FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns

Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, Massimiliano Di Penta

FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns

Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio
Università degli Studi dell'Aquila
L'Aquila, Italy
{firstname.lastname}@univaq.it

Lina Ochoa, Thomas Degueule
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
{firstname.lastname}@cwi.nl

Massimiliano Di Penta
Università degli Studi del Sannio
Benevento, Italy
dipenta@unisannio.it

***Abstract*—Software developers interact with APIs on a daily basis and, therefore, often face the need to learn how to use new APIs suitable for their purposes. Previous work has shown that recommending *usage patterns* to developers facilitates the learning process. Current approaches to usage pattern recommendation, however, still suffer from high redundancy and poor run-time performance. In this paper, we reformulate the problem of usage pattern recommendation in terms of a collaborative-filtering recommender system. We present a new tool, FOCUS, which mines open-source project repositories to recommend API method invocations and usage patterns by analyzing how APIs are used in projects similar to the current project. We evaluate FOCUS on a large number of Java projects extracted from GitHub and Maven Central and find that it outperforms the state-of-the-art approach PAM with regards to success rate, accuracy, and execution time. Results indicate the suitability of context-aware collaborative-filtering recommender systems to provide API usage patterns.**

I. Introduction

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails the use of external libraries. Rather than implementing new systems from scratch, developers look for, and try to integrate into their projects, libraries that provide functionalities of interest. Libraries expose their functionality through Application Programming Interfaces (APIs) which govern the interaction between a client project and the libraries it uses.

Developers therefore often face the need to learn new APIs. The knowledge needed to manipulate an API can be extracted from various sources: the API source code itself, the official website and documentation, Q&A websites such as StackOverflow, forums and mailing lists, bug trackers, other projects using the same API, etc. However, official documentation often merely reports the API description without providing non-trivial example usages. Besides, querying informal sources such as StackOverflow might become time-consuming and error-prone [32]. Also, API documentation may be ambiguous, incomplete, or erroneous [42], while API examples found on Q&A websites may be of poor quality [18].

Over the past decade, the problem of API learning has garnered considerable interest from the research community. Several techniques have been developed to automate the extraction of API *usage patterns* [33] 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 [23], [43], [45] or predictive modeling [10], still suffer from high redundancy [10] and—as we show later in the paper—poor run-time performance.

To cope with these limitations, we propose a new approach for API usage patterns mining that builds upon concepts emerging from collaborative-filtering recommender systems [36]. The fundamental idea of these systems is to recommend to users items that have been bought by similar users in similar contexts. By considering API methods as products and client code as customers, we reformulate the problem of usage pattern recommendation in terms of a collaborative-filtering recommender system. Informally, the question the proposed system can answer is:

> *"Which API methods should this piece of client code invoke, considering that it has already invoked these other API methods?"*

Implementing a collaborative-filtering recommender system requires to assess the similarity of two customers, i.e., two projects. Existing approaches assume that any two projects using an API of interest are equally valuable sources of knowledge. Instead, we postulate that not all projects are equal when it comes to recommending usage patterns: a project that is highly similar to the project currently being developed should provide higher quality patterns than a highly dissimilar one. Our recommender system attempts to narrow down the search scope by considering only the projects that are the most similar to the active project. Thus, methods that are typically used conjointly by similar projects in similar contexts tend to be recommended first.

We incorporate these ideas into a recommender system that mines open-source software (OSS) repositories to provide developers with API *FunctiOn Calls and USage patterns*: FOCUS. Our approach represents mutual relationships between projects using a 3D matrix and mines API usage from the most similar projects.

We evaluated FOCUS on different datasets comprising 610 Java projects from GitHub and $3,600$ JAR archives from the Maven Central Repository. In the evaluation, we simulate different stages of a development process, by removing portions of client code and assessing how FOCUS can recommend snippets with API invocations to complete them. We find that

```
public List<Boekrekening> findBoekrekeningen() {
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Boekrekening> criteriaQueryBoekrekening = cb
                                                         .createQuery(
                                                          Boekrekening.class
                                                         );
 Root<BoekrekeningPO> boekrekeningFrom = criteriaQueryBoekrekening
                                         .from(BoekrekeningPO.class);



}
```

(a) Initial version

```
public List<Boekrekening> findBoekrekeningen() {
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Boekrekening> criteriaQueryBoekrekening = cb
                                                         .createQuery(
                                                          Boekrekening.class
                                                         );
 Root<BoekrekeningPO> boekrekeningFrom = criteriaQueryBoekrekening
                                         .from(BoekrekeningPO.class);

 criteriaQueryBoekrekening.select(boekrekeningFrom);
 criteriaQueryBoekrekening.orderBy(cb.asc(boekrekeningFrom
                                          .get(BoekrekeningPO_.rekeningnr)));
 return entityManager.createQuery(criteriaQueryBoekrekening).getResultList();
}
```

(b) Final version

Fig. 1. Motivating example

FOCUS outperforms PAM, a state-of-the-art tool for API usage patterns mining [10], with regards to success rate, accuracy, and execution time.

This paper is organized as follows. Section II introduces a motivating example and background notions. Our recommender system for API mining, FOCUS, is introduced in Section III. The evaluation is presented in Section IV, with the key results being analyzed in Section V. Section VI discusses the threats to validity. In Section VII, we present related work and conclude the paper in Section VIII.

II. Background

This section presents a motivating example for introducing the problem addressed by this paper and the main components of the proposed solution. Then, we introduce the main notions underpinning our approach, mostly originating from Schafer et al. [37] and Chen [4].

A. Motivating Example

The typical setting considered in the paper is as shown in Fig. 1: (a) developer is implementing some method to satisfy the requirements of the system being developed. In the specific case shown in Fig. 1 (b), the `findBoekrekeningen` method queries the available entities and retrieve those of type `Boekrekening`. To this end, the `Criteria API` library[1] is used.

Fig. 1 (a) depicts the situation where the development is at an early stage and the developer already used some methods of the chosen API to develop the required functionality. However, she is not sure how to proceed from this point. In such cases, different sources of information may be consulted, such as StackOverflow, video tutorials, API documentation, etc. In this paper, we propose an approach aiming at providing developers with recommendations consisting of a list of API method calls that should be used next, and with usage patterns that can be used as a reference for completing the development of the method being defined (e.g., code snippets that could support developers in completing the method definition with the framed code in Fig. 1 (b)).

[1] https://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html

B. API Function Calls and Usage Patterns

A *software project* is a standalone source code unit that performs a set of tasks. Furthermore, an *API* is an interface that abstracts the functionalities offered by a project by hiding its implementation details. This interface is meant to support reuse and modularity [24], [32]. An API X built in an object-oriented programming language (e.g., the `Criteria API` in Fig. 1) consists of a set T_X of public types (e.g., `CriteriaBuilder` and `CriteriaQuery`). Each type in T_X consists of a set of public methods and fields that are available to client projects (e.g., the method `createQuery` of the type `CriteriaQuery`).

