre-processing, capturing context, producing recommendations, and presenting recommendations (Robillard et al. 2014) as shown in the upper side of Fig. 2. Accordingly, the CROSSMINER solution is made up of four main modules: the `Data Preprocessing` module contains tools that extract metadata from OSS repositories (see the middle part of Fig. 2). Data can be of different types, such as source code, configuration, or cross-project relationships. Natural language processing (NLP) tools are also deployed to analyze developer forums and discussions. The collected data is used to populate a knowledge base which serves as the core for the mining functionalities. By capturing developers' activities (`Capturing Context`), an IDE is able to generate and display recommendations (`Producing Recommendations` and `Presenting Recommendations`). In particular, the developer context is used as a query sent to the knowledge base that answers with recommendations that are relevant to the developer contexts (see the lower side of Fig. 2). Machine learning techniques are used to infer knowledge underpinning the creation of relevant real-time recommendations.
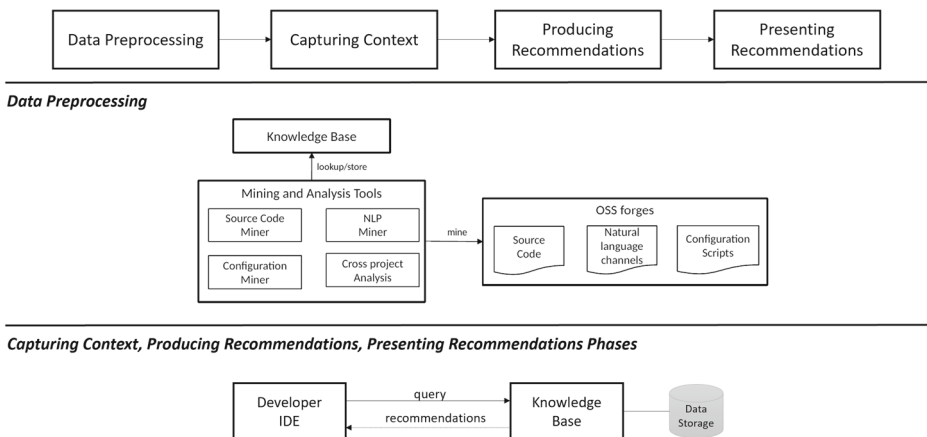

Fig. 2 High-level view of the CROSSMINER project

Table 1 The CROSSMINER use cases

No.	Artifact	Description	Developed tool
①	*Similar OSS projects*	We crawl data from OSS platforms to find similar projects to the system being developed, with respect to different criteria, e.g., external dependencies, or API usage (Nguyen et al. 2020). This type of recommendation is beneficial to the development since it helps developer learn how similar projects are implemented.	CrossSim
②	*Additional components*	During the development phase, programmers search for components that projects similar to the one under developed have also included, for instance, a list of external libraries (McMillan et al. 2010), or API function calls (Moreno et al. 2015).	CrossRec, FOCUS
③	*Code snippets*	Well-defined snippets showing how an API is used in practice are extremely useful. These snippets provide developers with a deeper insight into the usage of the APIs being included (Nguyen et al. 2019).	FOCUS
④	*Relevant topics*	GitHub uses tags as a means to narrow down the search scope. The goal is to help developers approach repositories, and thus increasing the possibility of contributing to their development and widespread their usage.	MNBN

The CROSSMINER `knowledge base` allows developers to gain insights into raw data produced by different mining tools, which are the following ones:

– *Source code miners* to extract and store actionable knowledge from the source code of a collection of open-source projects;
– *NLP miners* to extract quality metrics related to the communication channels, and bug tracking systems of OSS projects by using Natural Language Processing and text mining techniques;
– *Configuration miners* to gather and analyze system configuration artefacts and data to provide an integrated DevOps-level view of a considered open source project;
– *Cross-project miners* to infer cross-project relationships and additional knowledge underpinning the provision of real-time recommendations;

The CROSSMINER recommendation systems have been developed to satisfy the requirements by six industrial use-case partners of the project working on different domains including IoT, multi-sector IT services, API co-evolution, software analytics, software quality assurance, and OSS forges.[3] In particular, Table 1 specifies the main use cases solicited by our industrial partners. To satisfy the given requirements the following recommendation systems have been developed:

– *CrossSim* (Nguyen et al. 2018, 2020) – It is an approach for recommending similar projects with respect to the third-party library usage, stargazers and commiters, given a specific project;
– *CrossRec* (Nguyen et al. 2019) – It is a framework that makes use of **C**ross Projects **R**elationships among **O**pen **S**ource **S**oftware Repositories to build a library **Rec**ommendation System on top of CrossSim;

---

[3]https://www.crossminer.org/consortium

Fig. 3 Main activities underpinning the development of the CROSSMINER recommendation systems
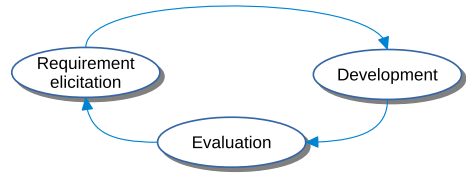
– *FOCUS* (Nguyen et al. 2019, 2021) – The system assists developers by providing them with API function calls and source code snippets that are relevant for the current development context;
– *MNBN* (Di Sipio et al. 2020) – It is an approach based on a Multinomial Naive Bayesian network technique to automatically recommend topics given the README file(s) of an input repository.

By referring to Fig. 2, the developed recommendation systems are implemented in the `Knowledge Base` component. Moreover, it is important to remark that even though such tools can be used in an integrated manner directly from the `Developer IDE`, their combined usage is not mandatory. They are different services that developers can even use separately according to their needs.

For more details about the recommendation systems developed in the context of the CROSSMINER projects, readers can refer to the related papers presenting them. Without giving details on each tool's inner technicalities, in the following sections, we focus on the challenges we faced while conceiving the CROSSMINER recommendation systems, and on the lessons that we learned on the way. We believe that sharing them with the community is desirable because of two main motivations:

– The recommendation systems have been developed in a real context to cope with industrial needs of different use-case partners;
– According to the evaluation procedure performed towards the end of the CROSS-MINER project, the industrial partners have been particularly satisfied by the developed recommendation systems, which have been mainly graded as *excellent* by most of the partners that were asked to express their judgement in the range *insufficient*, *sufficient*, *good*, *excellent* (see the public deliverable D8.16[4] for more details).

2.2 The CROSSMINER development process

The development of the CROSSMINER recommendation systems has been done by following an iterative process, as shown in Fig. 3. In particular, to produce the recommendation systems that are now part of the CROSSMINER platform, the following steps have been undertaken:

– *Requirement elicitation*: identification of the expected features provided by the CROSS-MINER platform in terms of recommendations and development support;
– *Development*: implementation of the needed recommendation systems to accommodate the requirements defined in the previous step;
– *Evaluation:* assess the performance of the produced recommendations by using properly defined evaluation procedures and selected metrics.

---

[4]CROSSMINER D8.16 Case Study Evaluation - https://cordis.europa.eu/project/id/732223/results
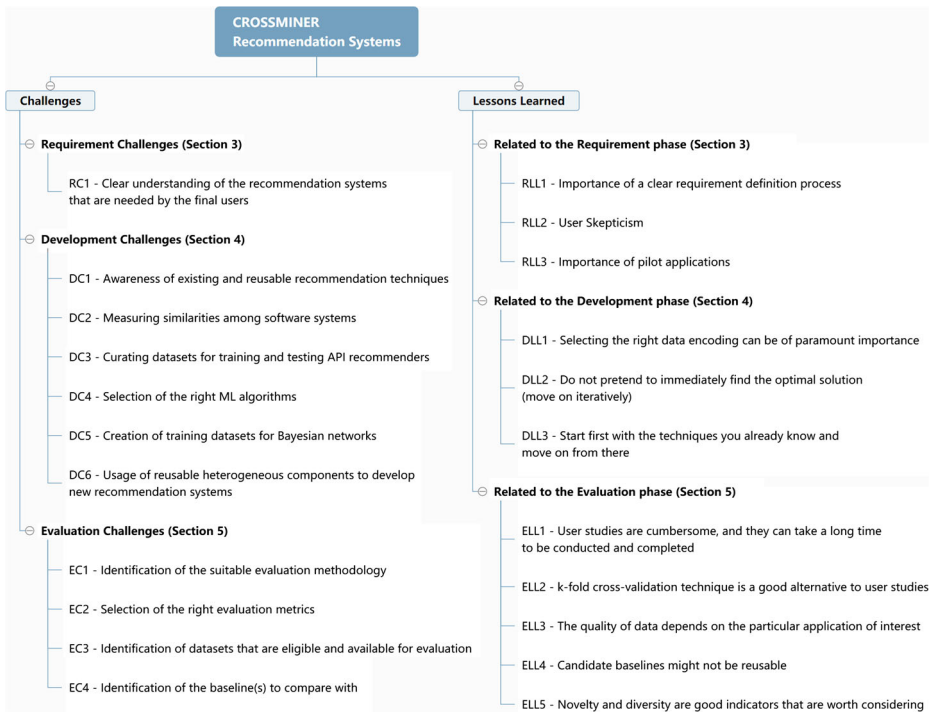
Fig. 4 Map of challenges and lessons learned

In the following sections, such steps are described in detail. For each of them, we discuss the challenges we had to overcome and the difficulties we had while conceiving the tools as asked by the projects' use-case partners. The methods we employed to address such challenges are presented together with the corresponding lessons learned.

An overview of all the challenges and lessons learned are shown in the map depicted in Fig. 4. For the sake of readability, challenges related to the requirement, development, and evaluation phases are identified with the three strings *RC*, *DC*, and *EC*, respectively, followed by a cardinal number. Similarly, the lessons learned are organized by distinguishing them with respect to the requirement (*RLL*), development (*DLL*), and evaluation (*ELL*) phases.

3 Challenges and lessons learned from eliciting the requirements of the CROSSMINER recommendation systems

During the first six months of the project, we worked in tight collaboration with the industrial partners of the consortium to understand what they were expecting from the project technologies in terms of development support. For instance, the main business activity of one of the use-case partners consisted in the development and maintainance of a software quality assessment tool built atop of SonarQube.[5] Whenever a new version of SonarQube
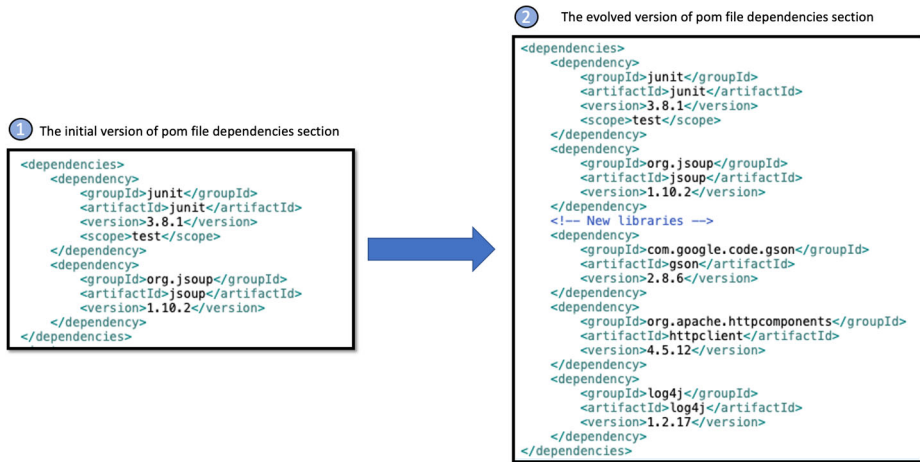
---

[5]https://www.sonarqube.org/

Fig. 5 `pom.xml` files of a project before (*left*) and after (*right*) having adopted third-party libraries recommended by CrossRec

was released, they needed to upgrade their tool to make it work with the new version of the SonarQube APIs. We, in turn, had to interact with the interested use-case partner to identify the types of recommendations that might have been useful for them to manage their API evolution problems. Other use case partners were asking for additional recommendations, which were posing requirements that might have had ripple effects on the other one's.

3.1 Challenges

> **RC1 - Clear understanding of the recommendation systems that are needed by the end users**: Getting familiar with the functionalities that are expected from the final users of the envisioned recommendation systems is a daunting task. We might risk spending time on developing systems that are able to provide recommendations, which instead might not be relevant and in line with the actual user needs.

