Knowledge-aware Recommender System for Software Development

Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio
Università degli studi dell'Aquila
Via Vetoio 2 – 67100 L'Aquila, Italy
{phuong.nguyen,juri.dirocco,davide.diruscio}@univaq.it

ABSTRACT

Open source software (OSS) forges contain rich data sources that are useful for the development process. We promote techniques and tools for providing open source developers with innovative features aiming at obtaining improvements in terms of development effort, cost savings, developer productivity. Our work is a coherent paradigm that facilitates multiple recommendations to assist software developers in different phases of the development process. In the scope of this paper, we introduce a graph-based representation to encode in a homogeneous manner different aspects of the OSS ecosystem. Furthermore, we develop a knowledge-aware recommender system for providing developers with suitable API function calls. An initial evaluation on real datasets shows that the system is able to produce relevant API calls.

1 INTRODUCTION

Open source software (OSS) is computer software available in source code being provided under a license that allows users to study, change, and improve the code free of charge. There are several high-quality and mature projects which deliver stable and well-documented products. Most OSS forges (e.g., GitHub, BitBucket, and SourceForge) typically sustain vibrant expert and user communities which in turn provide decent levels of support both with respect to answering user questions as well as to repairing reported software bugs. In this sense, developing a new software system by making use of existing open source components reduces development effort and thus being beneficial to the whole software life cycle.

However, properly exploiting such foundations poses several challenges including the miscellaneousness of resources and the huge information space that impedes effective mining. Developers need to deal with data coming from different sources, such as Source code, Q&A systems, bug reports, API documentation, tutorials, just to mention a few [12]. The mining of the different data sources necessitates at least the following activities: searching for candidate components that can be suitably reused, evaluating and comparing them, and adapting the selected components to fit the specific requirements of the new system being developed. Given the circumstances, it is impossible to exploit the underlying knowledge without suitable machinery.

Knowledge-aware and Conversational Recommender Systems (KaRS) Workshop 2018 (co-located with RecSys 2018), October 7, 2018, Vancouver, Canada.

Within the EU H2020 CROSSMINER[1] project, we aim at building a thorough framework for supporting OSS developers. In particular, we design and implement tools that automatically curate data from large OSS forges in order to feed dedicated recommendation engines. The augmented tools are provided in the form of an advanced Eclipse-based IDE, which instantly monitors developers' activities and triggers alerts as well as produces intelligent recommendations. To this end, our proposed framework consists of the following main functionalities [12]:

- *recommending sets of similar projects* with regards to various requirements, such as external dependencies, application domain, or API usage by employing various similarity algorithms. The similar projects help developers learn how to implement the given project at an early stage;
- *suggesting artifacts* that have been incorporated in similar projects, e.g. a list of external libraries [12] or code snippets [10]. These artifacts can then be directly embedded into the current project;
- *prompting code snippets* that demonstrate how an API is utilized in practice. The recommended snippets help developers gain a deeper insight into the usage of the API;
- *suggesting external information sources*, e.g. technical documents, tutorials, discussions, etc., related to the code being developed. For example, given an API, it is necessary to find external posts to understand how other developers use the API [14];
- External libraries evolve over the course of time, which impose change on the depending projects. Thus, it is necessary *to notify developers and suggest possible amendments* to preserve program compatibility.

In order to mine the external and explicit knowledge sources to feed the recommendation engines, the key point is to find a suitable model to represent the intrinsic relationships among several OSS artifacts. Furthermore, as input data comes from various sources, a conventional recommender system that deals with a fixed type of data cannot be applicable to the context of mining OSS repositories. We come across with the graph model to encode the semantic among artifacts and we employ a knowledge-aware recommender system to exploit cutting-edge recommendation technologies for mining software repositories. Since a knowledge-aware recommender system incorporates the underlying knowledge available at OSS repositories, it is expected that it can produce recommendations that fit well to developers' need.

