11-2018

Improving reusability of software libraries through usage pattern mining

Mohamed Aymen SAIED
University of Montreal

Ali OUNI
University of Quebec at Montreal

Houari A. SAHRAOUI
University of Montreal

Raula Gaikovina KULA
Osaka University

Katsuro INOUE
Osaka University

See next page for additional authors

Published in Journal of Systems and Software,
Vol 145, November 2018, Pages 164–179
DOI 10.1016/j.jss.2018.08.032

Improving reusability of software libraries through usage pattern mining

Mohamed Aymen Saied[*,a,e], Ali Ouni[b], Houari Sahraoui[a], Raula Gaikovina Kula[c], Katsuro Inoue[c], David Lo[d]

[a] DIRO, Université de Montréal, Montréal, Canada
[b] ETS Montreal, University of Quebec, Canada
[c] Graduate School of Information Science and Technology, Osaka University, Japan
[d] School of Information Systems, Singapore Management University, Singapore
[e] Engineering and Computer Science, Concordia University, Montreal, Canada

ARTICLE INFO

Keywords:
Software libraries
Software reuse
Clustering
Usage patterns

ABSTRACT

Modern software systems are increasingly dependent on third-party libraries. It is widely recognized that using mature and well-tested third-party libraries can improve developers' productivity, reduce time-to-market, and produce more reliable software. Today's open-source repositories provide a wide range of libraries that can be freely downloaded and used. However, as software libraries are documented separately but intended to be used together, developers are unlikely to fully take advantage of these reuse opportunities. In this paper, we present a novel approach to automatically identify third-party library usage patterns, i.e., collections of libraries that are commonly used together by developers. Our approach employs a hierarchical clustering technique to group together software libraries based on external client usage. To evaluate our approach, we mined a large set of over 6000 popular libraries from Maven Central Repository and investigated their usage by over 38,000 client systems from the Github repository. Our experiments show that our technique is able to detect the majority (77%) of highly consistent and cohesive library usage patterns across a considerable number of client systems.

1. Introduction

Third-party software libraries have become an integral part of modern software development. Today's software systems increasingly depend on external libraries, to reduce development time, and deliver reliable and quality software. Developers can take the benefit of freely reusing functionality provided by well-tested and mature third-party libraries and frameworks through their Application Programming Interfaces (APIs) (Robillard, 2009). Developers often reinvent the wheel and spend effort and time on implementing functionality, already provided by mature libraries. Automatically identifying existing library usage patterns, can help developers to overcome the complexity of writing their code from scratch, and avoid reinventing the wheel.

Much research efforts have been dedicated to the identification of library API usage patterns (Wang et al., 2013; Uddin et al., 2012; Zhong et al., 2009; Saied et al., 2015c; 2015a). The vast majority of existing works focus on the method level within a single library. Indeed, these approaches assume that the set of relevant libraries is already known to the developer, and it is only the methods in these libraries that are unknown. However, identifying relevant libraries and understand their usage trend is a hard and time-consuming activity.

Today's code repositories on the Internet provide an increasingly large number of reusable software libraries with a variety of functionalities. Automatically analyzing how software projects utilize these libraries, and understanding the extent and nature of software library reuse in practice is a challenging task for developers. Indeed, software developers can spend a considerable amount of time and effort to manually identify libraries that are useful and relevant for the implementation of their software. Worse yet, developers may even be unaware of the existence of these libraries, in particular when they are not popular. Developers tend to implement most of their features from scratch instead of reusing functionalities provided by third-party libraries as pointed out by several researchers (Uddin et al., 2012; Zhong et al., 2009; Duala-Ekoko and Robillard, 2011). Therefore, we believe that identifying patterns of libraries commonly used together, can help developers to discover and choose libraries that may be relevant for their projects' implementation.