To deal with such a challenge and thus mitigate the risks of developing systems that might not be in line with the user requirements, we developed proof-of-concept recommendation systems. In particular, we implemented demo projects that reflected real-world scenarios in terms of explanatory context inputs and corresponding recommendation items that the envisioned recommendation systems should have produced. For instance, concerning CrossRec, we experimented on the *jsoup-example*[6] explanatory Java project for scraping HTML pages. This project consists of source code and few related third-party libraries already included, i.e., *json-soup*[7] and *junit*[8] as shown in the left-hand side of Fig. 5.

---

[6]https://github.com/MDEGroup/FOCUS-user-evaluation

[7]https://jsoup.org/

[8]https://junit.org/

```
1  public static void getScoresFromLivescore()
2  throws IOException {
3    Document document =
4    Jsoup.connect("https://www.livescore.com/").get();
5    Elements scores =
6    document.getElementsByClass("sco");
7    scores = scores.parents();
8    ...
9  }
```

Listing 1 Partial implementation of the explanatory `getScoresFromLivescore()` method

By considering such project as input, CrossRec provides a list of additional libraries as a suggestion that the project under development should also include. For instance, some beneficial libraries to be recommended are as follows: *(i) gson*[9] for manipulating JSON resources; *(ii) httpclient*[10] for client-side authentication, HTTP state management, and HTTP connection management; and *(iii) log4j*[11] to enable logging at runtime.

By carefully examining the recommended libraries, we see that they have a positive impact on the project. To be concrete, the usage of the *httpcomponent* library allows the developer to access HTML resources by unloading the result state management and client-server authorization implementation on the library; meanwhile *gson* could provide a parallel way to crawl public Web data; finally introducing a logging library, i.e., *log4j*, can improve the project's maintainability.

Concerning FOCUS, the process was a bit different, i.e., use-case partners were providing us with incomplete source code implementation and their expectations regarding useful recommendations. Such artifacts were used as part of the requirements to implement the system able to resemble them. The use-case partner expects to get code snippets that include suggestions to improve the code, and predictions on next API function calls.

To agree with the use-case partners on the recommendations that were expected from FOCUS, we experimented on a partially implemented method of the *jsoup-example* project named `getScoresFromLivescore` shown in Listing 1. The method should be designed so as being able to collect the football scores listed in the livescore.com home page. To this end, a JSON document is initialized with a connection to the site URL in the first line. By using the JSOUP facilities, the list of HTML element of the class `sco` is stored in the variable `score` in the second line. Finally, the third line updates the scores with all of the parents and ancestors of the selected scores elements.

Figure 6 depicts few recommendations that our use-case partners expected when we presented the example shown in Listing 1. The blue box contains the recommendation for improving the code, i.e., the `userAgent` method is to prevent sites from blocking HTTP requests, and to predict the next *jsoup* invocation. Furthermore, some recommendations could be related to API function calls of a competitor library or extension. For this reason, the green and red boxes contain invocations of *HTMLUnit*,[12] a direct competitor of *jsoup* that includes different browser user agent implementations, and *jsoupcrawler* a custom extension of *jsoup*. FOCUS has been conceptualized to suggest to developers recommendations consisting of a list of API method calls that should be used next. Furthermore, it

---

[9]https://github.com/google/gson

[10]https://hc.apache.org/

[11]https://logging.apache.org/

[12]http://htmlunit.sourceforge.net/

```
0.71711844  org.jsoup.Connection.userAgent(java.lang.String)
0.71711844  org.jsoup.nodes.Document.children()
0.71711844  org.jsoup.nodes.Document.select(java.lang.String)
0.71711844  org.jsoup.nodes.Element.children()
0.71711844  org.jsoup.nodes.Element.select(java.lang.String)
0.71711844  org.jsoup.select.Elements.attr(java.lang.String)
0.71711844  org.jsoup.select.Elements.first()
0.71711844  org.jsoup.select.Elements.html()
0.71711844  org.jsoup.select.Elements.iterator()
0.63971543  com.gargoylesoftware.htmlunit.WebClient.WebClient(com.gargoyles
0.63971543  com.gargoylesoftware.htmlunit.WebClient.close()
0.63971543  com.gargoylesoftware.htmlunit.WebClient.getOptions()
0.63971543  com.gargoylesoftware.htmlunit.WebClient.getPage(java.lang.String)
0.63971543  com.gargoylesoftware.htmlunit.WebClientOptions.setCssEnabled(b
0.63971543  com.gargoylesoftware.htmlunit.WebClientOptions.setJavaScriptEna
0.63971543  com.gargoylesoftware.htmlunit.html.HtmlPage.asXml()
0.63971543  com.snicesoft.jsoupcrawler.config.entity.Info.isCss()
0.63971543  com.snicesoft.jsoupcrawler.config.entity.Info.isJs()
0.63971543  com.snicesoft.jsoupcrawler.core.ParseType.ordinal()
```

Fig. 6 Recommended API calls for the `getScoresFromLivescore()` method in Listing 1

also recommends real code snippets that can be used as a reference to support developers in finalizing the method definition under development. More code examples provided by FOCUS are available in an online appendix.[13]

3.2 Lessons learned

RLL1 – Importance of a clear requirement definition process As previously mentioned, we managed to address Challenge RC1 through a tight collaboration with the use case partners. In particular, we applied the requirement definition process shown in Fig. 7, which consists of the following steps and that in our opinion can be applied even in contexts that are different from the CROSSMINER one:

- *Requirement elicitation:* The final user identifies use cases that are representative and that identify the functionalities that the wanted recommendation systems should implement. By considering such use cases, a list of requirements is produced;
- *Requirement prioritization:* The list of requirements produced in the previous step can be very long, because users tend to add all the wanted and ideal functionalities even those that might be less crucial and important for them. For this reason, it can be useful to give a priority to each requirement in terms of the modalities *shall*, *should*, and *may*. *Shall* is used to denote essential requirements, which are of highest priority for validation of the wanted recommendation systems. *Should* is used to denote a requirement that would be not essential even though would make the wanted recommendation systems working better. *May* is used to denote requirements that would be interesting to satisfy and explore even though irrelevant for validating the wanted technologies;
- *Requirement analysis by R&D partners:* The prioritized list of requirements is analyzed by the *research and development* partners with the aim of identifying the major components that need to be developed. Possible technological challenges that might
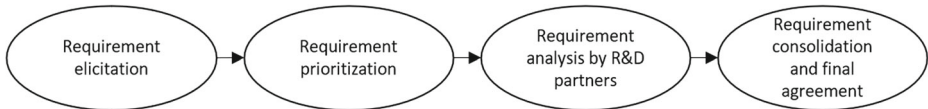
---

[13]https://github.com/crossminer/FOCUS

Fig. 7 Requirement definition process

compromise the satisfaction of some requirements are identified in this step and considered in the next step of the process;

– *Requirement consolidation and final agreement:* By considering the results of the analysis done by the R&D partners, the list of requirements is further refined and consolidated. After this step, user case partners have ensured highest priority requirements, which will be implemented by R&D partners.

We have applied such a process in different projects and we successfully applied it also for developing the recommendation systems that we identified in the context of the CROSSMINER project.

RLL2 – Users skepticism Especially at the early stages of the wanted recommendation systems development, target users might be skeptical about the relevance of the potential items that can be recommended. We believe that defining the requirements of the desired recommendation systems in tight collaboration with the final users is the right way to go. Even when the proposed approach has been evaluated employing adequate metrics, final users still might not be convinced about the retrievable recommendations' relevance. User studies can be one of the possible options to increase the final users' trust, even though a certain level of skepticism might remain when the intended final users have not been involved in the related user studies.

RLL3 – Importance of pilot applications Using a pilot application can be beneficial to support the interactions between the final users and the developers of the wanted recommendation systems. The application can allow the involved parties to tailor the desired functionalities utilizing explanatory inputs and corresponding recommendations that the envisioned system should produce.

4 Challenges and lessons learned from developing the CROSSMINER recommendation systems

Once the requirements of the expected RSs were agreed with the use-case partners, we started with the development of each identified RS.

> **DC1 – Awareness of existing and reusable recommendation techniques**: Over the last decades, several recommendation systems have been developed by both academia and industry. When realizing a new type of recommendation system, it is crucial to have a clear knowledge of the possible techniques and patterns that might be employed to develop new ones. Since the solution space is extensive, comparing and evaluating candidate approaches can be a very daunting task.

To overcome such challenge we performed a rigorous literature review by reviewing related studies emerging from premier venues in the Software Engineering domain, i.e., conferences such as ICSE, ASE, SANER, or journals such as TSE, TOSEM, JSS, to name a few. Being aware of existing systems is also important for the evaluation phase. For studying a recommendation system, besides conventional quantitative and qualitative evaluations, it is necessary to compare it with state-of-the-art approaches. Such an issue is also critical in other domains, e.g., Linked Data (Noia and Ostuni 2015), or music recommendations (Nguyen et al. 2015; Schedl et al. 2018). When we started with the design of the systems, the solution space was huge, considering the use-case partners' requirements. However, being aware of what already exists is very important to save time, resources, and avoid the reimplementation of already existing techniques and tools. To this end, by analyzing the existing literature about recommendation systems we identified and modeled their relevant variabilities and commonalities.

Section 4.1 brings in the main design features of our recommendation systems. Specific challenges that have faced to design the different tools are described in Sections 4.2–4.4. The lessons learned by designing the CROSSMINER recommendation systems are discussed in Section 4.5.

4.1 Main design features

Our results are documented using feature diagrams which are a common notation in domain analysis (Czarnecki 2002). Figure 8 shows the top-level features or recommendation systems, i.e., *Data Preprocessing*, *Capturing Context*, *Producing Recommendations*, and *Presenting Recommendations* in line with the main functionalities typically implemented by recommendation systems.

We extracted all the shown components mainly from existing studies (Bobadilla et al. 2013; LASER 2015; Robillard et al. 2014) as well as from our development experience under the needs of the CROSSMINER project. The top-level features shown in Fig. 8 are described below.

Data preprocessing In this phase techniques and tools are applied to extract valuable information from different data sources according to their nature. In particular, *structured data* adheres to several rules that organize elements in a well-defined manner. Source code and XML documents are examples of this category. Contrariwise, *unstructured data* may represent different content without defining a methodology to access the data. Documentation, blog, and plain text fall into this category. Thus, the data preprocessing component must be carefully chosen considering the features of these miscellaneous sources.

`ASTParsing` involves the analysis of structured data, typically the source code of a given software project. Several libraries and tools are available to properly perform operations on ASTs, e.g., fetching function calls, retrieving the employed variables, and analyzing the source code dependencies. Additionally, snippets of code can be analyzed using `Fingerprints`, i.e., a technique that maps every string to a unique sequence of bits. Such a strategy is useful to uniquely identify the input data and compute several operations on it, i.e., detect code plagiarism as shown in Zheng et al. (2018).