A *method declaration* consists of a name, a (possibly empty) list of parameters, a return type, and a (possibly empty) body (e.g., the method `findBoekrekeningen` in Fig. 1). Given a set of declarations D in a project P, an API *method invocation* i is a call made from a declaration $d \in D$ to another declaration m. Similarly, an API *field access* is an access to a field $f \in F$ from a declaration d in P. API method invocations MI and field accesses FA in P form the set of API usages $U = MI \cup FA$. Finally, an *API usage pattern* (or code snippet) is a sequence $(u_1, u_2, ..., u_n)$, $\forall u_k \in U$ [19].

C. Context-aware Collaborative Filtering

As stated by Schafer et al. [37] "*Collaborative Filtering* (CF) is the process of filtering or evaluating items through the opinions of other people." In a CF system, a *user* who buys or uses an *item* attributes a rating to it based on her experience and perceived value. Therefore, a *rating* is the association of a user and an item through a value in a given unit (usually in scalar, binary, or unary form). The set of all ratings of a given user is also known as a *user profile* [4]. Moreover, the set of all ratings given in a system by existing users can be represented in a so-called *rating matrix*, where a row represents a user and a column represents an item.

The expected outcome of a CF system is a set of predicted ratings (aka. *recommendations*) for a specific user and a subset of items [37]. The recommender system considers the most similar users (aka. *neighbors*) to the *active* user to suggest new ratings. A similarity function $sim_{usr}(u_a, u_j)$ computes the *weight* of the active user profile u_a against each of the user profiles u_j in the system. Finally, to suggest a recommendation for an item i based on this subset of similar profiles, the CF system computes a weighted average $r(u_a, i)$ of the existing ratings, where $r(u_a, i)$ varies with the value of $sim_{usr}(u_a, u_j)$ obtained for all neighbors [4], [37].

Context-aware CF systems compute recommendations based not only on neighbors' profiles but also on the *context* where the recommendation is demanded. Each rating is associated with a context [4]. Therefore, for a tuple C modeling different contexts, a *context similarity* metric $sim_{ctx}(c_a, c_i)$, for $c_a, c_i \in C$ is computed to identify relevant ratings according to a given context. Then, the weighted average is reformulated as $r(u_a, i, c_a)$ [4].

III. Proposed Approach

To tackle the problem of recommending API function calls and usage patterns, we leverage the wisdom of the crowd and existing recommender system techniques. In particular, we hypothesize that API calls and usages can be mined from existing codebases, prioritizing the projects that are similar to the one from where the recommendation is demanded.

More specifically, our tool FOCUS adopts a context-aware CF technique to search for invocations from closely relevant projects. This technique allows us to consider both project and declaration similarities to recommend API function calls and usage patterns. Following the terminology of recommender systems, we treat *projects* as the enclosing *contexts*, *method declarations* as *users*, and *method invocations* as *items*. Intuitively, we recommend a method invocation for a declaration in a given project, which is analogous to recommending an item to a user in a given context. For instance, the set of method invocations and the usage pattern (cf. framed code in Fig. 1 (b)) recommended for the declaration `findBoekrekeningen` can be obtained from a set of similar projects and declarations in a codebase. The *collaborative* aspect of the approach enables to extract recommendations from the most similar projects, while the *context-awareness* aspect enables to narrow down the search space further to similar declarations.

A. Architecture

The architecture of FOCUS is depicted in Fig. 2. To provide its recommendations, FOCUS considers a set of *OSS Repositories* ①. The *Code Parser* ② component extracts method declarations and invocations from the source code or bytecode of these projects. The *Project Comparator*, a subcomponent of the *Similarity Calculator* ③, computes the similarity between projects in the repositories and the project under development. Using the set of projects and the information extracted by the *Code Parser*, the *Data Encoder* ④ component computes rating matrices which are introduced later in this section. Afterwards, the *Declaration Comparator* computes the similarities between declarations. From the similarity scores, the *Recommendation Engine* ⑤ generates recommendations, either as a ranked list of API function calls using the *API Generator*, or as usage patterns using the *Code Builder*, which are presented to the developer. In the remainder of this section, we present in greater details each of these components.

1) Code Parser: FOCUS relies on Rascal M^3 [2], an intermediate model that performs static analysis on the source code, to extract method declarations and invocations from a set of projects. This model is an extensible and composable algebraic data type that captures both language-agnostic and Java-specific facts in immutable binary relations. These relations represent program information such as existing *declarations*, *method invocations*, *field accesses*, *interface implementations*, *class extensions*, among others [2]. To gather relevant data, Rascal M^3 leverages the Eclipse JDT Core Component[2] to build and traverse the abstract syntax trees of the target Java projects.

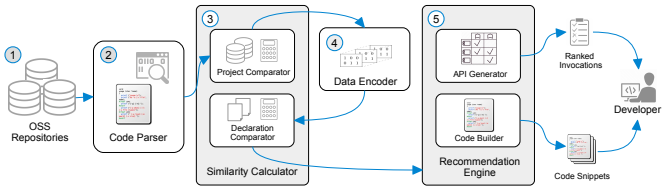

Fig. 2. Overview of the FOCUS architecture

In the context of FOCUS, we consider the data provided by the *declarations* and *methodInvocation* relations of the M^3 model. Both of them contain a set of pairs $\langle v_1, v_2 \rangle$, where v_1 and v_2 are values representing *locations*. These locations are uniform resource identifiers that represent artifact identities (aka. logical locations) or physical pointers on the file system to the corresponding artifacts (aka. physical locations). The *declarations* relation maps the logical location of an artifact (e.g., a method) to its physical location. The *methodInvocation* relation maps the logical location of a *caller* to the logical location of a *callee*. We refer the reader to a dedicated paper for the technical details of the inference of Java M^3 models [2].

Listing 1. Excerpt of the M^3 model extracted from Fig. 1

```
m3.declarations = {
<|java+method://StandaardBoekrekeningService/findBoekrekeningen|,
|file:// ... /StandaardBoekrekeningService.java
     (501,531,<17,4>,<33,5>)|>,
%[...]}
m3.methodInvocation = {
<|java+method://StandaardBoekrekeningService/findBoekrekeningen|,
|java+method://EntityManager/getCriteriaBuilder|>, [...]}
```

Listing 1 depicts an excerpt of the M^3 model extracted from the code presented in Fig. 1 (a). The *declarations* relation links the logical location of the method `findBoekrekeningen`, to its corresponding physical location in the file system. The *methodInvocation* relation states that the `getCriteriaBuilder` method of the `EntityManager` type is invoked by the `findBoekrekeningen` method in the current project.

2) Data Encoder: Once method declarations and invocations are extracted, FOCUS represents the relationships among them using a rating matrix. For a given project, each row in the matrix represents a method declaration and each column represents a method invocation. A cell is set to 1 if the declaration in the corresponding row contains the invocation in the column, otherwise it is set to 0. For example, Fig. 3 shows the rating matrix of a project with four declarations $p_1 \ni (d_1, d_2, d_3, d_4)$ and four invocations (i_1, i_2, i_3, i_4).