In this paper, we propose a novel approach for mining Library Co-Usage Patterns, namely LibCUP. We define a usage pattern of libraries as a collection of libraries that are jointly used in client systems. Our

* Corresponding author at: Engineering and Computer Science, 1455 de Maisonneuve Blvd. W., Montral, PQ H3G 1M8, Canada.
E-mail addresses: m_saied@encs.concordia.ca (M.A. Saied), ali.ouni@etsmtl.ca (A. Ouni), sahraouh@iro.umontreal.ca (H. Sahraoui), raula-k@ist.osaka-u.ac.jp (R.G. Kula), inoue@ist.osaka-u.ac.jp (K. Inoue), davidlo@smu.edu.sg (D. Lo).

approach is based on the analysis of the joint versus separate use of the libraries. The pattern's libraries are distributed on different usage cohesion levels/layers. Each layer reflects the co-usage frequency between a set of libraries, while the distribution on the different levels demonstrates the graduation in the degree of co-usage frequency. Our approach adopts a variant of DBSCAN, a widely used density-based clustering algorithm, to detect candidate library usage patterns based on the analysis of their frequency and consistency of usage within a variety of client systems. Different client systems may use utility libraries (e.g., `JUnit`, `log4j`, etc.) as well as domain-specific libraries (e.g., `httpclient`, `groovy`, `spring-context`, etc.). Thus, the rationale behind the distribution on different usage cohesion levels of libraries in a pattern, is also to distinguish between the most specific libraries and the less specific ones. Moreover, our approach is intended to be used first to identify patterns of particular libraries that interest a developer. These libraries could then be fed to existing approaches (Uddin et al., 2012; Zhong et al., 2009; Saied et al., 2015c; 2015a) to recommend particular methods to be used in different contexts. Moreover, LibCUP provides a user-friendly visualization tool to assist developers in exploring the different library usage patterns.

We evaluate our approach on a large dataset of over 6000 popular libraries, collected from Maven Central repository[1] and investigated their usage from a wide range of over 38,000 client systems from Github repository[2], from different application domains. Furthermore, we evaluated the scalability of LibCUP as compared to LibRec (Thung et al., 2013), a state-of-the-art library recommendation technique based on association rule mining and collaborative filtering. We also performed a ten-fold cross validation to evaluate the generalizability of the identified usage patterns to potential new client systems. Our results show that across a considerable variability of client systems, the identified usage patterns by LibCUP remain more cohesive than those identified by LibRec. The main contributions of the paper can be summarized as follows:

1. We introduce a novel approach for mining multi-level usage patterns of libraries using an adapted hierarchical clustering technique. Our approach is supported by a user-friendly tool to visualize and navigate through the identified library usage patterns[3].
2. We mine a large dataset of over 6000 popular libraries from Maven repository and investigated their usage from a wide range of over 38,000 client systems from Github.
3. We evaluate the effectiveness of our approach in terms of cohesiveness and generalizability of the identified patterns. Results show that our approach was able to identify a larger number of usage patterns, on different usage cohesion levels.

The remainder of the paper is organized as follows. Section 2 motivates the usefulness of LibCUP with two real-world examples. Section 3 presents the related work. We detail our approach in Section 4. We present our experimental study to evaluate the proposed approach in Section 5, while providing discussions in Section 6. Finally, we conclude and suggest future work in Section 7.

2. Motivation and challenges

In this section, we present two real-world scenarios to motivate the usefulness of library co-usage patterns. In the first example, the goal is to find a set of libraries that allow to meet the requirements of a given software system. In the second example, the goal is to decide between two libraries with similar functionalities to be used in a software system. In this context, we assume that a library with more potential to be used with other related libraries is preferred. The related libraries are assumed to extend the features of the software system.

2.1. Learning-environment example

Let us consider a software development team responsible of the task of maintaining a Web portal for a growing private university with around 4000 undergraduate and graduate students. The university is planing to move from a simple Web portal to an advanced course management system to provide adequate service to their students and faculty members. As a first step, the development team decided to go through an exploratory phase, during which they developed a situational application to assess the turnout rate in the new learning environment. This application allows students and faculty to schedule activities related to courses and maintain deadlines related to projects. It should also allow real-time conversations between course or project participants.

Based on these requirements, developers found that their application requires some basic functionalities including a *scheduling* and an *emailing* service that have to be integrated. In this situation, developers can either implement the different features from scratch, or reuse features provided by existing libraries. In both cases, they may spend a considerable time and effort for either implementing the features or finding compatible and useful libraries to be integrated in the application.