This paper presents an approach to support software development by means of a knowledge-aware recommender system. We

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

model the OSS ecosystem using a Knowledge Graph to enable the computation of similarities and giving recommendations [12]. As a proof of concept, we present a concrete use case by exploiting the Knowledge Graph to provide an important functionality: API function calls recommendation. To validate the proposed approach, we perform an evaluation on two datasets of Java projects curated from the Maven repository. The remainder of the paper is organized as follows. Section 2 presents the Knowledge Graph and recommendation techniques. The use case for recommending API function calls is introduced in Section 3. We recall some related work in Section 4. The paper discusses future work and concludes in Section 5.

2 THE PROPOSED RECOMMENDER SYSTEM

This section introduces the recommender system we are defining in the context of the CROSSMINER project for supporting developers that have to develop new systems by reusing existing open source components. Section 2.1 presents a graph-based representation of different artifacts that are involved when developing open source software. Section 2.2 presents the conceived recommendation techniques that rely on the proposed representation model.

2.1 A Knowledge Graph for the OSS ecosystem

We propose a representation model to capture the intrinsic implications among various artifacts of the OSS *ecosystem* [13]. By means of a Knowledge Graph [2], we incorporate both human (such as developers, users) and non-human factors (such as source code, and libraries) into a homogeneous representation. In such graph, a node represents either a person, or an artifact, such as a library, an API function call, and a directed edge represents the relationship between them. Table 1 explains all the relationships that we define in our current implementation [13]. The vocabularies can also be augmented upon additional features of input data, even though our definition seems to cover the most prevailing relationships.

For explanatory purpose, we introduce a Knowledge Graph in Figure 1. The graph encodes several relationships and interactions among various OSS artifacts. For instance, with `develops` and `commits`, we are able to represent the fact that two developers $\texttt{dev}_1$ and $\texttt{dev}_2$ take part in the development of source code belonging to two projects $\texttt{project}_1$ and $\texttt{project}_2$. Similarly, the edge `extends` dictates that two classes `iClass1.java` and `iClass2.java` are an instance of `AbstractClass.java`, they both extend a same abstract class and thus sharing common functionalities. The graph structure facilitates similarity computation, which is a building block for several recommendation algorithms [4, 6]. For example, the similarity between `iClass1.java` and `iClass2.java` can be inferred by considering two edges namely `extends` and `invokes`. The two nodes are indirectly connected through other nodes $\texttt{API}_1$ and `AbstractClass.java` and their similarity can be computed by means of several graph algorithms [12, 13].

The Knowledge Graph also sustains other types of mining to support OSS developers. Take as an example, in Figure 1 the StackOverflow[2] post $\texttt{Post}_1$ mentions code snippets containing two function calls $\texttt{API}_1$ and $\texttt{API}_2$ from external libraries via the edge `contains`. In practice, this a communication between users discussing the usage of $\texttt{API}_1$ and $\texttt{API}_2$. In this sense, it might be worthwhile to recommend $\texttt{Post}_1$ to the developer of `iClass2.java`, i.e. $\texttt{dev}_2$ as this class invokes both API function calls. Such recommendation is helpful for developers when they work on the related function calls, since it provides a deeper insight into the corresponding APIs. Since the related knowledge is already encoded in the graph, we are able to compute the similarity between $\texttt{Post}_1$ and `iClass2.java`. Eventually, the recommendation engine can present to the developer a list of StackOverflow posts that are relevant to the source code being developed.

In this sense, we see that the adoption of a Knowledge Graph paves the way for the deployment of recommender systems which can address the underlying knowledge contained in OSS forges. It is possible to transform the relationships among humans and non-human artifacts into a mathematically computable format, which then facilitates various types of calculations. The definition of a proper representation model is a preparatory phase to other developments, including several types of recommendations. It is worth noting that the creation of a knowledge graph for OSS forges is not trivial as we need to properly analyze both source code and metadata to mine the encoded relationships and eventually represent them in a homogeneous scheme. In the following section, we discuss the possibility of exploiting the Knowledge Graph for supporting OSS developers.