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{array} \begin{array}{c} \begin{array}{cccc} i_1 & i_2 & i_3 & i_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 3. Rating matrix for a project with 4 declarations and 4 invocations

To capture the intrinsic relationships among various projects, declarations, and invocations, we come up with a 3D context-based rating matrix [21]. The third dimension of this matrix

[2] https://www.eclipse.org/jdt/core/

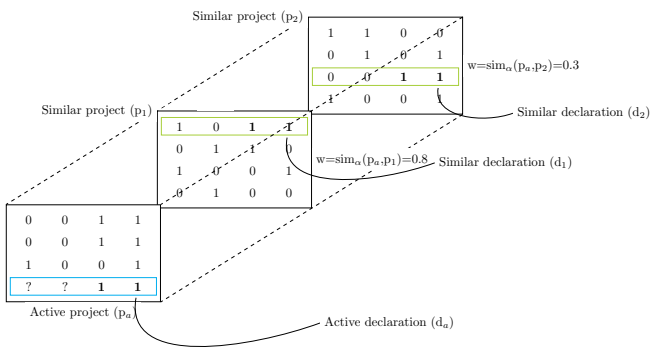

Fig. 4. 3D context-based rating matrix

represents a project, which is analogous to the so-called context in context-aware CF systems. For example, Fig. 4 depicts three projects $P = (p_a, p_1, p_2)$ represented by three slices with four method declarations and four method invocations. Project p_1 has already been introduced in Fig. 3 and for the sake of readability, the column and row labels are removed from all slices in Fig. 4. There, p_a is the *active project* and it has an *active declaration*. *Active* here means the artifact (project or declaration), being considered or developed. Both p_1 and p_2 are complete projects similar to the active project p_a. The former projects (i.e., p_1 and p_2) are also called *background data* since they are already available and serve as a base for the recommendation process. In practice, the higher the number of complete projects considered as background data, the higher the probability to recommend relevant invocations.

3) Similarity Calculator: Exploiting the context-aware CF technique, the presence of additional invocations is deduced from similar declarations and projects. Given an active declaration in an active project, it is essential to find the subset of the most similar projects, and then the most similar declarations in that set of projects. To compute similarities, we derive from [20] a weighted directed graph that models the relationships among projects and invocations. Each node in the graph represents either a project or an invocation. If project p contains invocation i, then there is a directed edge from p to i. The weight of an edge $p \rightarrow i$ represents the number of times a project p performs the invocation i. Fig. 5 depicts the graph for the set of projects introduced in Fig. 4. For instance, p_a has four declarations and all of them invoke i_4. As a result, the edge $p_a \rightarrow i_4$ has a weight of 4. In the graph, a question mark represents missing information. For the active declaration in p_a, it is not known yet whether invocations i_1 and i_2 should be included.

The similarity between two project nodes p and q is computed by considering their feature sets [7]. Given that p has a set of neighbor nodes $(i_1, i_2, .., i_l)$, the feature set of p is the vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_k being the weight of node i_k. This weight is computed as the *term-frequency inverse document frequency* value, i.e., $\phi_k = f_{i_k} * log(\frac{|P|}{a_{i_k}})$, where f_{i_k} is the weight of the edge $p \rightarrow i_k$; $|P|$ is the number

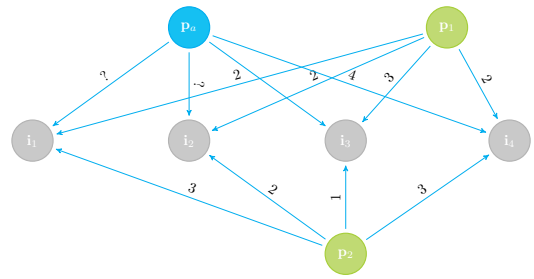

Fig. 5. Graph representation of projects and invocations

of all considered projects; and a_{i_k} is the number of projects connected to i_k. Eventually, the similarity between p and q with their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_k\}_{k=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$ is:

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

The similarities among method declarations are calculated using the Jaccard similarity index [15] as follows:

$$sim_\beta(d, e) = \frac{|\mathbb{F}(d) \bigcap \mathbb{F}(e)|}{|\mathbb{F}(d) \bigcup \mathbb{F}(e)|} \quad (2)$$

where $\mathbb{F}(d)$ and $\mathbb{F}(e)$ are the sets of invocations made from declarations d and e, respectively.

4) API Generator: This component, which is part of the *Recommendation Engine*, is in charge of generating a ranked list of API function calls. In Fig. 4, the active project p_a already includes three declarations, and the developer is working on the fourth declaration, which corresponds to the last row of the slice. p_a has only two invocations, represented in the last two columns of the matrix (i.e., cells filled with 1). The first two cells are marked with a question mark (?), indicating that it is unclear whether these two invocations should also be added into p_a. The recommendation engine attempts to predict additional invocations for the active declaration by computing the missing ratings using the following formula [4]:

$$r_{d,i,p} = \overline{r}_d + \frac{\sum_{e \in topsim(d)} (R_{e,i,p} - \overline{r}_e) \cdot sim_\beta(d, e)}{\sum_{e \in topsim(d)} sim_\beta(d, e)} \quad (3)$$

Eq. 3 is used to compute a score for the cell representing method invocation i, declaration d of project p, where $topsim(d)$ is the set of top similar declarations of d; $sim_\beta(d, e)$ is the similarity between d and a declaration e, computed using Eq. 2; $\overline{r}_d$ and $\overline{r}_e$ are the mean ratings of d and e, respectively; and $R_{e,i,p}$ is the combined rating of declaration d for i in all similar projects, computed as follows [4]:

$$R_{e,i,p} = \frac{\sum_{q \in topsim(p)} r_{e,i,q} \cdot sim_\alpha(p, q)}{\sum_{q \in topsim(p)} sim_\alpha(p, q)} \quad (4)$$

where $topsim(p)$ is the set of top similar projects of p; and $sim_\alpha(p, q)$ is the similarity between p and a project q, computed using Eq. 1. Eq. 4 implies that a higher weight is given to projects with higher similarity. In practice, it is reasonable since, given a project, its similar projects contain

```
public List<QuestionsStaged> findByIdentifier(String identifier) {
    log.fine("getting Session instance by identifier: " + identifier);
    try {
      CriteriaBuilder cb = entityManager.getCriteriaBuilder();
      CriteriaQuery<QuestionsStaged> criteria = cb.createQuery(QuestionsStaged.class);
      Root<QuestionsStaged> qs = criteria.from(QuestionsStaged.class);
      criteria.select(qs).where(cb.equal(qs.get("identifier"), identifier));
      log.fine("get identifier successful");
      return entityManager.createQuery(criteria).getResultList();

    } catch (RuntimeException re) {
      log.severe("get identifier failed" + re);
      throw re;
    }
  }
```

Fig. 6. Real source code recommended by FOCUS

more relevant API calls than less similar projects. Using Eq. 3 we compute all the missing ratings in the active declaration and get a ranked list of invocations with scores in descending order, which is then suggested to the developer.

5) Code Builder: This subcomponent is also part of the *Recommendation Engine*, and it is responsible for recommending usage patterns to developers. From the ranked list, *top-N* method invocations are used as a query to search the database for relevant declarations. To limit the search scope, only the most similar projects are considered. The Jaccard index is used to compute similarities between the selected invocations and a given declaration. For each query, we search for declarations that contain as many invocations of the query as possible. Once we identify the corresponding declarations we retrieve their source code using the *declarations* relation of the Rascal M^3 model. The resulting code snippet is then recommended to the developer.