The development team later finds out that they are required to use the `quartz` library to implement the scheduler. With this new constraint, the developers have to solve the following challenges:

- *What is the recommended emailing library that best complements the library* `quartz`*?* The selection should take into account assumed compatibility with the `quartz` library as well as the effort needed to integrate the library into the system.
- *More generally, what related libraries can be used to implement the remaining features of their software system?* The developers might be interested in related libraries that are commonly used by similar systems with the `quartz` library.

Addressing these two challenges could be a complex task for developers if done manually. Indeed, developers should check in open-source code repositories to find similar projects, and investigate their library usage. Manually finding libraries that are commonly used together in a particular scenario and understanding the current usage practice for a particular library is unlikely to be effective.

2.2. Web application frontend example

We now consider another scenario with Aaron, a freelance programmer, who seeks to implement an inventory management web application. Aaron decided to develop his web applications in an industrial setting, where the back-end is implemented in Java and the front-end is implemented in a Java/XML based framework. For the user interfaces, several libraries can be used; the most popular ones are `primefaces` the UI component library for Java Server Faces, and `gwt-user` of the Google Web Toolkit.

Aaron has to decide which library to use: `primefaces` or `gwt-user`. In other words:

- *Which library is the best option in terms of future extension of the software system's functionalities?* Aaron prefers libraries that are usually used with many other libraries, which offer a large variety of functionalities. This provides a high potential of extensions of his software system.

In both examples, we consider that mining patterns of libraries used jointly by many client systems may provide insights to make the best decisions.

[1] http://mvnrepository.com.
[2] www.github.com.
[3] https://saiedmoh.github.io/LibCUP/.

2.3. Challenges: mining library usage

In this work, we mine the 'wisdom of the crowd' to discover usage patterns of software libraries. Studying the current library usage within similar systems may provide hints on compatibility and relevance between existing libraries. We assume that libraries that are commonly used together are unlikely to have compatibility and integration issues.

The goal is to discover which sets of libraries are commonly used together by similar systems. To this end, our approach is designed to find multiple layers, i.e., levels, of relevant libraries according to their usage frequency. For effective reuse, developers can go through the different levels inside the usage patterns to discover relationships, with different strengths, between the collection of related libraries.

For the first motivating example, we use the usage patterns to discover that `commons-email` library , which is a popular emailing library, complements the `quartz` library. Furthermore, by using the multi-layers structure of our patterns, developers can then find related libraries that would complement, at different degrees, both `quartz` and `commons-email`.

For the second motivating example, we found that `gwt-user` library is part of a usage pattern with many other related libraries including `gwt-dev` , `gwt- servlet` , `gwt-incubator` , and `gin` . This collection of libraries covers different functionalities such as browser support, widgets, optimization, data binding, and remote communication. All these features are opportunities for future extensions, and we are confident that they can be integrated together as demonstrated by the client systems that already used them. Conversely, Aaron found that, although the `primefaces` library might be useful for his system, it is not widely used with other libraries and, then, does not offer a sufficient guarantee of future integration with other libraries.

These two examples show that the task of identifying library usage patterns becomes more and more complex, especially with the exponentially growing number of libraries available in the Internet. This motivates our proposal of automatically identify library usage patterns to assist developers in reusing and integrating libraries and, then, increase their productivity.

3. Background and related work

3.1. Background

In this paper, we evaluated our technique LibCUP as compared to LibRec, a state-of-the-art library recommendation technique. Thus we present in this subsection an overview of LibRec. LibRec combines association rule mining and collaborative filtering. The association rule mining component recommends libraries based on a set of inferred library usage patterns. The collaborative filtering component recommends libraries based on their usage on other similar client projects (Thung et al., 2013).

Association rule mining: An association rule is an "if/then" rule that can be written as: $X \Rightarrow Y$ where X is the precondition event of the rule and Y is the postcondition event of the rule. The precondition is a statement that must be satisfied for the rule to be applied, whereas the postcondition is the result if the precondition is met. For LibRec the rule's events are library usage in client systems. The previous rule could be read, if Lib X is used than Lib Y should be used. When the association rule is frequent and has a high likelihood to be correct, the rule could be seen as a pattern.

