Mining Software Repositories to Support OSS Developers: A Recommender Systems Approach

Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio

Department of Information Engineering, Computer Science and Mathematics
Università degli Studi dell'Aquila
Via Vetoio 2 – 67100 L'Aquila, Italy
{phuong.nguyen, juri.dirocco, davide.diruscio}@univaq.it

Abstract. To facilitate the development activities, software developers frequently look up external sources for related information. Consulting data available at open source software (OSS) repositories can be considered as their daily routine. Nonetheless, the heterogeneity of resources and their corresponding dependencies are the main obstacles to the effective mining and exploitation of the data. Given the context, the manual search for every single resource to find the most suitable ones is a daunting and inefficient task. Thus, equipping developers with techniques and tools to accelerate the search process as well as to improve the search results will help them enhance the work efficiency. Within the scope of the EU funded CROSSMINER project, advanced techniques and tools are being conceived for providing open software developers with innovative features aiming at obtaining improvements in terms of development effort, cost savings, developer productivity, etc. To this end, cutting-edge technologies are applied, such as information retrieval and recommender systems to solve the problem of mining the rich metadata available at OSS repositories to support software developers. In this paper, we present the main research problems as well the proposed approach together with some preliminary results.

1 Introduction

During the development phase, software programmers need to tackle various issues, such as mastering different programming languages, reusing source code, or choosing suitable external third-party libraries. By exploiting existing well-defined artifacts from open source software (OSS) repositories, such as code snippets and API usage patterns, one can avoid coding from scratch. Nevertheless, as illustrated in Fig. 1, the available information is huge and heterogeneous as it comes from different sources, e.g. source code, Q&A systems, or API documentations. Given the circumstances, even experienced and skilled developers might face difficulties in searching for suitable resources.

Over the last years, considerable effort has been devoted to data mining and knowledge inference techniques to provide automated assistance to developers in navigating large information spaces and giving recommendations [24], for instance, API documentation recommendation [27], mining Q&A systems [21], API usages recommendation [17,30], and third-party library recommendation [26] just to mention a few.

IIR 2018, May 28-30, 2018, Rome, Italy. Copyright held by the author(s).

CROSSMINER[1] [1] is a research project funded by the EU Horizon 2020 Research and Innovation Programme and aims at extending the EU OSS-METER FP7 project [9] by supporting the development of complex software systems by facilitating the comparison and adoption of already existing open source software components. To this end, CROSSMINER is conceiving techniques for knowledge extraction from large open source software repositories. Being equipped with cutting-edge technologies, developers can make use of existing similar modules, instead of reimplementing them. Thus, many traditional development tasks become semi- or fully- automated by means of meaningful recommendations. Thus, the job of developers is expected to be more effective and efficient.

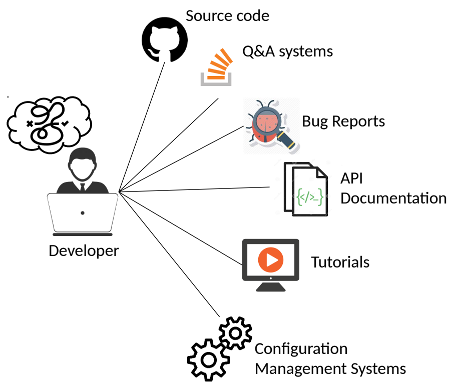

Fig. 1. Developers are overwhelmed by huge and miscellaneous sources of supporting materials

The work in CROSSMINER differs from other existing studies in the sense that it brings in a completely new paradigm for the representation of OSS artifacts so as to pave the way for various computations. Further than extending state-of-the-art approaches in the field of automated analysis and measurement of open source software, we develop advanced techniques to investigate relationships among different OSS projects and properly organize them in a dedicated knowledge base. The knowledge base fosters the deployment of recommender systems to present users with interesting items previously unknown to them.