For the sake of illustration, we now present an example of how FOCUS suggests real code snippets, considering the declaration `findBoekrekeningen` in Fig. 1 (a) as input. The invocations it contains are used together with the other declarations in the current project as query to feed the *Recommendation Engine*. The final outcome is a ranked list of real code snippets. The top one, named `findByIdentifier`, is depicted in Fig. 6. By carefully examining this code and the original one in Fig. 1 (b), we see that although they are not exactly the same, they indeed share several method calls and a common intent: both exploit a `CriteriaBuilder` object to build, perform a query and eventually get back some results. Furthermore, the outcome of both declarations is of the `List` type. Interestingly, compared to the original one, the recommended code appears to be of higher quality since it includes a `try/catch` construct to handle possible exceptions. Thus, the recommended code, coupled with the corresponding list of function calls (i.e., `get`, `equal`, `where`, `select`, etc.), provides the developer with helpful directions on how to use the API at hand to implement the desired functionality.

IV. Evaluation

The *goal* of this study is to evaluate FOCUS, and compare it with another state-of-the-art tool (PAM [10]), with the aim of assessing its capability to recommend API usage patterns to developers, while they are writing code. The *quality focus* is twofold: studying the API recommendation accuracy and completeness, as well as the time required by FOCUS and PAM to provide a recommendation. The *context* consists of 610 Java open source projects, and $3,600$ JAR archives from the Maven Central repository.[3] For the sake of reproducibility and ease of reference, all artifacts used in the evaluation, together with the tools are available online [22]. We choose PAM as a baseline for comparison, as it has been shown to outperform [10] other similar tools such as MAPO [45] and UP-Miner [43]. To conduct the comparison with PAM, we leverage its original source code made available online by its authors [9].

In the following, we detail our research questions, datasets, evaluation methodology, and metrics.

A. Research Questions

Our research questions are as follows:

RQ$_1$ *To what extent is FOCUS able to provide accurate and complete recommendations?* This research question relates to the capability of FOCUS to produce accurate and complete results. Having too many false positives would end up being counterproductive, whereas having too many false negatives would mean that the tool is not able to provide recommendations in many cases where this is needed.

RQ$_2$ *What are the timing performances of FOCUS in building its models and in providing recommendations?* This research question aims at assessing whether, from a timing point of view, FOCUS—compared to PAM—could be used in practice. We evaluate the time required by both tools to provide a recommendation. We mainly focus on the recommendation time because, while it is acceptable that the model training phase is relatively slow (i.e., the model could be built offline), the recommendation time has to be fast enough to make the tool applicable in practice.

RQ$_3$ *How does FOCUS perform compared with PAM?* Finally, this research question directly compares the recommendation capabilities of FOCUS and PAM.

B. Datasets

To answer our research questions, we relied on four different datasets. The first dataset, $\texttt{SH}_L$, has been assembled starting from $5,147$ randomly selected Java projects retrieved from GitHub via the Software Heritage archive [6]. To comply with the requirements of PAM, we first restricted the dataset to the list of projects that use at least one of the third-party libraries listed in Table I. Most of them were used to assess the performance of PAM [10]. Each row in Table I lists a third-party library, the number of projects that depend on it, and the number of classes that invoke methods of this library. To comply with the requirements of FOCUS, we then restricted the dataset to the list of projects containing at least one *pom.xml*, as it eases the creation of the M^3 models. We thus obtained our first dataset consisting of 610 Java projects.

From $\texttt{SH}_L$, we extracted a second dataset $\texttt{SH}_S$ consisting of the 200 smallest (in size) projects of $\texttt{SH}_L$.

As a third dataset, we randomly collected a set of $3,600$ JAR archives from the Maven Central repository, which we name $\texttt{MV}_L$. Through a manual inspection of $\texttt{MV}_L$, we noticed

[3] https://mvnrepository.com

TABLE I
EXCERPT OF THE THIRD-PARTY LIBRARIES USED BY DATASET $\mathtt{SH}_L$

Project Name	# of Client Projects	# of Client Classes
com.google.gson	51	337
io.netty	105	13,456
org.apache.camel	36	1,017
org.apache.hadoop	158	14,596
org.apache.lucene	15	397
org.apache.mahout	25	8,541
org.apache.wicket	44	3,360
org.drools	27	886
org.glassfish.jersey	105	3,811
org.hornetq	15	123
org.jboss.weld	39	1,875
org.jooq	16	243
org.jsoup	23	55
org.neo4j	28	4,983
org.restlet	19	326
org.springside	16	821
twitter4j	45	597
	610	**55,425**

that many projects only differ in their version numbers (*ant-1.6.5.jar* and *ant-1.9.3.jar*, for instance, are two versions of the same project *ant*). These cases are interesting as we assume two versions of the same project share many functionalities [39]. The collaborative-filtering technique works well given that highly similar projects exist, since it just "copies" invocations from similar methods in the very similar projects (see Eq. 3 and Eq. 4). However, a dataset containing too many similar projects may introduce a bias in the evaluation. Thus, we decided to populate one more dataset. Starting from $\mathtt{MV}_L$, we randomly selected one version for every project and filtered out the other versions. The removal resulted in a fourth dataset consisting of $1,600$ projects, which we name $\mathtt{MV}_S$.

Three datasets, i.e., $\mathtt{SH}_L$, $\mathtt{MV}_L$ and $\mathtt{MV}_S$ are used to assess the performance of FOCUS ($\mathbf{RQ}_1$). The smallest dataset $\mathtt{SH}_S$ is used to compare FOCUS and PAM ($\mathbf{RQ}_2$ and $\mathbf{RQ}_3$).

Eventually, the process of creating required metadata consists of the following main steps:

- for each project in the dataset the corresponding Rascal M^3 model is generated;
- for each M^3 model, the corresponding ARFF representations[4] are generated in order to be used as input for applying FOCUS and PAM during the actual evaluation steps discussed in the next sections.

C. Study Methodology

Performing a user study has been accepted as the standard method to validate an API usage recommendation tool [17], [45]. While user studies are valuable, they are limited in the size of the task a participant can conduct and are highly susceptible to individual skills and subjectiveness. In this paper, to study if FOCUS is applicable in real-world settings we perform a different, offline evaluation, by simulating the behavior of a developer working at different stages of a development project on partial code snippets.

[4]https://www.cs.waikato.ac.nz/ml/weka/arff.html

More specifically, we consider a programmer who is developing a project p. To this end, some parts of p are removed to mimic an actual development. Given an original project p, the total number of declarations it contains is called Δ. However, 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, originally there are Π invocations, however only the first π invocations ($\pi < \Pi$) are selected as query and the rest is removed and saved as ground-truth data for future comparison. In practice, δ is small 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 `C1.1` ($\delta = \Delta/2 - 1, \pi = 1$): Almost the first half of the declarations is used as testing data and the second half is removed. The last declaration of the first half is selected as the active declaration d_a. For d_a, only the *first* invocation is provided as a query, and the rest is used as ground-truth data which we call `GT(p)`. This configuration mimics a scenario where the developer is at an early stage of the development process and, therefore, only limited context data is available to feed the recommendation engine.