2.2 Recommendation techniques

We investigate recommendation techniques that are applicable to the context of mining software repositories. A *collaborative-filtering recommender system* exploits a two-dimensional matrix to represent the relationships between users and items and computes missing ratings [16]. A user's preference towards a prospective item is predicted by means of preferences from similar users [1]. The collaborative-filtering technique is applicable to mining OSS repositories, as long as suitable interpretation can be conceived. For instance, if we consider *projects* as *customers*, and *libraries* as *products*, then it is possible to exploit the collaborative-filtering technique to recommend third-party libraries. Following this scheme, we successfully developed a system for recommending third-party libraries which obtains a superior performance compared to a well-established baseline [12].

The conventional collaborative-filtering technique is not applicable to the situation when additional features are present. Given that the preference of a user changes depending on the context where the decision is made, the rating matrix is extended to three dimensions, i.e user, item, and context. Incorporating context into the computation process brings in a new level of recommender systems, so called *context-aware recommender systems* [1]. Considering a customer who needs recommendations on what additional products should be put into the shopping cart, the intuition is to collaboratively deduce the presence of prospective items from *those that have been purchased by similar customers in comparable contexts* [3]. In mining OSS repositories, a context-aware collaborative-filtering technique can be used if we find suitable interpretation of contexts.

A possible interpretation is as follows: if we use the following mappings *projects–contexts*, *developers–customers*, *StackOverflow posts–products*, we can apply the context-aware collaborative-filtering technique to suggest StackOverflow posts that may be

[2]https://stackoverflow.com/

Relationship	**Space**	**Description**
commits	*Developer* × *Project*	This relationship represents the projects or libraries that a user contributes to the development
contains	*File* × *API*	A source file or a communication post with code snippets containing an API function from a third-party library
develops	*Developer* × *Project*	A developer contributes to source code development for a given project
extends	*Class* × *Class*	A class inherits an abstract class. Two classes extend a same abtract class have a bond since they share a certain number of common functionalities
hasSourceCode	*Project* × *File*	An OSS project contains a source file
includes	*Library* × *Project*	A project includes a third-party library to make use of its functionalities
invokes	*Class* × *API*	This is the case when a class calls an API function from a third-party library. API calls can be extracted from source files using suitable code parsers
stars	*User* × *Project*	It represents projects that a given user has starred. This relationship is only applicable to GitHub

Table 1: Relationships in the Knowledge Graph

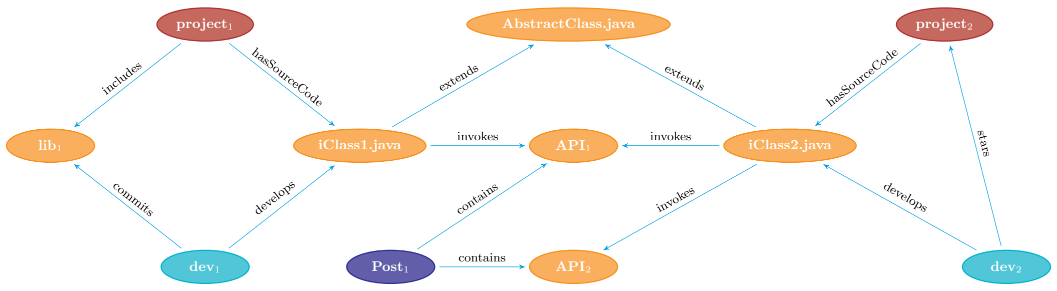


Figure 1: Knowledge Graph for the OSS ecosystem

helpful for a developer. The relationship among *projects*, *developers*, and *posts* is represented in a 3-D matrix, where each slice is a project, each row is a developer and each column is a post. A cell is set to 1 if the developer consults the post when she develops the project, otherwise it is set to 0. The context-aware collaborative-filtering technique is then exploited to perform computation on the 3-D matrix to find missing items. Eventually, the recommendation engine returns a list of posts that the developer may find useful.