In this paper, we describe our ongoing work with the focus on the mining of cross relationships among OSS projects to provide developers with helpful recommendations. To this end, the paper is organized as follows: an overview of the CROSSMINER project and of the envisioned recommendations is given in Section 2. Section 3 presents the main constituting elements of the proposed approach to realize such recommendations. Section 4 recalls popular metrics used for evaluating recommendation outcomes. Section 5 introduces some preliminary results and finally, Sect. 6 concludes the paper.

2 Overview of the envisioned CROSSMINER recommendations

CROSSMINER aims at supporting software developers by means of an advanced Eclipse-based IDE providing intelligent recommendations that go far beyond the current "code completion-oriented" practice. To this end, data retrieved from different sources has to be collected and processed so to properly feed the recommendation component. In particular, as shown in Fig. 2, four high-level modules compose the CROSSMINER platform.

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

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

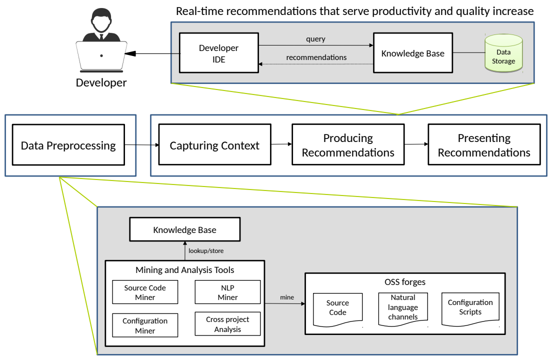

Fig. 2. A high-level view of CROSSMINER

To provide developers with useful support, we concentrate on working with the use cases depicted in Fig. 3. The knowledge base takes the developer context as input and returns recommendations as discussed below:

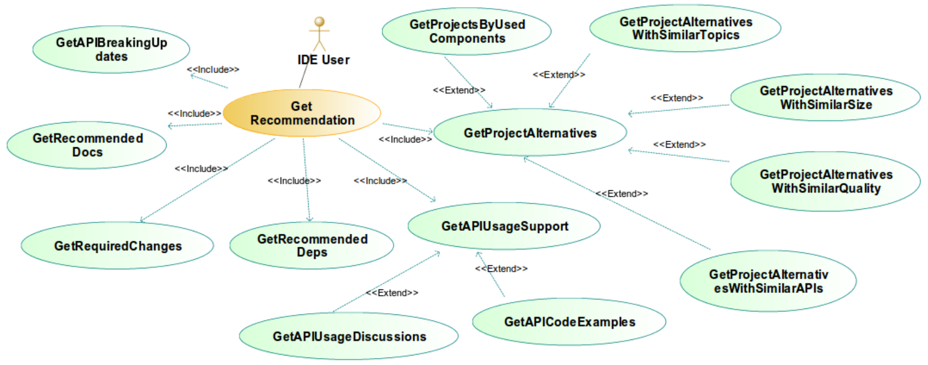

Fig. 3. Types of recommendations

- **GetProjectAlternatives**: we implement novel clustering and similarity mechanisms being able to suggest OSS projects that can be alternatively used instead of OSS components, which have been previously selected and integrated in the software being developed. Based on designated similarity functions, we are able to detect projects that are similar because of: Provided APIs (*GetProjectAlternativesWithSimilarAPIs*) [15]; Size (*GetProjectAlternativesWithSimilarSize*); Application domain (*GetProjectAlternativesWithSimilarTopics*); and Comparable quality (*GetProjectAlternativesWithSimilarQuality*);
- **GetProjectsByUsedComponents:** depending on the used components, the knowledge base is able to identify and suggest further components that, according to what

other developers have done in the past, should be also included in the system being implemented. Two prominent examples are recommendation of third-party libraries [26] and code snippets [16];