Configuration `C1.2` ($\delta = \Delta/2 - 1, \pi = 4$): Similarly to `C1.1`, almost the first half of the declarations is kept and the second half is discarded. d_a is the last declaration of the first half of declarations. For d_a, the first *four* invocations are provided as query, and the rest is used as `GT(p)`.

Configuration `C2.1` ($\delta = \Delta - 1, \pi = 1$): The last method declaration is selected as testing, i.e., d_a and all the remaining declarations are used as training data ($\Delta - 1$). In d_a, the *first* invocation is kept and all the others are taken out as ground-truth data `GT(p)`. This represents the stage where the developer almost finished implementing p.

Configuration `C2.2` ($\delta = \Delta - 1, \pi = 4$): Similar to `C2.1`, d_a is selected as the last method declaration, and all the remaining declarations are used as training data ($\Delta - 1$). The only difference with `C2.1` is that in d_a, the first *four* invocations are used as query and all the remaining ones are used as `GT(p)`.

When performing the experiments, we split a dataset into two independent parts, namely a *training set* and a *testing set*. In practice, the training set represents the OSS projects that have been collected a priori. They are available at developers' disposal, ready to be exploited for mining purposes. The testing set represents the project being developed, or *the active project*. This way, our evaluation mimics a real development scheme: *the system should produce recommendations for the active project based on the data from a set of existing projects*. We opt for *k-fold cross validation* [16] as it is widely chosen to evaluate machine learning models. Depending on the availability of input data, the dataset with n elements is divided into k equal parts, so-called *folds*. For each validation round, one fold is used as testing data and the remaining $k - 1$ folds are used as training data. For our evaluation, we select two values,

i.e., $k = 10$ and $k = n$. The former corresponds to *ten-fold cross validation* and the latter corresponds to *leave-one-out cross validation* [44].

D. Evaluation Metrics

For a testing project p, the outcome of a recommendation process is a ranked list of invocations, i.e., `REC(p)`. It is our firm belief that the ability to provide accurate invocations is important in the context of software development. Thus, we are interested in how well a system can recommend API invocations that eventually match with those stored in `GT(p)`. To measure the performance of the recommender systems, i.e., PAM and FOCUS, we utilize two metrics, namely *success rate* and *accuracy* [7]. Given a ranked list of recommendations, a developer typically pays attention to the *top-N* items only. *Success rate* and *accuracy* are computed by using N as the *cut-off value*. Given that $REC_N(p)$ is the set of *top-N* items and $match_N(p) = GT(p) \bigcap REC_N(p)$ is the set of items in the *top-N* list that match with those in the ground-truth data, then the metrics are defined as follows.

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

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} \times 100\% \tag{5}$$

where *count()* counts the number of times the boolean expression given as parameter evaluates to *true*.

Accuracy: Precision and recall are employed to measure accuracy [7]. *Precision@N* is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

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

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

$$recall@N = \frac{|match_N(p)|}{|GT(p)|} \tag{7}$$

Recommendation time: As mentioned in $\mathbf{RQ}_2$, we measure the time needed by both PAM and FOCUS to perform a prediction on a given infrastructure, which is a laptop with Intel Core i5-7200U CPU @ 2.50GHz $\times$ 4, 8GB RAM, and Ubuntu 16.04.

V. Results

$\mathbf{RQ}_1$: *To what extent is FOCUS able to provide accurate and complete recommendations?*

To answer this research question, we use the dataset $\mathtt{SH}_L$ and vary the length of the input data for every testing project. Two main configurations are taken into account, with two sub-configurations for each as introduced in Section IV-C. Table II shows the success rate for all the configurations. For a small N, i.e., $N = 1$ (the developer expects a very brief list of items) FOCUS is still able to provide matches. For example, the success rates of `C1.1` and `C1.2` are 24.59% and 30.65%, respectively. When the cut-off value N is increased, the corresponding success rates improve linearly. For example, when $N = 20$, FOCUS obtains 40.98% success rate for `C1.1` and 47.70% for `C1.2`. By comparing the results obtained for `C1.1` and `C1.2`, we see that when more invocations are incorporated into the query, FOCUS provides more precise recommendations. In practice, this means that the accuracy of recommendations improves with the maturity of the project.

TABLE II
SUCCESS RATE FOR $\mathtt{SH}_L$, $N = \{1, 5, 10, 15, 20\}$

N	$\mathtt{SH}_L$			
	C1.1	C1.2	C2.1	C2.2
1	24.59	30.65	23.44	29.83
5	31.96	40.00	31.31	39.01
10	35.90	43.77	35.73	43.77
15	39.34	47.21	37.70	45.57
20	40.98	47.70	39.34	46.88

We now consider the outcomes obtained for `C2.1` and `C2.2`. In these configurations, more background data is available for recommendation. For `C2.1` ($\delta = \Delta - 1$, $\pi = 1$), the success rates for the smallest values of N, i.e., $N = 1$ and $N = 5$ are 23.44% and 31.31%, respectively. In other words, it improves with N. The same trend can be observed with other cut-off values, i.e., $N = 10, 15, 20$: the success rates for these settings increase correspondingly. We notice the same pattern considering `C2.1` and `C2.2` together, or `C1.1` and `C1.2` together: if more invocations are used as query, FOCUS suggests more accurate invocations.

Fig. 7 and Fig. 8 depict the precision and recall curves (PRCs) for the above mentioned configurations by varying N from 1 to 30. In particular, Fig. 7 represents the accuracy when almost the first half of the declarations ($\delta = \Delta/2 - 1$) together with one (`C1.1`) and four invocations (`C1.2`) from the testing declaration d_a are used as query. As a PRC close to the upper right corner indicates a better accuracy [7], we see that the accuracy of `C1.2` is superior to that of `C1.1`. Similarly with `C2.1` and `C2.2`, as depicted in Fig. 8, the accuracy improves substantially when the query contains more invocations. These facts further confirm that FOCUS is able to recommend more relevant invocations when the developer keeps coding. This improvement is obtained since the similarity between declarations can be better determined when more invocations are available as comprehended in Eq. 4.

The results reported so far appear to be promising at the first sight. However, by considering Table II, Fig. 7, and Fig. 8 together, we realize that both success rate and accuracy are considerably low: The best success rate is 47.70% for `C1.2` when $N = 20$, which means that more than half of the queries do not get any matches at all. In this sense, it is necessary to ascertain the cause of this outcome: Is FOCUS only capable of generating such moderate recommendations, or is it because of the data? Our intuition is that $\mathtt{SH}_L$ is rather small in size, which means the background data available for the recommendation process is limited. Thus, to further validate the performance of FOCUS, we perform additional experiments by considering more data, using both $\mathtt{MV}_L$ and $\mathtt{MV}_S$. For this evaluation, we