Moving to unstructured input, `Tensors` can encode mutual relationships among data, typically users' preferences. Such a feature is commonly exploited by collaborative filtering approaches as well as by heavy computation on the input data to produce recommendations. Plain text is the most spread type of unstructured data and it includes heterogeneous

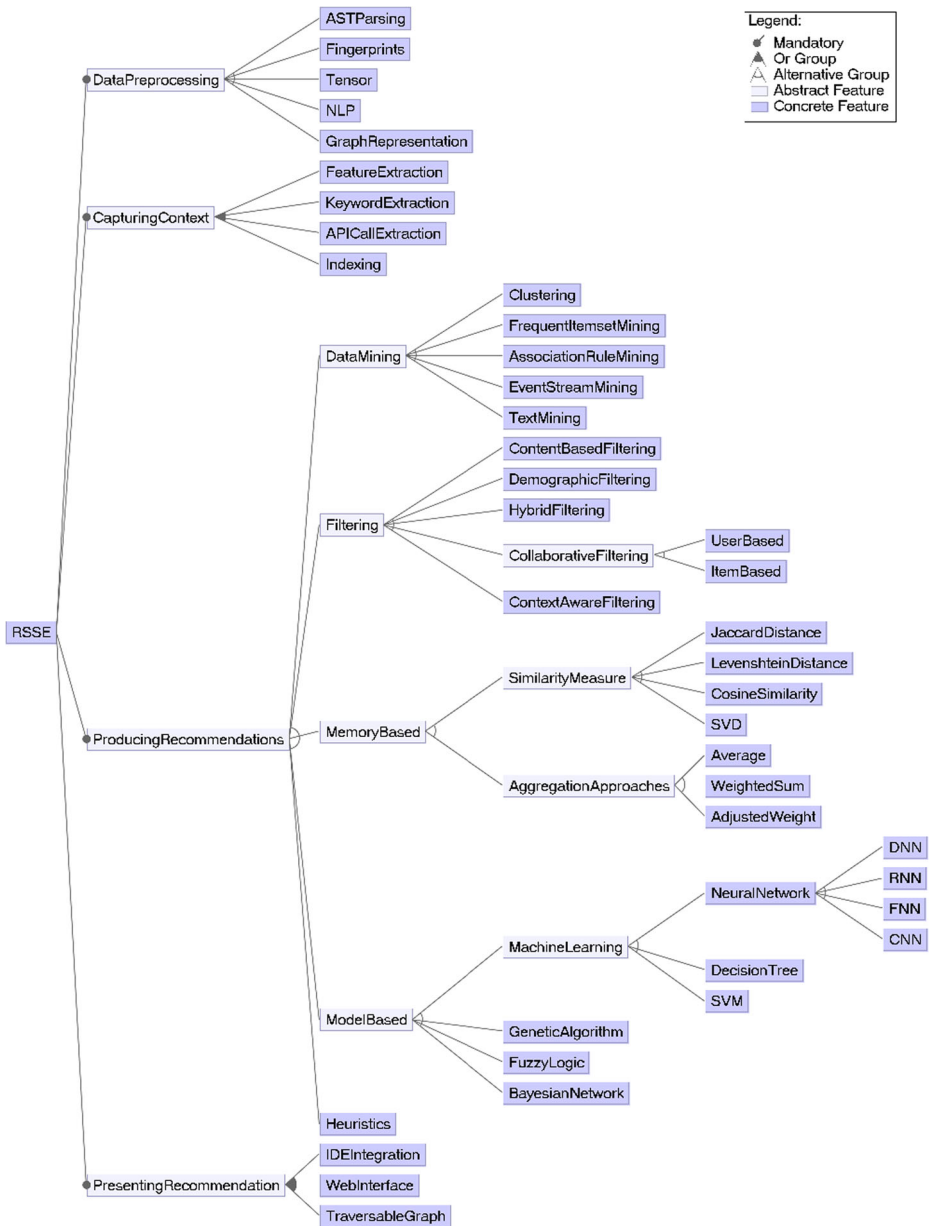**Fig. 8** Main design features of recommendation systems in software engineering

content, i.e., API documentation, repository's description, Q&A posts, to mention a few. A real challenge is to extract valuable elements without losing any relevant information. Natural processing language (NLP) techniques are employed to perform this task by means of both syntactic and semantic analysis. Stemming, lemmatization, and tokeniza-

tion are the main strategies successfully applied in existing recommendation systems. Even the *MNBN* approach previously presented employs such techniques as preparatory task before the training phase. Similarly to tensors, `GraphRepresentation` is useful to model reciprocal associations among considered elements. Furthermore, graph-based data encodings can be used to find peculiar patterns considering nodes and edges semantic.

Capturing context After the data preprocessing phase, the developer context is excerpted from the programming environment to enable the underpinning recommendation engine. A well-founded technique primarily employed in the ML domain is the `FeatureExtraction` to concisely represent the developer's context. Principal Component Analysis (PCA) and Latent Semantic analysis (LDA) are just two of such techniques employed for such a purpose. `Keyword extraction` and `APICallExtraction` are two techniques mostly used when the *Capturing Context* phase has to analyze source code. Capturing context often involves the search over big software projects. Thus, a way to store and access a large amount of data is necessary to speed up the recommendation item delivery. `Indexing` is a technique mainly used by the code search engines to retrieve relevant elements in a short time.

Producing recommendations In this phase, the actual recommendation algorithms are chosen and executed to produce suggestions that are relevant for the user context, once it is previously captured. By several variating parameters such as type of the required input and the underlying structure, we can elicit different features as represented in the diagram shown in Fig. 8. Concerning `Data Mining` techniques, some of them are based on pattern detection algorithms, i.e., `Clustering`, `FrequentItemsetMining`, and `AssociationRuleMining`. `Clustering` is usually applied to group objects according to some similarity functions. The most common algorithm is the K-means based on minimizing the distance among the items. A most representative element called centroid is calculated through a linkage function. After such a computation, this algorithm can represent a group of elements by referring to the most representative value. `FrequentItemsetMining` aims to group items with the same frequencies, whereas `AssociationRuleMining` uses a set of rules to discover possible semantic relationships among the analysed elements. Similarly, the `EventStreamMining` technique aims to find recurrent patterns in data streams. A stream is defined as a sequence of events usually represented by a Markov chain. Through this model, the algorithm can exploit the probability of each event to establish relationships and predict a specific pattern. Finally, `TextMining` techniques often involve information retrieval concepts such as entropy, latent semantic analysis (LSA), or extended boolean model. In the context of producing recommendations, such strategies can be used to find similar terms by exploiting different probabilistic models that analyze the correlation among textual documents.

The availability of users' preferences can affect the choice of recommendation algorithms. `Filtering` strategies dramatically exploit the user data, e.g., their ratings assigned to purchased products. `ContentBasedFiltering` (CBF) employs historical data referring to items with positive ratings. It is based on the assumption that items with similar features have the same score. Enabling this kind of filtering requires the extraction of the item attributes as the initial step. Then, CBF compares the set of active items, namely the context, with possible similar items using a similarity function to detect the closer ones to the user's needs. `DemographicFiltering` compares attributes coming from the

users themselves instead of the purchased items. These two techniques can be combined in `HybridFiltering` techinques to achieve better results.

So far, we have analyzed filtering techniques that tackle the features of items and users. `CollaborativeFiltering` (CF) approaches analyze the user's behaviour directly through its interaction with the system, i.e., the rating activity. `UserBased` CF relies on explicit feedback coming from the users even though this approach suffers from scalability issues in case of extensive data. The `ItemBased` CF technique solves this issue by exploiting users' ratings to compute the item similarity. Finally, `ContextAwareFiltering` involves information coming from the environment, i.e., temperature, geolocalization, and time, to name a few. Though this kind of filtering goes beyond the software engineering domain, we list it to complete the filtering approaches landscape.

The `MemoryBased` approach acts typically on user-item matrixes to compute their distance involving two different methodologies, i.e., `SimilarityMeasure` and `AggregatationApproach`. The former involves the evaluation of the matrix similarity using various concepts of similarity. For instance, `JaccardDistance` measures the similarity of two sets of items based on common elements, whereas the `LevenshteinDistance` is based on the edit distance between two strings. Similarly, the `CosineSimilarity` measures the euclidean distance between two elements. Besides the concept of similarity, techniques based on matrix factorization are employed to make the recommendation engine more scalable. Singular value decomposition (SVD) is a technique being able to reduce the dimension of the matrix and summarize its features. Such a strategy is used to cope with a large amount of data, even though it is computationally expensive. `AggregatationApproaches` analyze relevant statistical information of the dataset such as the variance, mean, and the least square. To mitigate bias lead by the noise in the data, the computation of such indexes use adjusted weigths as a coefficient to rescale the results.

To produce the expected outcomes, `MemoryBased` approaches require the direct usage of the input data that cannot be available under certain circumstances. Thus, `ModelBased` strategies can overcome this limit by generating a model from the data itself. `MachineLearning` offers several models that can support the recommendation activity. `NeuralNetwork` models can learn a set of features and recognize items after a training phase. By exploiting different layers of neurons, the input elements are labeled with different weights. Such values are recomputed during different training rounds in which the model learns how to classify each element according to a predefined loss function. Depending on the number of layers, the internal structure of the network, and other parameters, it is possible to use different kinds of neural networks including Deep Neural Networks (DNN), Recurrent Neural Networks (RNN), Feed-forward Neural Networks (FNN), or Convolutional Neural Networks (CNN). Besides ML models, a recommendation system can employ several models to suggest relevant items. `GeneticAlgorithms` are based on evolutionary principles that hold in the biology domain, i.e., natural species selection. `FuzzyLogic` relies on a logic model that extends classical boolean operators using continuous variables. In this way, this model can represent the real situation accurately. Several probabilistic models can be used in a recommendation system. `BayesianNetwork` is mostly employed to classify unlabeled data, although it is possible to employ it in recommendation activities.

Besides all these well-founded techniques, recommended items can be produced by means of `Heuristics` techniques to encode the knowhow of domain experts. Heuristics employ different approaches and techniques together to obtain better results as well as to

overcome the limitations of other techniques. On the one hand, heuristics are easy to implement as they do not rely on a complex structure. On the other hand, they may produce results that are sub-optimal compared to more sophisticated techniques.

Presenting recommendations As the last phase, the produced recommendation items need to be properly presented to the developer. To this end, several strategies involve potentially different technologies, including the development of extensions for IDEs and dedicated Web-based interfaces. `IDEIntegration` offers several advantages, i.e., auto-complete shortcuts and dedicated views showing the recommended items. The integration is usually performed by the development of a plug-in, as shown in existing recommendation systems (Lv et al. 2015; Ponzanelli et al. 2016). Nevertheless, developing such an artifact requires much effort, and the integration must take into account possible incompatibilities among all deployed components. A more flexible solution is represented by `WebInterfaces` in which the recommendation system can be used as a stand-alone platform. Even though the setup phase is more accessible rather than the IDE solution, presenting recommendations through a web service must handle some issues, including server connections, and suitable response times. For presentation purposes, interactive data structures might be useful in navigating the recommended items. `TraversableGraph` is just one successful example of this category. Strathcona (Holmes et al. 2005) makes use of this technique to show the snippets of code rather than simply retrieving them as ranked lists. In this way, final users can figure out additional details about the recommended items.

4.2 Development challenges for CrossSim and CrossRec

> **DC2 – Measuring similarities among software systems**: Considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail.

In OSS forges like GitHub, there are several connections and interactions, such as development commit to repositories, user star repositories, or projects contain source code files, to mention a few. To conceptualize CrossSim (Nguyen et al. 2020), we came up with the application of a graph-based representation to capture the semantic features among various actors, and consider their intrinsic connections. We modeled the community of developers together with OSS projects, libraries, source code, and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependencies and implications on the others. Graphs allow for flexible data integration and facilitates numerous similarity metrics (Blondel et al. 2004).

We decided to adopt a graph-based representation to deal with the project similarity issue because some of the co-authors already addressed a similar problem in the context of Linked Data. The analogy of the two problems inspired us to apply the similarity technique already developed (Nguyen et al. 2015) to calculate the similarity of representative software projects. The initial evaluations were encouraging and consequently, we moved on by refining the approach and improving its accuracy.

Despite the need to better support software developers while they are programming, very few works have been conducted concerning the techniques that facilitate the search for suitable third-party libraries from OSS repositories. We designed and implemented CrossRec on top of CrossSim: the graph representation was exploited again to compute similarity among software projects, and to provide inputs for the recommendation engine.

Understanding the features that are relevant for the similarity calculation was a critical task, which required many iterations and evaluations. For instance, at the beginning of the work we were including in the graph encoding information about developers, source code, GitHub star events when available, etc. However, by means of the performed experiments, we discovered that encoding only dependencies and star events is enough to get the best performance of the similarity approach (Nguyen et al. 2020).

To sum up, concerning the features shown in Fig. 8, both CrossSim and CrossRec make use of a graph-based representation for supporting the *Data Preprocessing* activity. Concerning the *Producing Recommendation* phase, item-based collaborative filtering techniques have been exploited. For the *Capturing Context* phase, the project being developed is encoded in terms of different features, including used third-party libraries, and README files. Recommendations are presented to the user directly in the used Eclipse-based development environment.