- **GetAPIUsageSupport:** The knowledge base provides developers with recommendations on how to use a given API and to manage the migration of the system in case of deprecated methods. This use case consists of:
  - *GetAPIUsageDiscussions:* given an API the developer has already included, it is possible to retrieve messages from communication channels (like forums, bug reports, and Stack Overflow posts) that are useful for understanding how to properly use it [21];
  - *GetAPIUsagePatterns:* in case of deprecated API methods, the knowledge base recommends code examples that can be considered as a reference for migrating the system and to make it work with the new version of the used API [20,30].
- **GetRecommendedDeps:** starting from a given configuration and by considering similar projects developed by other developers, the knowledge base recommends other additional third-party libraries that should be further included [26];
- **GetRecommendedDocs:** by considering the documentation examined by other developers that used similar APIs and frameworks, the knowledge base suggests additional sources of information, e.g. technical documents, tutorials, etc., that are useful for solving the development problem at hand [27];
- **GetAPIBreakingUpdates:** the knowledge base implements the notion of API evolution with the aim of identifying backward compatibility problems affecting source code that uses evolving APIs;
- **GetRequiredChanges:** given a changed API and a project using it, the knowledge base provides an overview of the impact that the changes have on the depending project. Communication channel items discussing about such API changes will be also shown.

It is worth noting that the recommendations previously summarized have been identified during the first 6 months of the CROSSMINER project to satisfy the requirements of the industrial partners that work in the domains of IoT, multi-sector IT services, API co-evolution, software analytics, software quality assurance, and OSS forges [1].

3 Proposed recommendation approach

Our approach is built based on the notation of *recommender systems* [24]. In the context of mining software repositories, those are systems that can provide recommendations to developers with regards to their development context. For recommender systems in general, the ability to measure the similarity between items plays an important role in obtaining relevant recommendations [10]. Intuitively, for software mining recommender systems, the measurement of similarities between artifacts, e.g. projects, dependencies, code snippets, or even developers shall also be a critical factor. Nevertheless, the computation of similarities between software systems/open source projects in particular has been identified as a thorny issue [15]. Furthermore, considering the miscellaneousness of artifacts in OSS repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail.

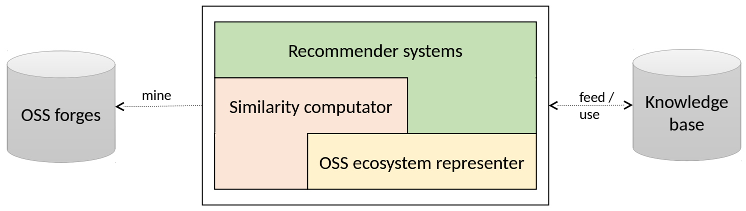

Fig. 4. The main components underpinning the CROSSMINER recommendation system

Fig. 4 depicts a layered architecture consisting of the core elements, which underpin the realization of the recommendations summarized in the previous section. An overview of such elements, which mine OSS forgers and manage the content of a knowledge base, is given in the next sub-sections.

3.1 OSS ecosystem representer

To enable both the representation of different OSS projects and the calculation of their similarity, a *graph-based* model has been conceived [18]. We consider the community of developers together with OSS projects and other artifacts as an *ecosystem*. Graphs are then used for representing different types of relationships in the OSS ecosystem.

The adoption of the graph-based representation allows for the transformation of the relationships among various artifacts in the OSS ecosystem into a mathematically computable format. The following relationships are used to construct graphs representing the OSS ecosystem and eventually to calculate similarity using graph algorithms.

- *includes* $\subseteq$ *Dependency*$\times$*Project*: according to [15,26], the similarity between two projects relies on the dependencies they have in common because they aim at implementing similar functionalities.
- *develops* $\subseteq$ *Developer* $\times$ *Project*: we assume that there exists a certain level of similarity between two projects if they are built by same developers [5];
- *stars* $\subseteq$ *User* $\times$ *Project*: this relationship models the star event to represent GitHub projects that a given user has starred.
- *develops* $\subseteq$ *User* $\times$ *Project*: this relationship is used to represent the projects that a given user contributes in terms of source code development;
- *implements* $\subseteq$ *File* $\times$ *File*: it depicts a specific relation that can occur between the source code given in two different files, e.g. a class specified in one file implementing an interface given in another file;
- *hasSourceCode* $\subseteq$ *Project* $\times$ *File*: it represents the source files in an OSS project.