By exploiting the proposed approach, we succeeded in developing two recommender engines for supporting OSS developers. In [13], CrossSim has been implemented to present to developers a list of similar projects given a project being developed. Similarly, CrossRec is a system for recommending third-party libraries by employing a dedicated OSS graph [12]. In the following section, we present another application of a Knowledge Graph to suggest API function calls to be embedded to source code, taking into consideration the current development context.

3 A USE CASE: RECOMMENDING API FUNCTION CALLS

Embedding well-established components developed by third parties into source code is beneficial to the development process. Rather than working from scratch, developers normally look for external libraries that implement the desired functionalities and integrate them into their existing projects [12]. 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, developers need to consult various sources, e.g. API documentation to see how a specific API instance is utilized in practice [14]. Generally, from these external sources, there are texts providing generic syntax or simple usage of the API, which may be less relevant to the current development context as a whole. In this sense, concrete examples of source code snippets that indicate how specific API function calls are deployed in actual usage, would come in handy [11].

Clustering has been considered as the *de facto* mechanism for finding similar source code snippets, aiming to remove redundant items [8, 18]. Nevertheless, a substantial amount of redundancy is still witnessed by approaches that rely on clustering [5]. In this section, we introduce a solution to recommend API function calls by exploiting the Knowledge Graph presented in Section 2.1. We aim at providing developers with highly relevant API function calls by carefully taking into account their development context. By means of the Knowledge Graph, we are able to compute similarity and eventually to feed the recommendation engine. First, we introduce the following definitions:

- Method invocation (or invocation): a function call from an external API;
- Method declaration (or declaration): a single source code unit, i.e. a function/procedure, that contains various invocations from different APIs;
- Software project (or project): a complete, standalone source code unit that consists of a set of declarations to perform a particular job.

Figure 2 presents the source code of a real function written in Java. According to the above notations, there are a declaration named `clone()` and some invocations, such as `entrySet()` and `getValue()`. These artifacts can be extracted from source using a suitable parser and they are used as input for the recommendation process.

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

Figure 2: A real Java code snippet

3.1 Predicting API function calls

To predict additional API function calls to be integrated, we derive a collaborative-filtering technique from the engine designed for product recommendation [3]. Instead of recommending products to customers with regards to context, we recommend invocations to declarations, taking into consideration the given project. In other words, by using the following mappings: *projects–contexts*, *declarations–customers*, *invocations–products*, we are able to transform the recommendation model applied for e-commerce systems into mining API function calls. A *tensor* τ is utilized to capture the intrinsic relationships among projects, declarations, and invocations and eventually to produce recommendation.

In tensor τ, each slice corresponds to a project, each row is a declaration and each column is an invocation. τ is in the form of $\tau \in \Gamma^{n \times m \times k}$, where n, m, and k are the number of projects, of declarations, and of invocations, respectively. Given a slice, a cell is set to 1 if the declaration in corresponding row consists the invocation in corresponding column, otherwise it is set to 0. Given a project that needs recommendation on what items should be integrated, the cells for missing invocations are set to -1.

Figure 3 depicts the tensor representing a set of five OSS projects $P = (p_1, p_2, p_3, p_4, p_5)$, with four declarations $D = (d_1, d_2, d_3, d_4)$ and seven invocations $I = (i_1, i_2, i_3, i_4, i_5, i_6, i_7)$ in total. In the tensor, slices correspond to projects, rows are declarations and columns correspond to invocations. For the sake of clarity, we only depict a part of projects p_1 and p_3. In p_3, there is no invocation i_4 in declaration d_2, so the corresponding cell is set to 0. Meanwhile, by declaration d_3, invocation i_3 is found, therefore the corresponding cell is set to 1. Respectively p_1 and d_1 are the *active project* and *active declaration*, i.e. the ones being developed. At the time of consideration, it is not clear if d_1 of p_1 should include i_2 and i_3, thus the corresponding cells are -1.