4.3 Development challenges for FOCUS

During the development process, rather than programming from scratch, developers look for libraries that implement the desired functionalities and integrate them into their existing projects (Nguyen et al. 2018). For such libraries, API function calls are the entry point which allows one to invoke the offered functionalities. However, in order to exploit a library to implement the required feature, programmers need to consult various sources, e.g., API documentation to see how a specific API instance is utilized in the field. Nevertheless, from these external sources, there are only texts providing generic syntax or simple usage of the API, which may be less relevant to the current development context. In this sense, concrete examples of source code snippets that indicate how specific API function calls are deployed in actual usage, are of great use (Moreno et al. 2015).

Several techniques have been developed to automate the extraction of API usage patterns (Robillard et al. 2013) in order to reduce developers' burden when manually searching these sources and to provide them with high-quality code examples. However, these techniques, based on clustering (Niu et al. 2017; Wang et al. 2013; Zhong et al. 2009) or predictive modeling (Fowkes and Sutton 2016), still suffer from high redundancy and poor run-time performance.

By referring to the features shown in Fig. 8, differently from other existing approaches which normally rely on clustering to find API calls, FOCUS implements a *context-aware collaborative-filtering* system that exploits the cross relationships among different artifacts in OSS projects to represent them in a graph and eventually to predict the inclusion of additional API invocations. Given an active declaration representing the user *context*, we search for prospective invocations from those in similar declarations belonging to comparable projects. Such a phase is made possible by a proper *data preprocessing* technique, which encodes the input data by means of a *tensor*. The main advantage of our tool is that it can recommend real code snippets that match well with the development context. In contrast with several existing approaches, FOCUS does not depend on any specific set of libraries

and just needs OSS projects as background data to generate API function calls. More importantly, the system scales well with large datasets using the collaborative-filtering technique that filters out irrelevant items, thus improving efficiency. The produced recommendations are shown to the users directly in the Eclipse-based IDE.

> **DC3 - Curating a dataset for training and testing API recommenders requires significant effort**: To provide input data in the form of a tensor, it is necessary to parse projects for extracting their constituent declarations and invocations.

A major obstacle that we needed to overcome when implementing FOCUS is as follows. To provide input data in the form of a tensor, it was necessary to parse projects to extract their constituent declarations and invocations. However, FOCUS relies on Rascal (Basten et al. 2015) to function, which in turn works only with compilable Java source code. To this end, we populated training data for the system from two independent sources. First, we curated a set of Maven jar files which were compilable by their nature. Second, we crawled and filtered out to select only GitHub projects that contain informative *.classpath*, which is an essential requirement for running Rascal. Once the tensor has been properly formulated, FOCUS can work on the collected background data, being independent of its origin. One of the considered datasets was initially consisting of 5,147 Java projects retrieved from the Software Heritage archive (Di Cosmo and Zacchiroli 2017). To satisfy the baseline constraints, we first restricted the dataset to the list of projects that use at least one of the considered third-party libraries. Then, to comply with the requirements of FOCUS, we restricted the dataset to those projects containing at least one pom.xml file. Because of such constraints, we ended up with a dataset consisting of 610 Java projects. Thus, we had to create a dataset ten times bigger than the used one for the evaluation.

4.4 Development challenges of MNBN

In recent years, GitHub has been at the forefront of platforms for storing, analyzing and maintaining the community of OSS projects. To foster the popularity and reachability of their repositories, GitHub users make daily usage of the star voting system as well as forking (Borges and Valente 2018; Jiang et al. 2017). These features allow for increasing the popularity of a certain project, even though the search phase has to cope with a huge number of items. To simplify this task, GitHub has introduced in 2017 the concept of *topics*, a list of tags aiming to describe a project in a succinct way. Immediately after the availability of the topics concept, the platform introduced Repo-Topix (Ganesan 2017) to assist developers when creating new projects and thus, they have to identify representative topics. Though Repo-Topix is already in place, there are rooms for improvements, e.g., in terms of the coverage of the recommended topics, and of the underpinning analysis techniques. To this end, we proposed MNBN (Di Sipio et al. 2020), an approach based on a Multinomial Naive Bayesian network technique to automatically recommend topics given the README file(s) of an input repository.

The main challenges related to the development of MNBN concern three main dimensions as follows: *(i) identification of the underpinning algorithm*, *(ii) creation of the training dataset*, and *(iii) usage of heterogeneous reusable complements*, and they are described below.

> **DC4 – Selection of the right ML algorithms**: Due to the well-known no-free lunch theorem that holds for any Machine Learning (ML) approach (Wolpert and Macready 1997), selecting the suitable model for the problem at hand is one of the possible pitfalls.

Concerning the Machine Learning domain, all relevant results have been obtained through empirical observations undertaken on different assessments. Thus, to better understand the context of the addressed problem we analyzed existing approaches that deal with the issue of text classification using ML models. Among the analyzed tools, the Source Code Classifier (SCC) tool (Alreshedy et al. 2018) can classify code snippets using the MNB network as the underlying model. In particular, this tool discovers the programming language of each snippet coming from StackOverflow posts. The results show that Bayesian networks outperform other models in the textual analysis task by obtaining 75% of accuracy and success rate. Furthermore, there is a subtle correlation between the Bayesian classifier and the TF-IDF weighting scheme (Kibriya et al. 2005). A comprehensive study has been conducted by comparing TF-IDF with the Supporting Vector Machine (SVM) using different datasets. The study varies the MNB parameters to investigate the impacts of the two mentioned preprocessing techniques for each of them. The evaluation demonstrates that the TF-IDF scheme leads to better prediction performance than the SVM technique. Thus, we decided to adopt the mentioned MNBN configuration considering these two findings *(i)* this model can adequately classify text content and *(ii)* the TF-IDF encoding leads benefits in terms of overall accuracy.

> **DC5 – Creation of training datasets for Bayesian networks**: To make the employed Bayesian network accurate, each topic must be provided with a similar number of training projects; otherwise, the obtained results can be strongly affected by the unbalanced distribution of the considered topics.

To mitigate such issues, we decided to train and evaluate the approach by considering 134 GitHub featured topics. In this respect, we analyzed 13,400 README files by considering 100 repositories for each topic. To collect such artifacts, we needed to be aware of the constraints imposed by the GitHub API, which limit the total number of requests per hour to 5,000 for authenticated users and 60 for unauthorized requests.

> **DC6 – Usage of reusable heterogeneous components to develop new recommendation systems**: The orchestration of heterogeneous components was another challenge related to the development of MNBN.

Though the employed Python libraries are flexible, they involve managing different technical aspects, i.e., handling access to Web resources, textual engineering, and language prediction. Moreover, each component has a well-defined set of input elements that dramatically impact on the outcomes. For instance, the README encoding phase cannot occur without the data provided by the crawler component, which gets data from GitHub. In the same way, the topic prediction component strongly relies on the feature extraction per-

formed by the TF-IDF weighting scheme. Thus, we succeeded in developing MNBN by putting significant efforts in composing all the mentioned components coherently.

To summarize, concerning Fig. 8, NLP techniques have been applied to support the data preprocessing phase of MNBN. A model-based approach consisting of a Bayesian network underpins the overall technique to produce recommendations. The user context consists of an input README file, which is mined employing a keyword extraction phase. The produced recommendations are shown to the user directly in the employed Eclipse-based IDE.

4.5 Lessons learned

Developing the CROSSMINER recommendation systems has been a long journey, which allows us to garner many useful experiences. These experiences are valuable resources, which we can rely on in the future whenever we are supposed to run similar projects.

> **DLL1 – Selecting the right representation can be of paramount importance**: A suitable encoding helps capture the intrinsic features of the OSS ecosystem to facilitate the computation of similarities among software projects, moreover it paves the way for various recommendations.

With respect to the features shown in Fig. 8, the used graph representation facilitates different recommendations, i.e., CrossSim, CrossRec, and FOCUS making use of a graph-based representation for supporting the *Data Preprocessing* activity. We selected such a representation since some of the co-authors have gained similar experiences in the past and consequently, we followed the intuition to try with the adoption of graphs, and graph-similarity algorithms also in the mining OSS repositories. We started with CrossSim, and subsequently we found that the graph-based representation is also suitable to develop CrossRec and FOCUS.

> **DLL2 – Do not pretend to immediately find the optimal solution (move on iteratively)**: Conceiving the right techniques and configurations to satisfy user needs can be a long and iterative process.

For conceiving all the recommendation systems we developed in CROSSMINER, we followed an iterative process aiming to find the right underpinning algorithms and configurations to address the considered problems with the expected accuracy. It can be a very strenuous and Carthusian process that might require some stepping back if the used technique gives evidence of inadequacy for the particular problem at hand, fine-tune the used methods, and collect more data both for training and testing. For instance, in the case of CrossSim we had to make four main iterations to identify the features of open source projects relevant for solving the problem of computing similarities among software projects. During the initial iterations, we encoded more than the necessary metadata. For instance, we empirically noticed that encoding information about developers contributing to projects might reduce the accuracy of the proposed project similarity technique.

> **DLL3 – Start first with the techniques you already know and move on from there**: Starting with low-hang fruits allowed us to gain experience of the domain of interest and to quickly produce even sub-optimal results that can still be encouraging while finding better solutions.

During the execution of CROSSMINER, we had a tight schedule, and once we agreed with the requirements by our use case partners, we started first with approaching problems that were similar to those we had already in the past. In other words, we first got low-hang fruits and then moved on from them. In this respect, we began early with CrossSim since we noticed some similarities with the problem that one of the co-authors previously addressed (Nguyen et al. 2015). In this way, we managed to gain additional expertise and knowledge in the domain of recommendation systems, while still satisfying essential requirements elicited from the partners. Afterwards, we succeeded in addressing more complicated issues, i.e., recommending third-party libraries with CrossRec (Nguyen et al. 2019) and API function calls and code snippets with FOCUS (Nguyen et al. 2019, 2021).

5 Challenges and lessons learned from the evaluation of the CROSSMINER recommendation systems

Once the recommendation systems had been realized, it was necessary to compare them with existing state-of-the-art techniques. Evaluating a recommendation system is a challenging task since it involves identifying different factors. In particular, there is no golden rule for evaluating all possible recommendation systems due to their intrinsic features as well as heterogeneity. To evaluate a new system, various questions need to be answered, as they are listed as follows:

– *Which evaluation methodology is suitable?* Assessing RSSE can be done in different ways. Conducting a user study has been accepted as the *de facto* method to analyze the outcome of a recommendation process by several studies (McMillan et al. 2012; Moreno et al. 2015; Ponzanelli et al. 2016; Zhang et al. 2017; Zhong et al. 2009). However, user studies are cumbersome, and they may take a long time to finish. Furthermore, the quality of a user study's outcome depends very much on the participants' expertise and willingness to participate. In this sense, setting up an automated evaluation, in which the manual intervention is not required (or preferably limited), is greatly helpful.
– *Which metric(s) can be used?* Choosing suitable metrics accounts for an important part of the whole evaluation process. While accuracy metrics, such as *success rate*, *precision* and *recall* have been widely used to measure the prediction performance, we suppose that additional metrics should be incorporated into the evaluation (Ge et al. 2010; Nguyen et al. 2019), aiming to study RSSE better.
– *How to prepare/identify datasets for the evaluation?* One needs to take into account different parameters when it comes to choosing a dataset for evaluation. Moreover, the data used to evaluate a system depends very much on the underpinning algorithms. In this sense, advanced techniques and methods for curating suitable data are highly desirable.
– *What could be a representative baseline for comparison?* To show the features of a new conceived tool and give evidence of its novelty and advantages, it is necessary

to compare it with existing approaches with similar characteristics. Since the solution space is vast, comparing and evaluating candidate approaches can be a daunting task.

Answering such questions gives place to different challenges as described in the following subsection.

5.1 Challenges

> **EC1 – Identification of the suitable evaluation methodolgy**: Deciding the evaluation methodolgy to be applied has to take into account several aspects including the time and efforts that have been allocated for such a phase in the project the work is contextualized.

User studies can be done as *field studies* or as *controlled experiments*. By the former, the participants with different programming experience levels have to complete a list of tasks using the proposed recommendation system without any intervention. The latter is conducted in a monitored environment, and the assigned tasks are carefully tailored for specific purposes. Although these strategies produce remarkable results in various work, there are some issues to be tackled. Among others, the selection of the participants has a crucial role to play.