Fig. 5 depicts an excerpt of a graph representing an explanatory example with two OSS projects `project#1` and `project#2`. The former contains `HttpSocket.java` and the latter contains `FtpSocket.java`. Both files implement `interface#1` marked by `implements`.

3.2 Similarity computator

Nodes, links, and the mutual relationships among them allow for the computation of similarity [4]. To the best of our knowledge, there are several techniques for computing

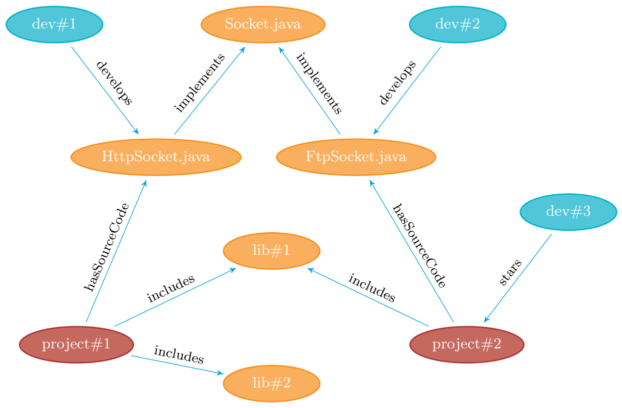

Fig. 5. Representation of OSS projects and their corresponding developers and artifacts

similarity in graph [8]. SimRank is among the most notable algorithms for computing graph similarity [12]. Given two nodes, SimRank computes the similarity by considering their neighbours: the more shared nodes point to them, the more similar the two nodes are. Besides SimRank, there are many other algorithms for calculating similarities in graph that cannot be recalled in this paper due to space limitation [11,14].

In Fig. 5, we can compute the similarity between `project#1` and `project#2` using related semantic paths, e.g. the `hasSourceCode` and `implements` two-hop path, or the one-hop path `includes`. This assumption relies on the fact that the projects are implementing common functionalities by using common libraries [15,26]. The graph also allows to compute the similarity between two developers, e.g. `dev#1` and `dev#2` as they are indirectly connected by the `develops` and `implements`. In summary, by transforming various OSS artifacts into a graph, using different similar algorithms, we are able to perform similarity calculation for different artifacts which then serves as a base for other computations.

3.3 Recommender system

We derive recommendation techniques from the mechanisms implemented for e-commerce systems [13]. There, given a customer, products that have been purchased by similar customers are recommended to her [25]. Similarly, given a software project we recommend artifacts that exist in projects that are similar to it. A *content-based recommender system* works by recommending to a developer various artifacts, e.g. code snippets, API method invocations, external libraries in projects that are similar to the ones being developed. Whereas, a *collaborative-filtering recommender system* gives to a developer recommendations that are based on the artifacts used by developers with similar behaviors [25].

By referring to the example shown in Fig. 5, we see that `project#1` is similar to `project#2` in terms of the functionalities [15,26]: they contain classes `HttpSocket.java` and `FtpSocket.java` which implement the same interface `Socket.java`. Furthermore, both projects share the third-party library `lib#1`. Thus, it is sensible to recommend `lib#2` to `project#2` since `lib#2` is being used by `project#1`.

With this recommendation, the developers of `project#2` are able to save the time spent on manual searching for `lib#2`. Analogously, since the two projects are similar, it is also worthwhile to suggest developer `dev#3` starring `project#1` since she already starred `project#2`[2]. In practice, the recommendation of OSS artifacts is much well-defined with the incorporation of several similar projects instead of only one.

By following this design, we implemented the first prototype of a recommender system that is able to recommend similar projects and third-party libraries. Before going to present some initial results in Section 5, we recall some popular metrics for evaluating recommendation outcomes in Section 4.