There are a number of techniques available for computing missing ratings, i.e. cells filled with -1, such as by using *tensor factorization* as proposed in [7]. There, the original tensor is decomposed into three sub-matrices and a central tensor, and the computation of missing ratings is done by minimizing a loss function. In the scope of this paper, we exploit the mechanism presented in [3] to predict the ratings since it allows for the exploitation of similarity scores computed by means of the Knowledge Graph presented in Section 1. A collaborative-filtering technique is then applied to perform computation on the tensor to find missing items. We refrain from recalling the technique due to space limitation and interested readers are referred to [3] for greater detail. Eventually, the recommendation engine returns a list of invocations that can be embedded into the active declaration. In the following section, we present an evaluation on Maven datasets to validate the performance of our proposed approach.

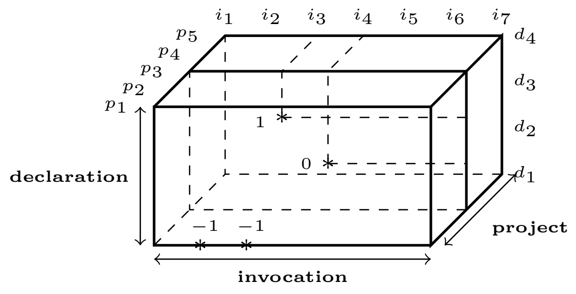


Figure 3: A tensor for representing the project-declaration-invocation relationship

3.2 Evaluation Settings and Metrics

A set of 3,600 jar files from the Maven repository[3] has been randomly collected and named as `Dataset#1`. From `Dataset#1`, for projects with same prefix but different version numbers, we randomly selected only one among them and discarded the others. The removal resulted in a dataset consisting of 1,600 and we named it `Dataset#2`. Evaluation was performed on both datasets to see how well the system recommends API invocations with respect to different input data. By the evaluation, a dataset is split into two independent parts, namely a *training set* and a *testing set*. In practice, the items in the training data correspond to the OSS projects that have been collected a priori. They are available at developers' disposal, ready to be exploited for any mining purpose. Whereas, each item in the testing data represents the project being developed, or *the active project*. In this sense, an evaluation attempts to mimic a real situation: *the recommender system should produce recommendations for a project based on the data available from a set of existing projects*. We opt for *ten-fold cross validation* [9] as it has been popularly chosen for evaluating a model in Machine Learning. By this method, the dataset is divided into 10 equal parts, so-called *folds*. For each validation round, one fold is used as testing data and the remaining 9 folds are used as training data.

We simulate different stages of a development process to study if our proposed system is applicable to a real deployment, by considering a programmer who is developing a software project p. By the evaluation, each item in the testing set is assumed to be p. To this end, some parts of p are removed to mimic an actual development. Given an original project p, the total number of method declarations it contains is called Δ. However for the testing, only δ declarations ($\delta < \Delta$) are used as input for recommendation and the rest is discarded. In practice, this corresponds to the situation when the developer already finished δ declarations, and she is now working on the *active declaration* d_a. For d_a, the developer has just

[3]https://mvnrepository.com/

written π invocations. In practice, δ is low at an early stage and increases over the course of time. Similarly, π is small when the developer just starts working on d_a. The two parameters δ, π are used to stimulate different development phases. In particular, we consider the following configurations.

Configuration C#1: $\delta = \Delta/2 - 1, \pi = 1$. Almost a half of the declarations are used as testing data and the other half are removed, one declaration is selected as testing. For the testing declaration, only one invocation is provided as query, and the rest is used as ground-truth data which is called `GT(p)`. This configuration mimics a scenario when the developer is at an early stage of the development process and therefore, only limited context data is available for feeding the recommendation engine.

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

For a testing project p, the outcome of a recommendation process is a ranked list of invocations, i.e. `REC(p)`. It is expected that the proposed system recommends items that eventually match with those stored as ground-truth data `GT(p)`. Normally, a developer pays attention only to the *top-N* items, we use *success rate* and *accuracy* as the evaluation metrics, considering N as the cut-off value.