It is worth noting that the selection of ground-truth data from an active project impacts the evaluation, and it might jeopardize the integrity of the evaluation process. Different aspects, i.e., the scope of the recommendation, the recommendation input, the size of the ground truth, and the characteristics of the selected objects, should be carefully considered to mimic a real usage scenario when it comes to an automatized evaluation. For instance, randomly choosing the ground truth size and objects does not guarantee that the evaluation mimics a real usage scenario. The ground truth extraction strategies that have been employed for evaluating the CROSSMINER recommendation systems are explained below.

CrossRec Given a set of libraries that an active project uses, CrossRec returns a set of additional libraries that similar projects to the active project have also included. For this reason, in the CrossRec evaluation process, given an active project, a half of its libraries are used as the ground-truth, and the remaining are used as the query. In this case, we split randomly into such sets the libraries that an active project includes.

FOCUS Given a list of method declaration and method invocations pairs, and an active method context, FOCUS predicts the next method invocations that can be added to the active declaration. To simulate a developer's behaviour at different stages of a development project, we performed various evaluation experiments by varying the size of the recommendation query and the size of the ground-truth data. In particular, four different configurations have been considered in the evaluation to mimic the following scenarios:

- the developer is at an early stage of the development process, and the active method is almost empty;
- the developer is at an early stage of the development process, and the active method implementation is well defined;
- the developer is near to the end of the development process, and the active method is almost empty;

– the project is in an advanced development phase, and the active method implementation is well defined.

The ground truth data is extracted accordingly to the scenario that the evaluation mimics.

MNBN Given an active project, the MNBN recommendation system uses the content of *README* file(s) to recommend relevant GitHub topics. Since the recommendation input does not coincide with the object of the recommendation, we used the whole list of topics that an active project uses as ground-truth.

It is our firm belief that user studies are inevitable in many contexts. For the evaluation of CrossSim, a user study is a must, since there are no other ways to evaluate the similarity between two OSS repositories, rather than the manual scoring done by humans. We may avoid user studies in some specific cases. For instance, when evaluating CrossRec, we realized that with the application of the ten-fold cross-validation technique, we can rely on the available data to perform the evaluation, without resorting to a user study. For FOCUS, while we can use data to evaluate its performance, we assume that its usability and usefulness can be properly studied only with a user study, where developers are asked to give their opinion on a specific API call recommended by the system.

> **EC2 – Selection of the right evaluation metrics**: Deciding which evaluation metrics for evaluating new recommendation systems has to be done by considering several aspects, including the datasets that are available, the existing baselines (if any), and the employed evaluation methodology.

The recommendation outcome is normally a ranked list of items, e.g., third-party libraries (Nguyen et al. 2019; Thung et al. 2013), API calls (Moreno et al. 2015; Nguyen et al. 2019), or GitHub topics (Di Sipio et al. 2020). Normally, a developer pays attention only to the *top-N* items. Thus, by comparing the items in the ranked list with those stored as ground-truth data, we can examine how well the recommendation system performs. There are various metrics to analyze the performance of a recommendation system. To our knowledge, several studies in RSSE focus only on accuracy (Bruch et al. 2008; Fowkes and Sutton 2016; McMillan et al. 2012; Thung et al. 2013). However, in the scope of CROSSMINER, we realized that while accuracy is a good metric for evaluating an RS, it is not enough for studying all the performance traits of the outcomes, as it is the case with conventional recommendation systems (Ge et al. 2010). As a result, other metrics should also be incorporated to analyze various quality aspects as they are presented as follows. First, the following notations are defined:

– N is the cut-off value for the list of recommended items;
– for a testing project p, the ground-truth dataset is named as *GT(p)*;
– $REC(p)$ is the *top-N* items, it is a ranked list in descending order of real scores, with $REC_r(p)$ being the item in the position r;
– if a recommended item $i \in REC(p)$ for a testing project p is found in the ground truth of p, i.e., *GT(p)*, hereafter we call this as a *match* or *hit*.

The metrics that have been employed for evaluating the CROSSMINER recommendation systems are explained below.

Success rate. Given a set of *P* testing projects, this metric measures the rate at which a system can return at least a match among *top-N* recommended items for every project $p \in P$ (Thung et al. 2013). It is formally defined as follows:

$$success\ rate@N = \frac{count_{p \in P}(\left| GT(p) \bigcap (\cup_{r=1}^{N} REC_r(p)) \right| > 0)}{|P|} \tag{1}$$

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

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

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

$$precision@N(p) = \frac{\sum_{r=1}^{N} \left| GT(p) \bigcap REC_r(p) \right|}{N} \tag{2}$$

Recall@N is the ratio of the ground-truth items appearing in the *N* items (Davis and Goadrich 2006; Di Noia et al. 2012; Nguyen et al. 2015):

$$recall@N(p) = \frac{\sum_{r=1}^{N} \left| GT(p) \bigcap REC_r(p) \right|}{|GT(p)|} \tag{3}$$

nDCG Precision and recall reflect well the accuracy, however they neglect ranking sensitivity (Bellogín et al. 2013). nDCG is an effective way to measure if a system can present highly relevant items on the top of the list:

$$nDCG@N(p) = \frac{1}{iDCG} \cdot \sum_{i=1}^{N} \frac{2^{rel(p,i)}}{log_2(i+1)} \tag{4}$$

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

TopRank It measures the percentage of the first top elements in the ground-truth data:

$$Top\ rank = \frac{TpRank(r)}{|R|} \times 100\% \tag{5}$$

where *TpRank(r)* returns 1 if the first predicted element belongs to *UsrTp(r)*, 0 otherwise.

Sales diversity In merchandising systems, *sales diversity* is the ability to distribute the products across several customers (Nguyen et al. 2015; Vargas and Castells 2014). In the context of mining software repositories, sales diversity means the ability of the system to suggest to projects as much items, e.g., libraries, code snippets, as possible, as well as to spread the concentration among all the items, instead of presenting a specific set of them (Robillard et al. 2014).

Catalog coverage measures the percentage of items recommended to projects:

$$coverage@N = \frac{\left| \cup_{p \in P} \cup_{r=1}^{N} REC_r(p) \right|}{|I|} \tag{6}$$

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

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

$$entropy = -\sum_{i \in I} \left(\frac{\#rec(i)}{total} \right) ln \left(\frac{\#rec(i)}{total} \right) \tag{7}$$

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

Novelty The metric gauges if a system is able to expose items to projects. *Expected popularity complement* (EPC) is utilized to measure *novelty* and is defined as follows (Vargas and Castells 2011, 2014):

$$EPC@N = \frac{\sum_{p \in P} \sum_{r=1}^{N} \frac{rel(p,r)*[1-pop(REC_r(p))]}{log_2(r+1)}}{\sum_{p \in P} \sum_{r=1}^{N} \frac{rel(p,r)}{log_2(r+1)}} \tag{8}$$

where $rel(p, r) = |GT(p) \bigcap REC_r(p)|$ represents the relevance of the item at the r position of the *top-N* list to project p; $pop(REC_r(p))$ is the popularity of the item at the position r in the *top-N* recommended list. It is computed as the ratio between the number of projects that receive $REC_r(p)$ as recommendation over the number of projects that are recommended items among the most often recommended ones. Equation 8 implies that the more unpopular items a system recommends, the higher the EPC value it obtains and vice versa.

Confidence Given a pair of *<query, retrieved item>* confidence is the score the evaluator assigns to the similarity between the two items;

Ranking In a ranked list, it is necessary to have a good correlation with the scores given by the human evaluation (Bruch et al. 2008). The Spearman's rank correlation coefficient r_s (Spearman 1904) is used to measure how well a similarity metric ranks the retrieved projects given a query. Considering two ranked variables $r_1 = (\rho_1, \rho_2, .., \rho_n)$ and $r_2 = (\sigma_1, \sigma_2, .., \sigma_n)$, r_s is defined as: $r_s = 1 - \frac{6 \sum_{i=1}^{n} (\rho_i - \sigma_i)^2}{n(n^2-1)}$. We also employed *Kendall's tau* coefficient (Kendall 1938), which is used to measure the ordinal association between two considered quantities. Both r_s and τ range from -1 (perfect negative correlation) to +1 (perfect positive correlation); $r_s = 0$ or $\tau = 0$ implies that the two variables are not correlated.

Recommendation time Being able to provide recommended items in a limited amount of time is important, especially for applications that require instant recommendations. This metric evaluates the duration of time, starting from when a user sends the query until the final recommendations are returned.

Depending on the context, we have to choose a suitable set of metrics to evaluate a recommendation system. For example, with CrossSim we can only make use of *Success rate*, *Confidence*, *Precision*, *Ranking*, and *Execution time* to evaluate the tool, since we relied on a user study. Meanwhile, with CrossRec or FOCUS, since we can use the ten-fold cross validation technique (i.e., by exploiting the testing data which was already split into query and ground-truth data), we evaluated them using *Accuracy*, *Precision*, *Recall*, *Diversity*, and *Novelty*.

> **EC3 – Identification of the datasets that are eligible and available for evaluation**: The datasets used for evaluation very much depend on the recommendation algorithms being used.

For each developed tool, we had to go through the following dimensions related to datasets:

– *Which format?* Depending on the employed recommendation techniques (e.g., collaborative filtering, CNNs, etc.) we had to identify the proper ways to encode the created datasets. For instance, to enable the application of a graph-based similarity algorithm underpinning CrossSim, we had to encode the different features of OSS projects in a graph-based representation. The same datasets needed to be represented in a TF-IDF format to enable the application of FOCUS;

– *Which preprocessing process should be applied to create the dataset?* To minimize the size of the input datasets and thus to make their manipulation efficient, we had to perform different data filtering tasks. For instance, in the case of CrossSim, to enable the application of the employed graph-similarity algorithm, we identified the features that are relevant for the task. For example, information about software developers, source code, and GitHub topics was filtered out from the available datasets even though it was easy to encode all of them as elements in the input graphs. Similar data filtering phases were also performed in CrossRec to enable the recommendation of third-party libraries that might be added in the project under development. Indeed, such data filtering phases have to be performed without compromising the performance (in terms of accuracy, precision, recall, etc.) of the approach under evaluation;

– *Which limitations should we tackle when collecting the dataset?* The primary limitations we experienced when evaluating the CROSSMINER recommendation systems were related to the GitHub APIs restrictions. Unfortunately, the adoption of alternative sources like GHTorrent (Gousios 2013) was not enough due to the lack of needed artifacts such as source code. Knowing such limitations in advance, when collecting projects from GitHub, we decided to save as much data as possible for every single project. The goal was to enable the reuse of the collected data even for perspective evaluations to be done for future recommendation systems to be developed in the context of CROSSMINER.

> **EC4 – Identification of the baseline(s) to compare with**: To showcase the features of a new conceived tool as well as to demonstrate its novelty, it is necessary to compare it with existing approaches with similar features. In fact, choosing the correct baseline is a challenging activity, as to ensure a fair comparison, the baselines to be considered must be endowed with reusable *tools* and *datasets*.

While in general, authors of the selected baselines published their tools and dataset online, it is the case that many of them are faulty, or not well maintained, or even worse: no longer available. In particular, while developing the CROSSMINER recommendation systems we always tried our best to identify the baselines to be used for the evaluations. Unfortunately, often they were not available, which indeed led to difficulties in the evaluation. For instance, for evaluating CrossSim, since the implementations of the baselines

were no longer available for public use, we had to re-implement them by strictly following the descriptions in the original papers (Garg et al. 2004; McMillan et al. 2012; Zhang et al. 2017). That was not possible for evaluating MNBN due to the lack of details in the publicly available documents describing the corresponding baseline. In general, whenever a baseline is selected, and it is not available online, we contacted its authors for the original implementation. It was rare that we got a response from the authors with the tool and/or data. Thus, for the particular cases of the developed recommendation systems, either we re-implemented the tool, as it is the case with CrossSim, or we compared by performing experiments on the datasets that have been used in the original papers, as we did for CrossRec.