4 Evaluation metrics

Given a query, the outcome of the recommendation process is a ranked list of items that are considered to be relevant for the query. For instance, a system that recommends third-party libraries for a given project returns a list in descending order of real similarity scores corresponding to libraries [26]. To validate the performance of a recommender system, we perform *cross validation* using a *training* and a *testing* dataset [6]. Training data is used to build the model whereas testing data is used to validate the outcome. Considering a project that needs library recommendation, the graph model is used to compute similarities and then to find most k similar projects. The outcome of the recommendation is a ranked list of libraries. Normally, a developer pays attention only to the *top-N* items. We use k and N as parameter for the evaluation later on.

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

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

Using these notations, the metrics utilized to measure the recommendation outcomes are explained in the following. Among others, we consider *success rate* [26], *accuracy* [15], *sales diversity*, and *novelty* [19] the most suitable metrics for evaluating a recommender system in mining OSS repositories [24].

4.1 Success rate

Given a set of *P* testing projects, this metric measures the rate at which a recommender system can return at least a match among *top-N* recommended items for every project $p \in P$ [26]. The metric is formally defined as follows:

[2] Starring is used by GitHub developers as a means to bookmark an OSS repository and to thank its contributors. The GitHub star has nothing to do with ratings as by TripAdvisor or YouTube

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

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

4.2 Accuracy

Given a list of *top-N* items, *precision@N*, *recall@N*, and *normalized discounted cumulative gain* are utilized to measure the *accuracy* of the recommendation results.
Precision@N is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

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

Recall@N is the ratio of the ground-truth items appearing in the *N* items [7,8,19]:

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

Normalized Discounted Cumulative Gain Precision and recall reflect well the accuracy, however they neglect ranking sensitivity [3]. nDCG is an effective way to measure if a system can present highly relevant items on the top of the list:

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

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

4.3 Sales Diversity

In e-commerce systems, *sales diversity* is the ability to improve the coverage as also the distribution of products across customers [19,29]. In the context of mining software repositories, sales diversity means the ability of the system to suggest to projects as much items, e.g. libraries, code snippets, as possible, as well as to disperse the concentration among all, instead of focusing only on a specific set of items [24].
Catalog coverage measures the percentage of items recommended to projects:

$$coverage@N = \frac{\left|\cup_{p \in P} \cup_{r=1}^{N} REC_r(p)\right|}{|I|} \quad (5)$$

Entropy evaluates if the recommendations are concentrated on only a small set or spread across a wide range of items [22]:

$$entropy = -\sum_{i \in I} \left(\frac{\#rec(i)}{total}\right) ln \left(\frac{\#rec(i)}{total}\right) \quad (6)$$

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

4.4 Novelty

Novelty measures if a system is able to expose items to projects. *Expected popularity complement* (EPC) is utilized to measure *novelty* and is defined as follows [28,29]:

$$EPC@N = \frac{\sum_{p \in P} \sum_{r=1}^{N} \frac{rel(p,r) * [1 - pop(REC_r(p))]}{log_2(r+1)}}{\sum_{p \in P} \sum_{r=1}^{N} \frac{rel(p,r)}{log_2(r+1)}} \tag{7}$$

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 and the number of projects that receive the most ever recommended items as recommendation. Equation 7 implies that the more unpopular items a system recommends, the higher the EPC value it obtains and vice versa.

5 Preliminary results

For explanatory purpose, we introduce an example of how the CROSSMINER recommender system can be applied to assist OSS developers in a specific context. During the software development phase, among other tasks, programmers regularly search for and reuse third-party libraries [2]. A third-party library is an interface to a reusable source code and can be embedded in external software projects independently from environment code [23]. To help developers quickly locate suitable dependencies, in the context of the CROSSMINER project we implemented CrossRec, a framework that exploits **C**ross Projects **R**elationships in **O**pen **S**ource **S**oftware Repositories to build a **Rec**ommender System. CrossRec employs a collaborative-filtering technique based on the similar model applied in e-commerce systems [13]. Instead of recommending products to customers, we recommend third-party libraries to projects using exactly the same mechanism: *"given a project, libraries come from similar projects."* To be precise, in this section we address the use case *GetRecommendedDeps* outlined in Section 2.