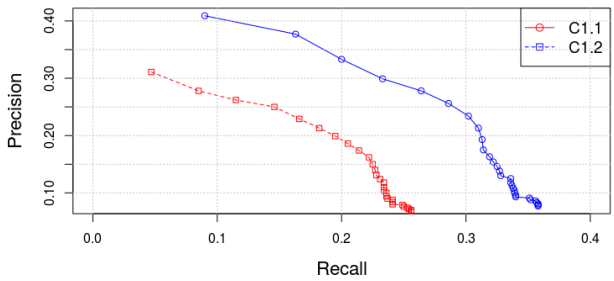

Fig. 7. Precision and recall for C1.1 and C1.2 on $\mathtt{SH}_L$

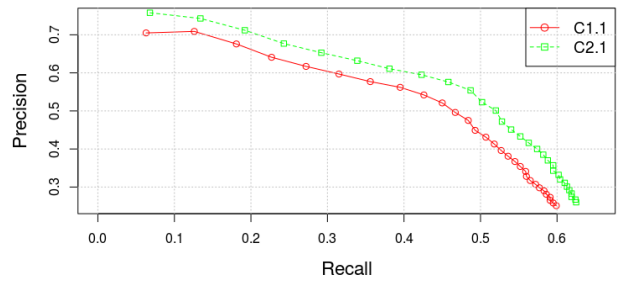

Fig. 9. Precision and recall for C1.1 and C2.1 on $\mathtt{MV}_L$

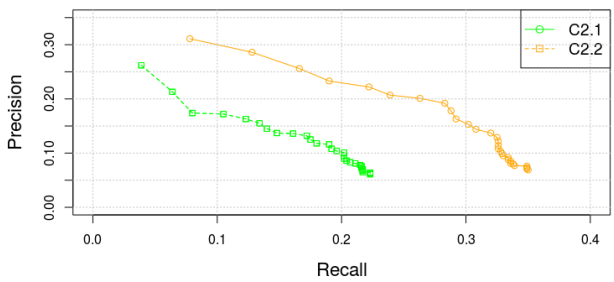

Fig. 8. Precision and recall for C2.1 and C2.2 on $\mathtt{SH}_L$

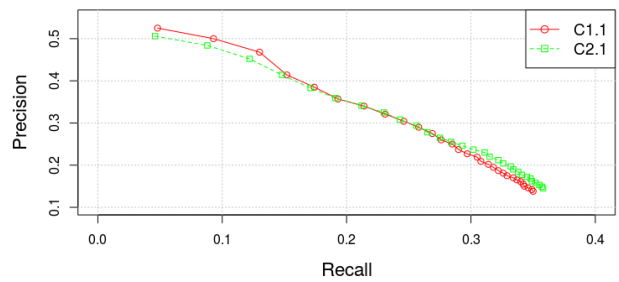

Fig. 10. Precision and recall for C1.1 and C2.1 on $\mathtt{MV}_S$

just consider the case when only one invocation together with other declarations are used as query, i.e., C1.1 and C2.1. This aims at validating the performance of FOCUS, given that the developer just finished only one invocation in d_a.

Table III depicts the success rate obtained for different cut-off values using both datasets. The success rates for all configurations are much better than those of $\mathtt{SH}_L$. The scores are considerably high, even when $N = 1$, the success rates obtained by C1.1 and C2.1 are 72.30% and 72.80%, respectively. For $\mathtt{MV}_S$, the corresponding success rates are lower. However, this is understandable since the set has less data compared to $\mathtt{MV}_L$.

The PRCs for $\mathtt{MV}_L$ and $\mathtt{MV}_S$ are shown in Fig. 9 and Fig. 10, respectively. We see that for $\mathtt{MV}_L$, a superior performance is obtained by configuration C2.1, i.e., when more background data is available for recommendation compared to C1.1. For $\mathtt{MV}_S$, we witness the same trend as with success rate: the difference between C1.1 and C2.1 is negligible. Considering both Fig. 9 and Fig. 10, we observe that the overall accuracy for $\mathtt{MV}_L$ is much better than that of $\mathtt{MV}_S$. The maximum precision and recall

TABLE III
SUCCESS RATE FOR MV_L AND MV_S, $N = \{1, 5, 10, 15, 20\}$

N	$\mathtt{MV}_L$		$\mathtt{MV}_S$	
	C1.1	C2.1	C1.1	C2.1
1	72.30	72.80	49.40	50.10
5	82.80	82.70	64.60	65.40
10	86.40	86.40	69.30	70.10
15	88.10	87.90	71.60	72.20
20	89.20	89.00	73.30	74.30

for $\mathtt{MV}_L$ are 0.75 and 0.62, respectively. Whereas, the maximum precision and recall for $\mathtt{MV}_S$ are 0.52 and 0.36, respectively. This further confirms the fact that with more similar projects, FOCUS can provide better recommendations. Referring back to the outcomes of $\mathtt{SH}_L$, we see that the performance on $\mathtt{MV}_L$ and $\mathtt{MV}_S$ is improved substantially.

To sum up, we conclude that the performance of FOCUS relies on the availability of background data. The system works effectively given that more OSS projects are available for recommendation. In practice, it is expected that we can crawl as many projects as possible, and use them as background data for the recommendation process.

RQ$_2$: *What are the timing performances of FOCUS in building its models and in providing recommendations?*

To measure the execution time of PAM and FOCUS, for the very first attempt we ran both systems on the $\mathtt{SH}_L$ dataset, consisting of 610 projects. With PAM, for each testing project, we combined the extracted query with all the other training projects to produce a single ARFF file provided as input for the recommendation process [10]. Nevertheless, we then realized that the execution of PAM is very time-consuming. For instance, for one fold containing 1 testing and 549 training projects (i.e., $610/10 \times 9$ training folds) with 80MB in size, PAM takes around 320 seconds to produce the final recommendations. Instead, the corresponding execution time by FOCUS is quite faster than PAM, around 1.80 seconds. Given the circumstances, it is not feasible to run PAM on a large dataset.

Therefore, we decided to use the $\mathtt{SH}_S$ dataset (consisting of 200 projects) for this purpose. For the experiments, we opt for *leave-one-out cross-validation* [44], i.e., one project is used

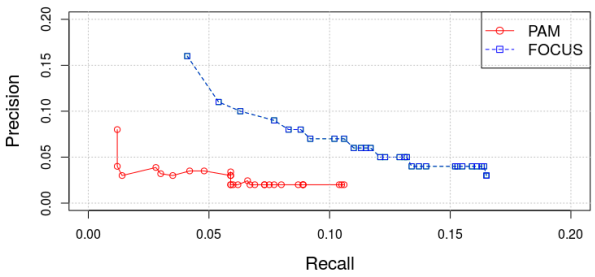

Fig. 11. Precision and recall for PAM and FOCUS using $\texttt{SH}_S$.

as testing, and all the remaining 199 projects are used for the training. The rationale behind the selection of this method instead of ten-fold cross-validation is that we want to exploit as much as possible the projects available as background data, given a testing project. The validation was executed 200 times, and we measured the time needed to finish the recommendation process. On average, PAM requires 9 seconds to provide each recommendation while FOCUS just needs 0.095 seconds, i.e., it is two orders of magnitude faster and suitable to be integrated into a development environment.

RQ$_3$: *How does FOCUS perform compared with PAM?*

For the reasons explained in **RQ$_2$**, the comparison between PAM and FOCUS has been performed on the $\texttt{SH}_S$ dataset. FOCUS gains a better success rate than PAM does, i.e., 51.20% compared to 41.60%. Furthermore, as depicted in Fig. 11, there is a big gap between the PRCs for PAM and FOCUS, with the one representing FOCUS closer to the upper right corner. This implies that the accuracy obtained by FOCUS is considerably superior to that of PAM.