Table 2 summarizes the main factors related to the evaluation of our proposed recommendation systems. Depending on the intrinsic characteristics of each tool, different metrics and methodologies were employed to evaluate them. For example, to study Cross-Sim (Nguyen et al. 2018, 2020), a user study with several developers' involvement was the only option since there is no automated method to evaluate the similarity between two OSS projects. Meanwhile, with CrossRec (Nguyen et al. 2019), FOCUS (Nguyen et al. 2019), and MNBN (Di Sipio et al. 2020), we relied only on data to investigate their performance. Moreover, depending on the availability of baselines and quality requirements, we used different evaluation metrics, such as Accuracy (Precision, Recall, TopRank) or Sales Diversity (Coverage, Entropy). Choosing suitable data plays an important role in the evaluations, and it depends on various factors, such as systems' characteristics, baselines, evaluation purposes, or even constraints imposed by OSS platforms, e.g., GitHub and the Maven central repository. The selection of baselines was also a significant issue, considering their complexity and relevance with our tools. For evaluating CrossRec, we were able to consider three different tools for comparison, i.e., LibRec (Thung et al. 2013), LibFinder (Ouni et al. 2017), and LibCUP (Saied et al. 2018). While with FOCUS, only PAM (Fowkes and Sutton 2016) was selected to benchmark since other relevant tools such as MAPO (Zhong et al. 2009) and UP-Miner (Wang et al. 2013) were no longer available. In summary, we believe that there are many factors when it comes to design and evaluate a recommendation system, and we should carefully investigate the most probable scenarios to select the optimal one.

5.2 Lessons learned

ELL1 – User studies are cumbersome, and they can take a long time to be conducted and completed The quality of a user study's outcome depends very much on the participants' expertise and willingness to participate. People are often not very keen on the experiments, since there is no incentive/reward for performing the required tasks. Moreover, there is a trade-off between domain-expert developers, who may not need a recommendation system to develop, and students who have never used this type of system. As a result, we evaluated CrossSim by involving 15 developers of different background of knowledge. Aiming at a reliable evaluation, for each query we mixed and shuffled the top-5 results generated from the computation by each similarity metric in a single Google form and presented them to the evaluators who then inspected and given a score to every pair. Thus, we managed to mimic a *taste test* where users are asked to judge a product, e.g., food or drink, without having a priori knowledge about what is being evaluated (Ghose and Lowengart 2001; Pettigrew and Charters 2008). In this way, we removed any bias or prejudice against a specific similarity metric. The participants were asked to label the similarity for each pair of query and retrieved project regarding their application domains and functionalities. Furthermore, we also allowed for cross-checking, i.e., the results of one developer were validated by the others. To perform such evaluation for CrossSim and compare it with the baselines, it has been

Table 2 CROSSMINER recommendation systems: evaluation facts

		CrossSim (Nguyen et al. 2018)	CrossRec (Nguyen et al. 2019)	FOCUS (Nguyen et al. 2019, 2021)	MNBN (Di Sipio et al. 2020)
Methodology	Cross-Val.		✓	✓	✓
	User study	✓			
Metric	Success rate	✓			
	Precision	✓	✓	✓	✓
	Recall		✓	✓	✓
	nDCG		✓		
	TopRank		✓		✓
	Coverage		✓		
	Entropy		✓		
	Novelty		✓		
	Confidence	✓			
	Ranking	✓			
	Time	✓		✓	
Dataset	Source	GitHub	GitHub	GitHub, Maven central repository	GitHub
	Size	580 projects	1,200 projects	3,600 projects	13,400 projects
Artifact		Metadata	Metadata	Source code	README files
Baseline		MUDABlue (Garg et al. 2004), CLAN (McMillan et al. 2012), RepoPal (Zhang et al. 2017)	LibRec (Thung et al. 2013), LibFinder (Ouni et al. 2017), LibCUP (Saied et al. 2018)	UP-Miner (Wang et al. 2013), PAM (Fowkes and Sutton 2016)	None

crucial to design the experimental settings properly and clearly define the manual evaluation tasks by adhering to the taste-test methodology.

ELL2 – In some certain contexts, the k-fold cross-validation technique is a good alternative to user studies As previously mentioned, through CROSSMINER, we realized that user studies are cumbersome, and they can take a long time to conduct and complete. However, we experienced that the assessment can also be automatized by means of case studies or data itself. By the former, use cases are pre-selected for the recommendation. By the latter, we set up an automated evaluation, in which the manual intervention is not required, or preferably limited. Depending on the availability of data, we managed to avoid performing user studies by employing the *k-fold cross validation* technique (Wong 2015), which has been popularly chosen as the evaluation method for a model in Machine Learning. By this method, a dataset is divided into *k* equal parts (*folds*). For each validation round, one fold is used as a testing and the remaining *k-1* folds are used as training data. Such an evaluation attempts to mimic a real scenario: *the system should produce recommendations for a project based on the data available from a set of existing projects*. The artifact being considered as the recommendation target is called *object*. For instance, regarding third-party libraries recommendation (Nguyen et al. 2019; Thung et al. 2013), objects are libraries that a system provides as its outcome. It is essential to study if the recommendation system is useful by providing the active project with relevant libraries, exploiting the training data. To this end, we keep a certain amount of objects for each active project and use them as input for the recommendation engine, which can be understood as the query. The rest is taken out and used as *ground-truth data*. The ground-truth data is compared with the recommendation outcomes to validate the system's performance. It is expected that the recommendation system can retrieve objects that match up the ones stored as ground-truth data.

ELL3 – The quality of data depends on the particular application domain of interest Through CROSSMINER, we further confirmed the importance of having the availability of big data and high-quality data for training and evaluation activities. The definition of data quality cannot be given in general, and it very much depends on the particular application of interest. According to our experience, creating a dataset, which can be rightly used for both training and evaluating the developed recommendation systems, can require significant effort, which can be comparable to that needed to realize the conceived approach. For instance, to implement MNBN, we devoted a huge effort to create the dataset that was supposed to be balanced with respect to the considered GitHub featured topics. Moreover, it can be challenging to collect big datasets, especially when there are several constraints to be satisfied. For instance, in the case of the FOCUS evaluation, one of the considered datasets was initially consisting of 5,147 Java projects retrieved from the Software Heritage archive (Di Cosmo and Zacchiroli 2017). To comply with the requirements of the baseline, we first restricted the dataset to the list of projects that use at least one of the considered third-party libraries. Then, to comply with the requirements of FOCUS, we restricted the dataset to those projects containing at least one `pom.xml` file. Because of such constraints, we ended up with a dataset consisting of 610 Java projects. Thus, we had to create a dataset ten times bigger than the used one for the evaluation.

ELL4 – Candidate baselines might not be reusable When conceiving new recommendation systems there can be no baselines to compare with. There are at least two motivations: *(i)* the proposed approach is the first attempt dealing with the considered problem; *(ii)* the tools and datasets of existing baselines are no longer available or reusable. In such cases,

according to the facts shown in Table 2, the k-fold cross-evaluation has been a valuable technique that allowed us to evaluate most of the proposed recommendation systems even when the baselines were not available. Concerning CrossSim, we decided to perform a user study, to mitigate any bias related to the fact that we re-implemented all the baselines.

ELL5: Novelty and diversity are good indicators that are worth considering Many existing approaches just choose to recommend *popular* items, e.g., USE (Moreno et al. 2015), PROMPTER (Ponzanelli et al. 2016), LibRec (Thung et al. 2013). Through the evaluation of CrossRec, we demonstrated that further than popularity, *novelty* and *diversity* are good indicators for assessing if the recommendation outcomes are meaningful. Among others, the ability to recommend items in the long tail is essential: we can suggest things even when extremely unpopular since a small number of projects use each. However, they turn out to be useful as all of them match those stored as ground-truth. This implies that the novelty of a ranked list is important: a system should recommend libraries that are *novel* (Castells et al. 2011), i.e., those that have been rarely seen. In this sense, we see that CrossRec can produce good outcomes, not only in terms of success rate and accuracy but also *sales diversity* and *novelty*. Moreover, *serendipity* has been widely exploited to evaluate recommendation systems in other domains. Serendipity means that items are obtained by chance but turn out to be useful. However, it seems that the metric has been neglected in evaluating recommendation systems in software engineering. Investigating the importance of serendipity in the context of source code/library recommendation can be an interesting topic. For example, a recommendation engine provides a developer with an artifact, e.g., a third-party library or an API function call, which does not belong to the ground-truth data at all; however, it is indeed useful for the current project.

6 Related work

In this section we provide a literature review on the development and usage of recommendation systems in software engineering. More importantly, we associate our work to various existing studies, aiming to highlight its main contributions.

In their book (Robillard et al. 2014), Robillard et al. focus on the techniques and applications of recommendation systems in software engineering. The work presents a pragmatic approach to system design, implementation, and evaluation. Similarly, Proksch et al. (2014) present a comprehensive report on different phases that need to be considered when developing an effective recommender system to support the development activities. Though these studies are highly related to our work, they provide a set of guidelines for developing and evaluating a generic recommendation system. In other words, such guidelines are not tailored to any specific recommendation systems. There is a lack of proper references for anyone who wants to customize their implementation for a specific context. For example, a developer is interested in understanding which techniques can be used for producing recommendations; or which evaluation metrics are suitable for the results obtained by conducting a user study. And this is where our work comes in: we complement the existing studies by reporting on a specific use case, i.e., recommendation systems developed through the CROSSMINER project to satisfy requirements imposed by various industrial partners. More importantly, we provide the community at large with detailed challenges and lessons. In this respect, our work is expected to be a practical benchmark, when it comes to design and implementation of a recommendation system for mining OSS repositories.

Pakdeetrakulwong et al. (2014) investigate the impact of recommendation systems on the software development life cycle (SDLC). By analyzing several state-of-the-art studies, they identified three main components of a recommendation system, i.e., a mechanism to collect data, a recommendation engine, and a user interface to deliver recommendations. A recommendation system should support a developer throughout the SDLC phases, ranging from design to testing. Among others, the most supported one by existing recommender tools is the implementation phase, in which software engineers turn components designed in a previous phase into code. Although implementation is a crucial phase in the software engineering (SE) domain, the other phases are required, i.e., collecting requirements, design phase, and testing. To be more concrete, a recommendation system should be able to provide several types of items, including UML diagrams and artifacts. A promising filed is Semantic Web and Ontologies, which are used to describe software components in SE that allow information sharing among team members. Moreover, most of RSSEs use a pull approach to deliver recommendations: it means that recommended items are provided without any specific request made by the user. Reversely, the work (Pakdeetrakulwong et al. 2014) suggests using a push approach, in which the user can trigger the delivery of the recommendations reactively.

Maki et al. (2015) propose a feature model to represent the problem of capturing contexts in the RSSE domain. As a preliminary analysis, the authors discuss 23 papers and classify them in the following six categories:

– *Change task*: this type of recommendation system aims to support the developer in managing the evolution of the current programming task;
– *API usage*: this type of RSSEs supports the usage of external third-party libraries;
– *Refactoring task*: recommendation systems that support refactoring activities fall in this category;
– *Solving exception, failure, and bug*: this kind of RSSEs handles the exception and unexpected behaviours of the considered software systems;
– *Recommending software components and components' design*: it recommends entire software components to be integrated into the software projects under development;
– *Exploring local codebases and visited source locations*: this type of system supports the information search over different data sources, i.e., online datasets

In the same paper (Maki et al. 2015), the authors show that the examined tools work in practice, but they fail to address the *capturing context* phase accurately. Such a phase plays an essential role in the overall recommendation process, as it is performed at the early stages of the production. According to the authors (Maki et al. 2015) the context extraction phase should be triggered *(i) reactively* or *(ii) proactively*. The former is led by certain actions captured in the development environment, e.g., page scrolling or considering idle times. Contrariwise, the latter is activated directly by the user. Then, the capturing phase is performed by setting the scope and the elements to be extracted. The scope dimension depends on the goal that the RSSE wants to achieve. Thus, recommendation systems can consider as snippets of code as the context as well as the entire project. Considering the possible extracted elements, they can be related to valuable elements of the code, i.e., variables, methods or identifiers. Similarly to the previous phase, the treatment of the excepted data can be different according to the RSSE's aims. Standard techniques to perform this step involve parsing, weighting or filtering. The final step is the delivery of the recommendations, according to various output formats, i.e., bag-of-words, AST, dendrograms, annotated graphs, and weighted vectors. The findings of the work suggest that most of the

analyzed tools do not cover the context extraction activity properly. Thus, there are room of improvements in this filed that can bring substantial contribution in the RSSE domain.