To evaluate CrossRec, we considered a well-established tool as baseline. In particular, to the best of our knowledge, LibRec [26] is one the most advanced techniques for library recommendation. Based on the set of third-party libraries that a project has already included, LibRec searches for relevant libraries with a high *success rate* (see Section 4.1). Using the available implementation[3], we conducted an evaluation on LibRec and CrossRec with the same dataset consisting of 5.200 GitHub Java projects to see how well they can search for suitable third-party libraries.

Since *success rate* was used as the only evaluation metric for LibRec [26], we exploited it as a means to compare directly the performance of both systems. Table 1 shows *success rate@5* for k={5, 10, 15, 20, 25}. As can be seen, the success rates obtained by CrossRec are always superior to those of LibRec. The maximum *success rate@5* of LibRec is 0.8780, whereas CrossRec obtains success rates that are always greater than 0.9073 for all configurations, with 0.9286 being the maximum value. For

[3] We would like to thank Ferdian Thung and David Lo at the School of Information Systems, Singapore Management University for providing us with the original LibRec implementation

Table 1. Success rate for N={5,10}, k={5,10,15,20,25}

	N=5		N=10	
k	LibRec	CrossRec	LibRec	CrossRec
5	0.8576	**0.9073**	0.9143	**0.9421**
10	0.8757	**0.9230**	0.9332	**0.9526**
15	0.8767	**0.9269**	0.9313	**0.9550**
20	0.8780	**0.9286**	0.9334	**0.9557**
25	0.8769	**0.9284**	0.9334	**0.9532**

Table 2. Success rate for N={1,3,5,7,10}, k={10,20}

	k=10		k=20	
N	LibRec	CrossRec	LibRec	CrossRec
1	0.6248	**0.7482**	0.6565	**0.7650**
3	0.8192	**0.8892**	0.8228	**0.8951**
5	0.8757	**0.9230**	0.8780	**0.9286**
7	0.9078	**0.9386**	0.9055	**0.9442**
10	0.9332	**0.9526**	0.9334	**0.9557**

$N = 10$ both LibRec and CrossRec get a considerable improvement in their performance *success rate@10* compared to the case with $N = 5$. It is evident that in all test configurations, CrossRec gains a better performance than that of LibRec.

Next we investigate success rate with respect to the length of the recommendation list N, i.e. $N = \{1, 3, 5, 7, 10\}$. In the first experiment, k is fixed to 10 and the outcomes are depicted in Table 2. For $N = 1$, LibRec gets a success rate of 0.6248 which is much lower than 0.7482, the corresponding value by CrossRec. The value of 0.7482 shows that CrossRec is able to provide relevant recommendations to the developer at an encouraging match rate, even when she expects only an extremely brief list. Starting from $N = 5$, CrossRec gains a quick increase in its performance as the success rates are always greater than 0.9286.

In the second experiment, k is changed to 20 and both systems have a slight increase in their success rate, however the change is marginal. By conducting more experiments with an increasing k, we noticed that incorporating more similar projects for recommendation does not improve success rate (the outcomes of these experiments are omitted from the paper due to space limitation).

In summary, by considering the results depicted in Table 1 and Table 2, we come to the conclusion that CrossRec obtains a better success rate in comparison to LibRec in all cases. This confirms our hypothesis that the collaborative-filtering technique is a profitable deployment for recommendation of third-party libraries, and consequently it deserves to be further investigated in the context of the CROSSMINER project.

6 Conclusions

We presented our proposed framework to assist software developers in mining OSS repositories. We exploit the graph model to represent the semantic relationships of the OSS ecosystem. Afterwards, a knowledge base with metadata curated from OSS repositories has been populated to serve for various mining techniques. We built our first prototype of a recommender system using the collaborative-filtering technique. A preliminary evaluation on a dataset of 5.200 GitHub Java projects shows that our system for recommending third-party libraries outperforms a well-known baseline. For future work, we are going to address all the recommendations mentioned in Section 2 following the model proposed in this paper.