Success rate. Given a set of P testing projects, this metric measures the rate at which the recommendation engine can return at least a match among *top-N* recommended items for every project $p \in P$. The metric is formally defined as follows [12]:

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

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

Accuracy. Precision and recall are employed to measure accuracy [4]. *Precision@N* is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

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

and *recall@N* is the ratio of the ground-truth items appearing in the *top-N* items:

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

3.3 Result Analysis

Fig. 4(a) and Fig. 4(b) depict the success rate obtained for different cut-off values, i.e. $N = \{1, 5, 10, 15, 20\}$ for both configurations `C#1` and `C#2`. We investigate the outcome by each dataset, i.e. `Dataset#1` and `Dataset#2` separately. Interestingly, there are no big differences in success rate between two configurations for all values of N by both `Dataset#1` and `Dataset#2`. Considering that by `C#1` only a half of the declarations are used as input for recommendation. This demonstrates that the recommender system is able to generate relevant recommendations also when only limited background data is available, i.e. the developer doesn't write much. By comparing the Fig. 4(a) and Fig. 4(b) we see that the system produces better matches given that more similar projects are available, as in `Dataset#1` there exist similar projects with different version numbers. This implies that the system can efficiently exploit background data for recommendation.

For a small cut-off value N, i.e. $N = 1$, that means when the developer expects a very brief list of recommendations, the system is still able to generate matches. For example, with `Dataset#1`, the success rates of `C#1` and `C#2` are `72.30%` and `72.80%`, respectively. Meanwhile, the outcome by `Dataset#2` is much lower for both configurations with $N = 1$, the success rates of `C#1` and `C#2` are `49.40%` and `50.10%`, respectively. This implies that the performance of the system improves substantially, given that more similar projects are available.

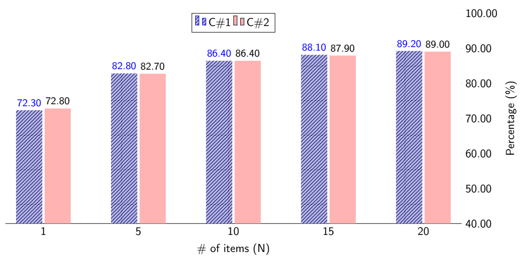
Next, we investigate the accuracy of both configurations by varying the cut-off value N from 1 to 30 to get *Precision@N* and *Recall@N* and sketching the Precision-Recall curves as shown in Figure 5(a) and Figure 5(b). Since a curve being close to the upper right corner represents better accuracy [4], we see that by `Dataset#1`, a superior performance is obtained by configuration `C#2`, i.e. when more background data is available for recommendation in comparison to `C#1`. For `Dataset#2`, we witness the same trend as by success rate, there are no distinctions between two configurations `C#1` and `C#2`. Considering both Figure 5(a) and Figure 5(b), it can be seen that the overall accuracy for `Dataset#1` is much better than that of `Dataset#2`. The maximum precision and recall for `Dataset#1` are `0.75` and `0.62`, respectively. Whereas the maximum precision and recall for `Dataset#2` are `0.52` and `0.36`, respectively. This further confirms the fact that with more similar projects, the system can provide better recommendations.

We come to the conclusion that the proposed system is able to provide relevant invocations with respect to different amount of input data. Furthermore, it works effectively given that more similar projects are available for recommendation. In practice, that means it is expected that we can find as much complete projects as possible since the more relevant background data we have, the higher is the possibility we are able to mine relevant invocations. We confirm the importance of the ability to search for similar OSS projects [13].