A statistical comparison of PAM and FOCUS using Fisher's exact test [8] indicates that, for $1 \leq N \leq 20$, FOCUS always outperforms PAM: We achieved p-values < 0.001 (adjusted using the Holm's correction [13]) in all cases, with an Odds Ratio between 2.21 and 3.71, and equal to 2.54 for $N = 1$. In other words, FOCUS has over twice the odds of providing an accurate recommendation than PAM.

It is worth noting that the overall accuracy of FOCUS achieved and reported in this experiment is, although better than that of PAM, still considerably low. Following the experiments on $\texttt{MV}_L$ and $\texttt{MV}_S$ from **RQ$_1$**, we believe that this attributes to the limited background data available for the evaluation, since we only consider 200 projects.

In summary, by considering both **RQ$_2$** and **RQ$_3$**, we come to the conclusion that FOCUS obtains a better performance in comparison to PAM with regards to success rate, accuracy and execution time. Lastly, since PAM takes considerable time to produce the final recommendations, it might be impractical to deploy PAM in a development environment.

VI. Threats to Validity

The main threat to *construct validity* concerns the simulated setting used to evaluate the approaches, as opposed to performing a user study. We mitigated this threat by introducing four configurations that simulate different stages of the development process. In a real development setting, however, the order in which one writes statements might not fully reflect our simulation. Also, in a real setting, there may be cases in which a recommender is more useful, and cases (obvious code completion) where it is less useful. This makes a further evaluation involving developers highly desirable.

Threats to *internal validity* concern factors internal to our study that could have influenced the results. One possible threat can be seen through the results obtained for the datasets $\texttt{SH}_L$ and $\texttt{SH}_S$. As noted, these datasets exhibit lower precision/recall with respect to $\texttt{MV}_L$ and $\texttt{MV}_S$ due to the limited size of the training sets. However, these datasets were needed to compare FOCUS and PAM due to the limited scalability of PAM.

The main threat to *external validity* is that FOCUS is currently limited to Java programs. As stated in Section III, however, FOCUS makes few assumptions on the underlying language and only requires information about method declarations and invocations to build the 3D rating matrix. This information could be extracted from programs written in any object-oriented programming language, and we wish to generalize FOCUS to other languages in the future.

VII. Related Work

In this section, we summarize related work about API usage recommendation and relate our contributions to the literature.

A. API Usage Pattern Recommendation

Acharya et al. [1] present a framework to extract API patterns as partial orders from client code. While this approach proposes a representation for API patterns, suggestions regarding API usage are still missing.

MAPO (Mining API usage Pattern from Open source repositories) is a tool that mines API usage patterns from client projects [45]. MAPO collect API usages from source files, groups API methods into clusters. Then, it mines API usage patterns from the clusters, ranks them according to their similarity with the current development context, and recommends code snippets to developers. Similarly, UP-Miner [43] mines API usage patterns by relying on *SeqSim*, a clustering strategy that reduces patterns redundancy and improves coverage. Differently from FOCUS, these approaches are based on clustering techniques, and consider all client projects in the mining regardless of their similarity with the current project.

Fowkes et al. introduce PAM (Probabilistic API Miner), a parameter-free probabilistic approach to mine API usage patterns [10]. PAM uses the structural Expectation-Maximization (EM) algorithm to infer the most probable API patterns from client code, which are then ranked according to their probability. PAM outperforms both MAPO and UP-Miner (lower redundancy and higher precision). We directly compare FOCUS to PAM in Section IV.

Niu et al. extract API usage patterns using API class or method names as queries [23]. They rely on the concept of object usage (method invocations on a given API class) to

extract patterns. The approach of Niu et al. outperforms UP-Miner and Codota,[5], a commercial recommendation engine, in terms of coverage, performance, and ranking relevance. In contrast, FOCUS relies on context-aware CF techniques—which favors recommendations from similar projects and uses the whole development context to query API method calls.

The NCBUP-miner (Non Client-based Usage Patterns) [35] is a technique that identifies unordered API usage patterns from the API source code, based on both structural (methods that modify the same object) and semantic (methods that have the same vocabulary) relations. The same authors also propose MLUP [34], which is based on vector representation and clustering, but in this case client code is also considered.

DeepAPI [12] is a deep-learning method used to generate API usage sequences given a query in natural language. The learning problem is encoded as a machine translation problem, where queries are considered the source language and API sequences the target language. Only commented methods are considered during the search. The same authors [11] present CODEnn (COde-Description Embedding Neural Network), where, instead of API sequences, code snippets are retrieved to the developer based on semantic aspects such as API sequences, comments, method names, and tokens.

With respect to the aforementioned approaches, FOCUS uses CF techniques to recommend and rank API method calls and usage patterns from a set of similar projects. In the end, not only relevant API invocations are recommended, but also code snippets are returned to the developer as usage examples.

B. API-Related Code Search Approaches

Strathcona [14] is a recommender system used to suggest API usage. It is an Eclipse plug-in that extracts the structural context of code and uses it as a query to request a set of code examples from a remote repository. Six heuristics (associated to class inheritance, method calls, and field types) are defined to perform the match. Similarly, Buse and Weimer [3] propose a technique for synthesizing API usage examples for a given data type. An algorithm based on data-flow analysis, k-medoids clustering and pattern abstraction is designed. Its outcome is a set of syntactically correct and well-typed code snippets where example length, exception handling, variables initialization and naming, and abstract uses are considered.

Moreno et al. [17] introduce MUSE (Method USage Examples), an approach designed for recommending code examples related to a given API method. MUSE extracts API usages from client code, simplifies code examples with static slicing, and detects clones to group similar snippets. It also ranks examples according to certain properties (i.e., reusability, understandability, and popularity) and documents them.

SWIM (Synthesizing What I Mean) [28] seeks API structured call sequences (control and data-flows are considered), and then synthesizes API-related code snippets according to a query in natural language. The underlying learning model is also built with the EM algorithm. Similarly, Raychev et al. [30] propose a code completion approach based on natural language processing, which receives as input a partial program and outputs a set of API call sequences filling the gaps of the input. Both invocations and invocation arguments are synthesized considering multiple types of an API.

Thummalapenta and Xie propose SpotWeb [40], an approach that provides starting points (hotspots) for understanding a framework, and highlights where examples finding could be more challenging (coldspots). Other tools exploit StackOverflow discussions to suggest context-specific code snippets and documentation [5], [25], [26], [27], [29], [31], [38], [41].