Acknowledgments

The research described in this paper has been carried out as part of the CROSSMINER Project, EU Horizon 2020 Research and Innovation Programme, grant agreement No. 732223

References

1. A. Bagnato et. al. Developer-centric knowledge mining from large open-source software repositories (crossminer). In *Software Technologies: Applications and Foundations*, pages 375–384. Springer International Publishing, 2018.
2. V. Bauer, L. Heinemann, and F. Deissenboeck. A structured approach to assess third-party library usage. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 483–492. IEEE Computer Society, 2012.
3. A. Bellogín, I. Cantador, and P. Castells. A comparative study of heterogeneous item recommendations in social systems. *Inf. Sci.*, 221:142–169, Feb. 2013.
4. V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. V. Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Rev.*, 46(4):647–666, Apr. 2004.
5. N. Chen, S. C. Hoi, S. Li, and X. Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 305–314. ACM, 2015.
6. P. Cremonesi, R. Turrin, E. Lentini, and M. Matteucci. An evaluation methodology for collaborative recommender systems. In *Proceedings of the 2008 International Conference on Automated Solutions for Cross Media Content and Multi-channel Distribution*, AXMEDIS '08, pages 224–231. IEEE Computer Society, 2008.
7. J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 233–240. ACM, 2006.
8. T. Di Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker. Linked open data to support content-based recommender systems. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 1–8. ACM, 2012.
9. D. Di Ruscio, D. S. Kolovos, I. Korkontzelos, N. Matragkas, and J. J. Vinju. Ossmeter: A software measurement platform for automatically analysing open source software projects. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 970–973. ACM, 2015.
10. G. Guo, J. Zhang, and N. Yorke-Smith. A novel bayesian similarity measure for recommender systems. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 2619–2625. AAAI Press, 2013.
11. T. H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 517–526. ACM, 2002.
12. G. Jeh and J. Widom. Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 538–543. ACM, 2002.
13. G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, Jan. 2003.
14. W. Lu, J. Janssen, E. Milios, N. Japkowicz, and Y. Zhang. Node similarity in the citation graph. *Knowledge and Information Systems*, 11(1):105–129, Jan 2007.

15. C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374. IEEE Press, 2012.
16. C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25. ACM, 2010.
17. L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890. IEEE Press, 2015.
18. P. T. Nguyen, J. D. Rocco, R. Rubei, and D. D. Ruscio. Crosssim: exploiting mutual relationships to detect similar oss projects. In *Procs. of 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) - to appear*, 2018.
19. P. T. Nguyen, P. Tomeo, T. Di Noia, and E. Di Sciascio. Content-based recommendations via dbpedia and freebase: A case study in the music domain. In *Proceedings of the 14th International Conference on The Semantic Web - ISWC 2015 - Volume 9366*, pages 605–621. Springer-Verlag New York, Inc., 2015.
20. H. Niu, I. Keivanloo, and Y. Zou. Api usage pattern recommendation for software development. *J. Syst. Softw.*, 129(C):127–139, July 2017.
21. L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111. ACM, 2014.
22. A. Ragone, P. Tomeo, C. Magarelli, T. Di Noia, M. Palmonari, A. Maurino, and E. Di Sciascio. Schema-summarization in linked-data-based feature selection for recommender systems. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 330–335. ACM, 2017.
23. M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *IEEE Trans. Softw. Eng.*, 39(5):613–637, May 2013.
24. M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-642-45135-5.
25. B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 285–295. ACM, 2001.
26. F. Thung, D. Lo, and J. Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191, Oct 2013.
27. F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic recommendation of api methods from feature requests. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 290–300. IEEE Press.
28. S. Vargas and P. Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, RecSys '11, pages 109–116. ACM, 2011.
29. S. Vargas and P. Castells. Improving sales diversity by recommending users to items. In *Eighth ACM Conference on Recommender Systems, RecSys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014*, pages 145–152, 2014.
30. H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 318–343. Springer Berlin Heidelberg, 2009.