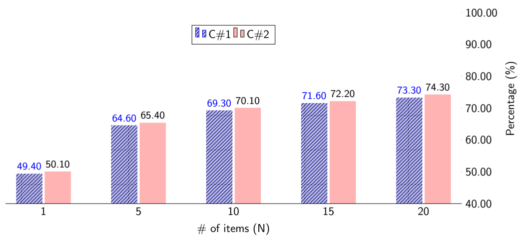
4 RELATED WORK

In [12], we presented a framework for supporting software developers. The work introduces a preliminary version of a Knowledge Graph to build a recommender system to provide third-party libraries recommendation. Furthermore, it also proposes a set of quality metrics for evaluating recommendation outcomes. Similarly, in [13] a graph structure has been devised to compute similarity among OSS projects. The authors in [2] exploit a knowledge graph to build connections among several cells in a neural network. They investigate how to extract and weight semantic features from the knowledge graph to mitigate the cold user problem and eventually to build a recommender system.

Several studies have been conducted to solve the problem of API function calls recommendation. MAPO was among the first approaches that mine API usage patterns from client code projects

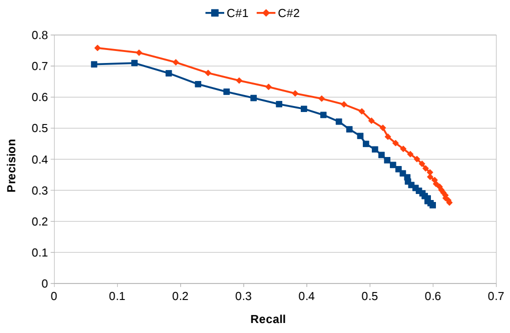


(a) Success rate for Dataset#1

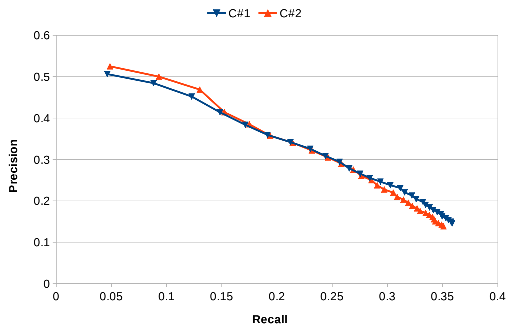


(b) Success rate for Dataset#2

Figure 4: Success rate



(a) Precision and recall for Dataset#1



(b) Precision and recall for Dataset#2

Figure 5: Accuracy

[18]. The system analyzes source files to collect API usage information and groups the API methods into clusters. Afterwards, it mines API usage patterns from the clusters, ranks them according to the similarity with developer context, and eventually recommends complete API code snippets to developers. MUSE is a practical way to recommend code examples related to a specific function [11]. MUSE parses source code to extract method usage, it simplifies examples and detects clones to group similar code snippets. Furthermore, it is able to rank recommendation outcomes according to various characteristics, i.e. reusability, understandability and popularity. *Wang et al.* proposed UP-Miner, aiming at reducing redundancy as well as covering a wide range of API usage patterns from source code [17]. From an input API method, the technique automatically finds all usage patterns and returns related code snippets. Both clustering steps adopt the complete linkage technique and they rely on sequence similarity functions. The experimental results show that UP-Miner outperforms some baselines with respect to different quality metrics.

MLUP [15] is an approach for mining multi-level API usage patterns, which are clusters of methods that co-exist in a method performing a specific functionality. This technique analyses the frequency and consistency of co-usage relations among APIs from different source code projects. MLUP is able to identify usage patterns regardless of the variability of features and usage scenarios. As input, the technique takes the source code and extracts the relevant methods of the considered API. Each API public method is characterized by a vector, where each entry corresponds to a client method. The DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering technique is then used to group API methods that are usually used together by projects.

Most tools mentioned in this section use clustering as the only way to identify relevant API function calls, with the aim of removing redundant items [8, 18]. However, as shown in [5], there is still a substantial amount of redundancy by those approaches that rely on clustering. Our proposed approach is novel since it attempts to exploit the underlying semantic in source code and directly mines API calls from similar projects and thus obtaining a promising outcome as demonstrated in Section 3.