Happel et al. (2008) discuss relevant issues that a recommendation system must address: *context-awareness*, *pro-activeness* of the system, and appropriate *knowledge representation* are the main factors that impact on the quality of recommendations. Context-awareness becomes very relevant in software projects in which developers collaboratively work on shared resources. The pro-activeness of recommendation systems still demands further research due to the limited maturity of existing tools concerning such an aspect. A proactive approach should improve the accuracy of the recommendation by reducing the scope of the context. Finally, recommendation systems should take into account more flexible representations of knowledge by considering new techniques, e.g., Semantic Web analysis to make the system more transparent. After these considerations, the authors suggest that a combination of the context-awareness and the information provision can significantly improve the quality of the recommended items. In our work, we focus on recommendation systems for mining OSS forges, e.g., GitHub, the Maven Central Repository, or Stack Overflow. Moreover, we tailor their design to satisfy different requirements of our industrial partners.

Gasparic and Janes (2016) provide a systematic literature review (SLR) on RSSE tools. The study aims to characterize RSSEs following the software engineers' perspective. In particular, the SLR focuses on the required inputs, on the benefits offered by the recommendation process, and on the required effort to provide the recommended items. As the topic is too broad, the authors put some constraints in building the research query. Recommendation systems that do not belong to the software engineering domain are excluded from the analysis. Among these, the study considers only stable tools, i.e., not snapshot versions or reusable components. Additionally, the considered tools strictly support source code development and exclude other types of activities. Considering these boundaries, they conduct the review by analyzing every aspect of the process from the input extraction phase to the retrieved item. After an iterative process, 46 papers are selected for consideration. The major finding of the work is that most of the considered RSSEs use a reactive approach to provide recommendations. Reversely, proactive recommendations are not popular yet. The analyzed tools are focused on the development phase, but the testing phase is crucial in SE projects. Thus, future RSSEs should consider it as a possible application domain. Moreover, the examined context extraction techniques are not able to excerpt broader context, i.e., they miss crucial elements useful to recommend valuable items. Concerning the presentation layer, tools should improve the explanation of the given recommendations and provide users with more information about their usage in concrete situations, i.e., context-aware recommendations.

A taxonomy of recommendation systems as well as the possible phases involved in the recommendation process has been provided (Isinkaye et al. 2015). The work describes recommendation process as an iterative cycle represented by three steps, i.e., *information gathering*, *learning*, and *recommendations delivery*. Information gathering involves the user's feedback collection at the beginning of the entire process after the system has issued the wanted recommendation. The user profile is considered to retrieve the proper items by looking at the user's needs in the presentation. Concerning information gathering, the authors classify possible feedback into *explicit*, *implicit* and *hybrid*. Explicit feedback is given directly by the user, i.e., through ratings which contain accurate information, but require effort to obtain. On the other hand, implicit feedback is inferred directly by the system without involving the user. However, they are less accurate than the explicit ones. The hybrid feedback is a combination of the previous two techniques. Typical implementations use the inferred data to check the feedback given by the user or allow the user to provide only

a subset of information. Then, the learning phase employs them to excerpt the user's characteristics and to build a custom profile. Overall, this study (Isinkaye et al. 2015) highlights the contribution of different techniques as well as their strengths and weakness considering several factors, i.e., availability of the meta-data, user ratings, and the learner model, to name a few. In the present paper, we conceptualized a novel taxonomy for the main design features for recommendation systems in software engineering. Such a taxonomy is intended to provide system designers with the most pertinent technical details for their problems, i.e., tailoring the design to satisfy the requirements imposed by use-case partners.

An extensive guideline that deals with challenges, issues, and basic blocks related to the development of RSSEs has been provided (LASER 2015) to summarize well-established practices to show all the phases needed for the development of such systems. The first step involves *problem framing*, i.e., the identification of context, the tasks to be completed, and the target users of the recommended items. The context represents the development environment which brings plenty of information about the current task, which is the issue or functionality that the user is addressing. Finally, the target user determines what kind of recommended items have to be provided and when: a novice developer's needs are profoundly different from those of expert ones. Thus, the final recommendation items might be completely different according to their experience level. Considering these aspects, the authors grouped RSSEs into four main categories:

– *Hotspot recommender*: it provides recommendations about methods and classes that belong to the current context;
– *Navigation recommender*: it suggests locations where the developer can find hints related to the current task;
– *Snippet recommender*: it produces snippets of code related to the developer's context;
– *Documentation recommender*: it aggregates posts coming from websites and A&Q forums to enhance the documentation of the APIs of interest.

Moving to the *recommendation process*, input sources must be handled to capture the developer's context. Due to their heterogeneity, RSSEs can employ different strategies and choose the most suitable for a certain context, such as static analysis, user feedback, structuring or destructuring techniques. This phase is usually followed by a preprocessing phase, in which data is rearranged for the following steps. Then, a recommendation algorithm is performed to obtain the recommendation item. According to the study (LASER 2015), recommendation algorithms fall in one of the following classes:

– *Heuristic approaches* make effort on the implementation and usually derive from empirical evaluation;
– *Data mining and machine learning techniques* are adopted when a large amount of data are available, by exploiting different algorithm and models;
– *Collaborative filtering* typically employs user-item matrices to filter data and find similar items.

The outcome of the recommendation algorithm is delivered to the target user in the *presenting recommendation* phase. It is characterized by the level of interaction with the target user, which defines if the RSSE is proactive or reactive.

The work presented in this paper shares several similarities with the studies previously outlined. However, by leveraging the experiences we developed in the context of the CROSSMINER project, the presented challenges and lessons learned can complement the previous attempts of conceptualizing recommendation systems with the aim of identifying their strengths and limitations when being applied in the context of software development.

7 Conclusion and future work

Developing complex software systems by reusing existing open source components is a challenging task. In the EU CROSSMINER project we worked on dealing with such a problem by conceiving several recommendation systems to meet the needs identified by six use-case partners working on different domains including IoT, multi-sector IT services, API co-evolution, software analytics, software quality assurance, and OSS forges.

In this paper, we presented an experience report on the various recommendation systems that have been developed in CROSSMINER. We attempt to share with the community the main challenges we had to overcome as well as the corresponding lessons during the three different phases to build and evaluate recommendation systems, i.e., requirement, development, and evaluation.

Being focused on heterogeneous recommendation systems allowed us to garner many useful experiences and learn important lessons. In the first place, the process yields up a list of actionable items when designing and implementing recommendation systems , namely: *(i)* the skepticism that final users can have especially at the early stages of the development and usage of the proposed recommendation systems; *(ii)* difficulties in retrieving and creating datasets to be used both for training and evaluation purposes; *(iii)* criticalities related to the selection of baselines for evaluation especially when the related tools are no longer available; *(iv)* the variety of evaluation approaches and metrics that can be employed to assess the strengths and limitations of the conceived recommendation systems.

The first contribution of our work is a taxonomy of the main design features for recommendation systems in software engineering. We believe that such a taxonomy would come in handy for those who perform a fresh start on investigating which techniques are most suitable for their problems, i.e., tailoring their design to meet the requirements imposed by industrial use-case partners. *The second contribution* of the paper is a benchmark consisting of the challenges and lessons learned that might be useful for developers that need to conceive new recommendation systems and thus, have to encompass the three related phases, i.e., requirement elicitation, development, and evaluation. For instance, through the evaluation of various systems, we realized that the selection of suitable evaluation metrics helps shed light on the performance traits that cannot be revealed by conventional indicators. In particular, while precision and recall represent the right choices for assessing the quality of the proposed recommendation systems, additional metrics typically used in entirely different domains like sales diversity, and serendipity can also be useful for studying a system's performance. Among others, the ability to recommend items in the long tail is important: we can recommend items even when they are extremely unpopular since each is used by a small number of projects. However, they turn out to be useful as all of them match those stored as ground-truth. 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.

For future work, we plan to consolidate the lessons learned by applying in the Model Driven Engineering domain the techniques and tools we developed in CROSSMINER. Moreover, we are also working on a low-code infrastructure to support the development of recommendation systems. In particular, by relying on the presented taxonomy, we developed a metamodel to represent and manage the peculiar components of recommendation systems (Di Sipio et al. 2020). Dedicated supporting tools are also under development to enable citizen developers to easily model and build their custom recommendation systems. The results obtained so far are encouraging even though there is still significant work to be done to enable the development of recommendation systems in a low-code manner.

References

Alreshedy K., Dharmaretnam D., German D. M., Srinivasan V., Gulliver T. A. (2018) SCC: automatic classification of code snippets. CoRR arXiv:1809.07945

Basten B., Hills M., Klint P., Landman D., Shahi A., Steindorfer M. J., Vinju J. J. (2015) M3: a general model for code analytics in rascal. In: 2015 IEEE 1st international workshop on software analytics (SWAN), pp 25–28

Bellogín A., Cantador I., Castells P. (2013) A comparative study of heterogeneous item recommendations in social systems. Information Science 221:142–169. https://doi.org/10.1016/j.ins.2012.09.039

Blondel V. D., Gajardo A., Heymans M., Senellart P., Dooren P. V. (2004) A measure of similarity between graph vertices: applications to synonym extraction and web searching. SIAM Review 46:647–666. https://doi.org/10.2307/20453570

Bobadilla J., Ortega F., Hernando A., Gutiérrez A. (2013) Recommender systems survey. Knowledge-Based Systems 46:109–132

Borges H., Valente M. T. (2018) What's in a GitHub star? Understanding repository starring practices in a social coding platform. Journal of Systems and Software 146:112–129. arXiv:1811.07643. https://doi.org/10.1016/j.jss.2018.09.016

Bruch M., Schäfer T., Mezini M. (2008) On evaluating recommender systems for API usages. In: Proceedings of the 2008 international workshop on recommendation systems for software engineering, RSSE '08. ACM, New York, pp 16–20

Castells P, Vargas S, Wang J (2011) Novelty and diversity metrics for recommender systems: choice, discovery and relevance. In: International workshop on diversity in document retrieval (DDR 2011) at the 33rd European conference on information retrieval (ECIR 2011). Dublin, Ireland. http://ir.ii.uam.es/rim3/publications/ddr11.pdf

Cosentino V., Cánovas Izquierdo J. L., Cabot J. (2017) A systematic mapping study of software development with github. IEEE Access 5:7173–7192. https://doi.org/10.1109/ACCESS.2017.2682323

Czarnecki K. (2002) *Domain engineering*. pp. 433–444. American Cancer Society. https://doi.org/10.1002/0471028959.sof095. https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof095

Dagenais B., Ossher H., Bellamy R. K. E., Robillard M. P., de Vries J. P. (2010) Moving into a new software project landscape. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering - volume 1, ICSE '10. ACM, New York, pp 275–284

Davis J., Goadrich M. (2006) The relationship between precision-recall and ROC curves. In: Proceedings of the 23rd international conference on machine learning, ICML '06. ACM, New York, pp 233–240

Di Cosmo R., Zacchiroli S. (2017) Software heritage: why and how to preserve software source code. In: 14th international conference on digital preservation, pp 1–10. Kyoto

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, I-semantics '12. ACM, New York, pp 1–8

Di Rocco J., Di Ruscio D., Di Sipio C., Nguyen P., Rubei R. (2020) Topfilter: an approach to recommend relevant github topics. In: Proceedings of the 14th ACM / IEEE international symposium on empirical

software engineering and measurement (ESEM), ESEM '20. Association for Computing Machinery, New York. https://doi.org/10.1145/3382494.3410690

Di Sipio C., Di Ruscio D., Nguyen P. T. (2020) Democratizing the development of recommender systems by means of low-code platforms. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings, MODELS '20. Association for Computing Machinery, New York. https://doi.org/10.1145/3417990.3420202

Di Sipio C., Rubei R., Di Ruscio D., Nguyen P. T. (2020) Using a multinomial naïve bayesian (MNB) network to automatically recommend topics for github repositories. In: Proceedings of the 24th international conference on evaluation and assessment in software engineering, EASE2020, Trondheim, Norway, April 15-17, 2020, EASE'20. ACM, pp 24–34. https://doi.org/10.1145/3383219.3383227

Fowkes J., Sutton C. (2016) Parameter-free probabilistic API mining across GitHub. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016. ACM, New York, pp 254–265

Ganesan K. (2017) Topic suggestions for millions of repositories - The GitHub Blog. https://github.blog/2017-07-31-topics/