[5] https://www.codota.com/

VIII. Conclusions

In this paper, we introduced FOCUS, a context-aware collaborative-filtering system to assist developers in selecting suitable API function calls and usage patterns. To validate the performance of FOCUS, we conducted a thorough evaluation on different datasets consisting of GitHub and Maven open source projects. The evaluation was twofold. First, we examined whether the system is applicable to real-world settings by providing developers with useful recommendations as they are programming. Second, we compared FOCUS with a well-established baseline, i.e., PAM, with the aim of showcasing the superiority of our proposed approach. Our results show that FOCUS recommends API calls with high success rates and accuracy. Compared to PAM, FOCUS works both effectively and efficiently as it can produce more accurate recommendations in a shorter time. The main advantage of FOCUS 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. Lastly, FOCUS also scales well with large datasets by using the collaborative-filtering technique that helps sweep irrelevant items, thus improving efficiency. With these advantages, we believe that FOCUS is suitable for supporting developers in real-world settings. For future work, we plan to conduct a user study to thoroughly study the system's performance. Moreover, we will embed FOCUS directly into the Eclipse IDE.

Acknowledgment

The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223. Moreover, the authors would like to thank Claudio Di Sipio for his hard work on supporting the evaluation of FOCUS, and Morane Gruenpeter for her hard work on collecting the dataset from the Software Heritage archive.

References

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

[2] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju, "M3: A General Model for Code Analytics in Rascal," in *1st International Workshop on Software Analytics*. Piscataway: IEEE, 2015, pp. 25–28.

[3] R. P. L. Buse and W. Weimer, "Synthesizing API Usage Examples," in *34th International Conference on Software Engineering*. Piscataway: IEEE, 2012, pp. 782–792.

[4] A. Chen, "Context-Aware Collaborative Filtering System: Predicting the User's Preference in the Ubiquitous Computing Environment," in *First International Conference on Location- and Context-Awareness*. Berlin, Heidelberg: Springer, 2005, pp. 244–253.

[5] J. Cordeiro, B. Antunes, and P. Gomes, "Context-Based Recommendation to Support Problem Solving in Software Development," in *Third International Workshop on Recommendation Systems for Software Engineering*. Piscataway: IEEE, 2012, pp. 85–89.

[6] R. Di Cosmo and S. Zacchiroli, "Software Heritage: Why and How to Preserve Software Source Code," in *14th International Conference on Digital Preservation*, Kyoto, 2017, pp. 1–10.

[7] T. Di Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker, "Linked Open Data to Support Content-based Recommender Systems," in *8th International Conference on Semantic Systems*. New York: ACM, 2012, pp. 1–8.

[8] R. A. Fisher, "Confidence limits for a cross-product ratio," *Australian Journal of Statistics*, 1962.

[9] J. Fowkes and C. Sutton, "PAM: Probabilistic API Miner," https://github.com/mast-group/api-mining, last access 24.08.2018.

[10] ——, "Parameter-free Probabilistic API Mining Across GitHub," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York: ACM, 2016, pp. 254–265.

[11] X. Gu, H. Zhang, and S. Kim, "Deep Code Search," in *40th International Conference on Software Engineering*. New York: ACM, 2018, pp. 933–944.

[12] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API Learning," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York: ACM, 2016, pp. 631–642.

[13] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, 1979.

[14] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," in *27th International Conference on Software Engineering*. New York: ACM, 2005, pp. 117–125.

[15] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[16] R. Kohavi, "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection," in *14th International Joint Conference on Artificial Intelligence*. San Francisco: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[17] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?" in *37th International Conference on Software Engineering*. Piscataway: IEEE, 2015, pp. 880–890.

[18] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow," in *28th IEEE International Conference on Software Maintenance*. Piscataway: IEEE, 2012, pp. 25–34.

[19] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical Learning Approach for Mining API Usage Mappings for Code Migration," in *29th ACM/IEEE International Conference on Automated Software Engineering*. New York: ACM, 2014, pp. 457–468.

[20] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio, "CrossSim: Exploiting Mutual Relationships to Detect Similar OSS Projects," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2018, pp. 388–395.

[21] P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, "Knowledge-aware Recommender System for Software Development," in *Proceedings of the Workshop on Knowledge-aware and Conversational Recommender Systems 2018 co-located with 12th ACM RecSys, KaRS@RecSys 2018, Vancouver, Canada, October 7, 2018.*, 2018, pp. 16–22.

[22] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "crossminer/focus: Icse19-artifact-evaluation," Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2550379

[23] H. Niu, I. Keivanloo, and Y. Zou, "API Usage Pattern Recommendation for Software Development," *Journal of Systems and Software*, vol. 129, no. C, pp. 127–139, 2017.

[24] D. L. Parnas, "Information Distribution Aspects of Design Methodology," Departement of Computer Science, Carnegie Mellon University, Pittsburgh, Tech. Rep., 1971.

[25] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging Crowd Knowledge for Software Comprehension and Development," in *17th European Conference on Software Maintenance and Reengineering*. Washington: IEEE, 2013, pp. 57–66.

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

[27] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting Software Developers with a Holistic Recommender System," in *39th International Conference on Software Engineering*. Piscataway: IEEE, 2017, pp. 94–105.

[28] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis," in *38th International Conference on Software Engineering*. New York: ACM, 2016, pp. 357–367.

[29] M. Rahman, S. Yeasmin, and C. Roy, "Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions," in *Conference on Software Maintenance, Reengineering, and Reverse Engineering*. Piscataway: IEEE, 2014, pp. 194–203.

[30] V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York: ACM, 2014, pp. 419–428.

[31] P. C. Rigby and M. P. Robillard, "Discovering Essential Code Elements in Informal Documentation," in *35th International Conference on Software Engineering*. Piscataway: IEEE, 2013, pp. 832–841.

[32] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[33] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API Property Inference Techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[34] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining Multi-level API Usage Patterns," in *22nd International Conference on Software Analysis, Evolution, and Reengineering*. Piscataway: IEEE, 2015, pp. 23–32.

[35] M. A. Saied, H. Abdeen, O. Benomar, and H. Sahraoui, "Could We Infer Unordered API Usage Patterns Only Using the Library Source Code?" in *23rd International Conference on Program Comprehension*. Piscataway: IEEE, 2015, pp. 71–81.

[36] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based Collaborative Filtering Recommendation Algorithms," in *10th International Conference on World Wide Web*. New York: ACM, 2001, pp. 285–295.

[37] J. B. Schafer, D. Frankowski, J. L. Herlocker, and S. Sen, "Collaborative filtering recommender systems," in *The Adaptive Web, Methods and Strategies of Web Personalization*, 2007, pp. 291–324.

[38] W. Takuya and H. Masuhara, "A Spontaneous Code Recommendation Tool Based on Associative Search," in *3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. New York: ACM, 2011, pp. 17–20.

[39] C. Teyton, J. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 192–201.

[40] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web," in *23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington: IEEE, 2008, pp. 327–336.

[41] C. Treude and M. P. Robillard, "Augmenting API Documentation with Insights from Stack Overflow," in *38th International Conference on Software Engineering*. New York: ACM, 2016, pp. 392–403.

[42] G. Uddin and M. P. Robillard, "How API Documentation Fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[43] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining Succinct and High-coverage API Usage Patterns from Source Code," in *10th Working Conference on Mining Software Repositories*. Piscataway: IEEE, 2013, pp. 319–328.

[44] T.-T. Wong, "Performance Evaluation of Classification Algorithms by K-fold and Leave-one-out Cross Validation," *Pattern Recognition*, vol. 48, no. 9, pp. 2839–2846, 2015.

[45] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *23rd European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2009, pp. 318–343.