5 CONCLUSIONS

Aiming to assist developers in their development activities by mining open source software repositories, we exploit a Knowledge Graph to encode the relationships among several OSS artifacts and build a knowledge-aware recommender system for providing API function calls. An evaluation on two datasets curated from the Maven repository shows that the proposed approach obtains a good performance with respect to two quality indicators. We believe that the deployment of a knowledge-aware recommender system is beneficial to the context of software development. For future work, we are going to thoroughly evaluate our proposed approach by using other similar techniques as baseline, with the consideration of more data.

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] Gediminas Adomavicius and Alexander Tuzhilin. 2008. Context-aware Recommender Systems. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys '08)*. ACM, New York, NY, USA, 335–336. https://doi.org/10.1145/1454008.1454068

[2] Vito Bellini, Vito Walter Anelli, Tommaso Di Noia, and Eugenio Di Sciascio. 2017. Auto-Encoding User Ratings via Knowledge Graphs in Recommendation Scenarios. In *Proceedings of the 2Nd Workshop on Deep Learning for Recommender Systems (DLRS 2017)*. ACM, New York, NY, USA, 60–66. https://doi.org/10.1145/3125486.3125496

[3] Annie Chen. 2005. Context-Aware Collaborative Filtering System: Predicting the User's Preference in the Ubiquitous Computing Environment. In *Proceedings of LoCA'05*. Springer-Verlag, Berlin, Heidelberg, 244–253. https://doi.org/10.1007/11426646_23

[4] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. 2012. Linked Open Data to Support Content-based Recommender Systems. In *Proceedings of the 8th International Conference on Semantic Systems (I-SEMANTICS '12)*. ACM, 1–8. https://doi.org/10.1145/2362499.2362501

[5] Jaroslav Fowkes and Charles Sutton. 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, NY, USA, 254–265. https://doi.org/10.1145/2950290.2950319

[6] Guibing Guo, Jie Zhang, and Neil Yorke-Smith. 2013. A Novel Bayesian Similarity Measure for Recommender Systems. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI '13)*. AAAI Press, 2619–2625. http://dl.acm.org/citation.cfm?id=2540128.2540506

[7] Alexandros Karatzoglou, Xavier Amatriain, Linas Baltrunas, and Nuria Oliver. 2010. Multiverse Recommendation: N-dimensional Tensor Factorization for Context-aware Collaborative Filtering. In *Proceedings of the Fourth ACM Conference on Recommender Systems (RecSys '10)*. ACM, New York, NY, USA, 79–86. https://doi.org/10.1145/1864708.1864727

[8] Nikolaos Katirtzis, Themistoklis Diamantopoulos, and Charles Sutton. 2018. Summarizing Software API Usage Examples Using Clustering Techniques. In *Fundamental Approaches to Software Engineering*, Alessandra Russo and Andy Schürr (Eds.). Springer International Publishing, Cham, 189–206.

[9] Ron Kohavi. 1995. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1137–1143. http://dl.acm.org/citation.cfm?id=1643031.1643047

[10] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2010. Recommending Source Code Examples via API Call Usages and Documentation. In *Proceedings of RSSE'10*. ACM, 21–25. https://doi.org/10.1145/1808920.1808925

[11] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 880–890. http://dl.acm.org/citation.cfm?id=2818754.2818860

[12] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. 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*. http://ceur-ws.org/Vol-2140/paper9.pdf

[13] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. 2018. CrossSim: Exploiting Mutual Relationships to Detect Similar OSS Projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 388–395. https://doi.org/10.1109/SEAA.2018.00069

[14] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of MSR 2014*. ACM, 102–111. https://doi.org/10.1145/2597073.2597077

[15] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. 2015. Mining Multi-level API Usage Patterns. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 23–32. https://doi.org/10.1109/SANER.2015.7081812

[16] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based Collaborative Filtering Recommendation Algorithms. In *Proceedings of WWW '01*. ACM, 285–295. https://doi.org/10.1145/371920.372071

[17] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 319–328. https://doi.org/10.1109/MSR.2013.6624045

[18] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, 318–343.