Garg P. K., Kawaguchi S., Matsushita M., Inoue K. (2004) MUDABLue: an automatic categorization system for open source repositories. In: 2013 20th Asia-Pacific software engineering conference (APSEC), pp 184–193

Gasparic M., Janes A. (2016) What recommendation systems for software engineering recommend: a systematic literature review. Journal of Systems and Software 113:101–113. https://doi.org/10.1016/j.jss.2015.11.036. https://linkinghub.elsevier.com/retrieve/pii/S0164121215002605

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, RecSys '10. Association for Computing Machinery, New York, pp 257–260. https://doi.org/10.1145/1864708.1864761

Ghose S., Lowengart O. (2001) Taste tests: impacts of consumer perceptions and preferences on brand positioning strategies. Journal of Targeting, Measurement and Analysis for Marketing 10(1):26–41

Gomez-Uribe C. A., Hunt N. (2015) The netflix recommender system: algorithms, business value, and innovation. ACM Transactions on Management Information Systems 6(4):13:1–13:19

Gousios G. (2013) The ghtorrent dataset and tool suite. In: Proceedings of the 10th working conference on mining software repositories, MSR '13. IEEE Press, Piscataway, pp 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

Happel H. J., Maalej W. (2008) Potentials and challenges of recommendation systems for software development. In: Proceedings of the 2008 international workshop on Recommendation systems for software engineering - RSSE '08. ACM Press, Atlanta, Georgia, p 11

Holmes R., Walker R. J., Murphy G. C. (2005) Strathcona example recommendation tool. SIGSOFT Softw. Eng. Notes 30(5):237–240. https://doi.org/10.1145/1095430.1081744

Isinkaye F., Folajimi Y., Ojokoh B. (2015) Recommendation systems: principles, methods and evaluation. Egyptian Informatics Journal 16(3):261–273

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

Karlsson E. A. (ed) (1995) Software reuse: a holistic approach. Wiley, New York

Kendall M. G. (1938) A new measure of rank correlation. Biometrika 30(1/2):81–93. http://www.jstor.org/stable/2332226

Kibriya A. M., Frank E., Pfahringer B., Holmes G. (2005) Multinomial naive bayes for text categorization revisited. In: Webb G. I., Yu X. (eds) AI 2004: advances in artificial intelligence. Springer, Berlin, pp 488–499

LASER (2015) LASER: software engineering: international summer schools, LASER 2013-2014, Elba, Italy: revised tutorial lectures. No. 8987 in Lecture notes in computer science Programming and software engineering. Springer, Cham

Linden G., Smith B., York J. (2003) Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Computing 7(1):76–80

Lv F., Zhang H., Lou J. G., Wang S., Zhang D., Zhao J. (2015) Codehow: effective code search based on API understanding and extended boolean model (E). In: 30th IEEE/ACM international conference on automated software engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, pp 260–270

Maki S., Kpodjedo S., Boussaidi G. E. (2015) Context extraction in recommendation systems in software engineering: a preliminary survey. In: CASCON '15, Markham, Canada. IBM Corp., USA, pp 151–160

McMillan C., Grechanik M., Poshyvanyk D. (2012) Detecting similar software applications. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway, pp 364–374

McMillan C., Poshyvanyk D., Grechanik M. (2010) Recommending source code examples via API call usages and documentation. In: Proceedings of the 2Nd international workshop on recommendation systems for software engineering, RSSE '10. ACM, New York, pp 21–25

Moreno L., Bavota G., Di Penta M., Oliveto R., Marcus A. (2015) How can I use this method? In: 37th international conference on software engineering. IEEE, Piscataway, pp 880–890

Nguyen P., 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, WWW '15 companion. Association for Computing Machinery, New York, pp 477–1482. https://doi.org/10.1145/2740908.2742141

Nguyen P. T., Di Rocco J., Di Ruscio D. (2018) Mining software repositories to support OSS developers: a recommender systems approach. In: Proceedings of the 9th italian information retrieval workshop, Rome, Italy, May, 28-30, 2018

Nguyen P. T., Di Rocco J., Di Ruscio D., Di Penta M. (2019) CrossRec: supporting software developers by recommending third-party libraries. *Journal of Systems and Software*, 110460. https://doi.org/10.1016/j.jss.2019.110460. http://www.sciencedirect.com/science/article/pii/S0164121219302341

Nguyen P. T., Di Rocco J., Di Ruscio D., Ochoa L., Degueule T., Di Penta M. (2019) FOCUS: A recommender system for mining API function calls and usage patterns. In: Proceedings of the 41st international conference on software engineering, ICSE '19. IEEE Press, Piscataway, pp 1050–1060

Nguyen P. T., Di Rocco J., Di Sipio C., Di Ruscio D., Di Penta M. (2021) Recommending api function calls and code snippets to support software development. *IEEE Trans Softw Eng*, 1–1. https://doi.org/10.1109/TSE.2021.3059907

Nguyen P. T., Di Rocco J., Rubei R., Di Ruscio D. (2018) Crosssim: exploiting mutual relationships to detect similar OSS projects. In: 2018 44th euromicro conference on software engineering and advanced applications (SEAA), pp 388–395

Nguyen P. T., Di Rocco J., Rubei R., Di Ruscio D. (2020) An automated approach to assess the similarity of GitHub repositories. *Software Quality Journal*. https://doi.org/10.1007/s11219-019-09483-0

Nguyen P. T., Tomeo P., Di Noia T., Di Sciascio E. (2015) Content-based recommendations via DBpedia and freebase: a case study in the music domain. In: Proceedings of the 14th international conference on the semantic web - ISWC 2015 - volume 9366. Springer, New York, pp 605–621

Niu H., Keivanloo I., Zou Y. (2017) API usage pattern recommendation for software development. Journal of Systems Software 129:127–139

Noia T. D., Ostuni V. C. (2015) Recommender systems and linked open data. In: Faber W., Paschke A. (eds) Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures, Lecture Notes in Computer Science, vol 9203. Springer, pp 88–113. https://doi.org/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. Information and Software Technology 83(C):55–75

Pakdeetrakulwong U., Wongthongtham P., Siricharoen W. V. (2014) Recommendation systems for software engineering: a survey from software development life cycle phase perspective. pp. 137–142. IEEE

Pettigrew S., Charters S. (2008) Tasting as a projective technique. Qualitative Market Research: An International Journal 11(3):331–343

Ponzanelli L., Bavota G., Di Penta M., Oliveto R., Lanza M. (2016) Prompter: turning the IDE into a self-confident programming assistant. *Empirical Software Engineering* 21(5), 2190–2231. https://doi.org/10.1007/s10664-015-9397-1. http://link.springer.com/10.1007/s10664-015-9397-1

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

Proksch S., Bauer V., Murphy G. C. (2014) How to build a recommendation system for software engineering. In: Meyer B., Nordio M. (eds) Software engineering - international summer schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures, Lecture Notes in Computer Science, vol 8987. Springer, pp 1–42. https://doi.org/10.1007/978-3-319-28406-4_1

Proksch S., Bauer V., Murphy G. C. (2015) How to build a recommendation system for software engineering. In: Meyer B., Nordio M. (eds) Advances in the theory and practice of software engineering - LASER 2013-2014, LNCS, vol 8987. Springer, pp 1–42. http://tubiblio.ulb.tu-darmstadt.de/77729/

Robillard M. P., Bodden E., Kawrykow D., Mezini M., Ratchford T. (2013) Automated API property inference techniques. IEEE Transactions on Software Engineering 39(5):613–637

Robillard M. P., Maalej W., Walker R. J., Zimmermann T. (eds) (2014) Recommendation systems in software engineering. Springer, Berlin

Rubei R., Di Sipio C., Nguyen P. T., Di Rocco J., Di Ruscio D. (2020) PostFinder: mining stack overflow posts to support software developers. *Information and Software Technology* 127,

106367. https://doi.org/10.1016/j.infsof.2020.106367. http://www.sciencedirect.com/science/article/pii/S0950584920301361

Saied M. A., Ouni A., Sahraoui H., Kula R. G., Inoue K., Lo D. (2018) Improving reusability of software libraries through usage pattern mining. Journal of Systems Software 145:164–179

Schedl M., Zamani H., Chen C., Deldjoo Y., Elahi M. (2018) Current challenges and visions in music recommender systems research. *International Journal Multimedia Information Retrieval* 7(2), 95–116. https://doi.org/10.1007/s13735-018-0154-2

Spearman C. (1904) The proof and measurement of association between two things. The American Journal of Psychology 15(1):72–101

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

Thung F., Lo D., Lawall J. (2013) Automated library recommendation. In: 2013 20th working conf. on reverse engineering (WCRE), pp 182–191

Vargas S., Castells P. (2011) Rank and relevance in novelty and diversity metrics for recommender systems. In: Proceedings of the fifth ACM conference on recommender systems, RecSys '11. ACM, New York, pp 109–116

Vargas S., Castells P. (2014) Improving sales diversity by recommending users to items. In: Eighth ACM conference on recommender systems, recsys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014, pp 145–152

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: 10th working conference on mining software repositories. IEEE, Piscataway, pp 319–328. https://doi.org/10.1109/MSR.2013.6624045

Wolpert D., Macready W. (1997) No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1):67–82. https://doi.org/10.1109/4235.585893. http://ieeexplore.ieee.org/document/585893/

Wong T. T. (2015) Performance evaluation of classification algorithms by K-fold and leave-one-out cross validation. Pattern Recognition 48(9):2839–2846. https://doi.org/10.1016/j.patcog.2015.03.009

Wu L., Shah S., Choi S., Tiwari M., Posse C. (2014) The browsemaps: Collaborative filtering at LinkedIn. In: RSWEb@recsys, CEUR workshop proceedings, vol 1271. CEUR-WS.org

Zhang Y., Lo D., Kochhar P. S., Xia X., Li Q., Sun J. (2017) Detecting similar repositories on GitHub. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), vol 00, pp 13–23

Zheng M., Pan X., Lillis D. (2018) CodEX: source code plagiarism detection based on abstract syntax tree. In: Proceedings for the 26th AIAI irish conference on artificial intelligence and cognitive science trinity college Dublin, Dublin, Ireland, December 6-7th, 2018, pp 362–373

Zhong H., Xie T., Zhang L., Pei J., Mei H. (2009) MAPO: mining and recommending API usage patterns. In: 23rd European conference on object-oriented programming. Springer, Berlin, pp 318–343

Juri Di Rocco is a postdoctoral researcher at the University of L'Aquila, Italy. He obtained a PhD in Computer Science from the University of L'Aquila. He is interested in several aspects of software language engineering and Model Driven Engineering (MDE) including domain specific modelling languages, model transformation, model differencing, modelling repositories and mining techniques. More information is available at https://jdirocco.github.io/. Contact him at juri.dirocco@univaq.it.

Davide Di Ruscio is an 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, coupled evolution, and recommendation systems. He has published more than 150 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 IEEE Software, of the Journal of Object Technology, and of the IET Software journal. More information is available at http://people.disim.univaq.it/diruscio/. Contact him at davide.diruscio@univaq.it.

Claudio Di Sipio is a PhD student at the Unversity of L'Aquila, Italy. He is working on mining techniques to analyse open source software and he is also investigating the application of low-code platforms to support the development of recommendation systems. Contact him at claudio.disipio@graduate.univaq.it.

Phuong T. Nguyen is a postdoctoral researcher at the University of L'Aquila, Italy. He obtained a PhD 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, and Recommender Systems. Recently, he has been working to develop recommender systems in Software Engineering for mining open source code repositories. Contact him at phuong.nguyen@univaq.it.

Riccardo Rubei is a PhD student at the Unversity of L'Aquila, Italy. He is working on mining techniques to analyse open source software with the aim of providing developers with useful real-time recommendations. Contact him at riccardo.rubei@graduate.univaq.it.

Affiliations

Juri Di Rocco[1] · Davide Di Ruscio[1] ⒾD · Claudio Di Sipio[1] · Phuong T. Nguyen[1] · Riccardo Rubei[1]

Juri Di Rocco
juri.dirocco@univaq.it

Claudio Di Sipio
claudio.disipio@graduate.univaq.it

Phuong T. Nguyen
phuong.nguyen@univaq.it

Riccardo Rubei
riccardo.rubei@graduate.univaq.it

[1] DISIM - University of L'Aquila, L'Aquila, Italy