Collaborative filtering: Collaborative filtering is an automatic technique to make predictions about an entity based on information collected about other similar entities. A basic method to perform collaborative filtering is by finding the nearest neighbors of the target entity. A target entity is compared with all other entities and a list of most similar entities based on a distance metric is produced. The similarities among the entities are used as a basis for making predictions about the entity. In LibRec, an entity is a client project, and the prediction task is the prediction of libraries that are useful for the project. To measure the similarity of two projects, LibRec is based on the set of libraries that are used in common between the two projects.

3.2. Related work

Recently, different aspects around library usage have gained considerable attention. Existing contributions can be organized into different categories according to the purpose of their proposed techniques: (*i*) third-party library usage, (*ii*) code completion, (*iii*) library API usage example, (*iv*) API usage visualization, (*v*) exploration of API usage obstacles, and (*vi*) mining API usage patterns.

Third-party usage at the library level. Several works were interested in the third-party usage at the library level, but for different purposes such as library refactoring (Penta et al., 2002), library recommendation (Thung et al., 2013; Ouni et al., 2017) and library miniaturization (Antoniol and Penta, 2003; Antoniol et al., 2003). The most related to our work is the one by Thung et al. (2013). As mentioned previously the authors proposed a hybrid approach that combines association rule mining and collaborative filtering, to recommend libraries based on their usage on similar client systems. This approach is most related to our in the sense that it is also based on library usage through their client systems dependency. However, both approaches have a different purpose. Thung et al. (2013) are interested in the similarity between client systems to recommend libraries based on the dependency of similar clients, whereas in our case we are interested in the overall libraries usage to discover multi-layers library patterns.

Code completion. Enhancing current completion systems to work more effectively with large APIs has been investigated in Nguyen et al. (2012), Bruch et al. (2009), Hou and Pletcher (2011), McMillan et al. (2011) and Endrikat et al. (2014). This body of work makes use of a database of API usage recommendation, type hierarchy, context filtering and API methods functional roles for improving the performance of API method call completion. Recently, Asaduzzaman et al. (2014) proposed a context sensitive code completion technique that uses, in addition to the aforementioned information, the context of the method call.

API usage example. A similar body of work is interested in example recommendation of API usage (Wang et al., 2011; Duala-Ekoko and Robillard, 2011; Buse and Weimer, 2012; Montandon et al., 2013). Existing contributions can be organized into two groups: IDE-based recommendation systems and JavaDoc-based recommendation systems. These contributions tried to instrument API documentation with usage examples based on a static slicing, clustering and pattern abstraction.

API usage visualization. Other contributions tried to enhance understanding API usage through explorative and interactive methods (Moritz et al., 2013; Kula et al., 2014; Parnin et al., 2012; De Roover et al., 2013; Saied et al., 2015d). This body of work described multi-dimensional exploration of API usage. The explored dimensions are related to the hierarchical organization of projects and APIs, metrics of API usage and API domains. A visualization strategy would necessarily enrich the usefulness of our approach.

Exploration of API usage obstacles. From another perspective, the work in Hou and Li (2011), Wang and Godfrey (2013) and Saied et al. (2015e) explored API usage obstacles through analyzing developers questions in Q&A website. This allows API designers to understand the problems faced while using their API, and to make corresponding improvements.

Mining API usage patterns. Other contributions related to ours are those interested in mining API usage patterns (Wang et al., 2013; Uddin et al.,

2012; Zhong et al., 2009; Li and Zhou, 2005; Saied and Sahraoui, 2016). These contributions adopted different categories of API usage patterns, different techniques for inferring patterns and different ways to assess patterns correctness and usefulness. The most prominent categories are temporal (Uddin et al., 2012; Huppe et al., 2017), unordered (Li and Zhou, 2005; Saied et al., 2015b) and sequential (Wang et al., 2013; Zhong et al., 2009) usage patterns. These categories were assessed through consistency, coverage and succinctness of the mined usage patterns. Zhong et al. (2009) developed the MAPO tool for mining sequential API usage patterns. MAPO clusters frequent API method call sequences extracted from code snippets, based on the number of called API methods and textual similarity of class and method names between different snippets. Our work considers a different level of granularity from these existing works. Previous approaches infer API usage patterns at the API element level (i.e. methods call). In our case, we are interested in inferring usage pattern at the granularity of the entire library. Past approaches assume that the developer already selected the relevant library and he only needs to learn how to use the methods in this library. Our work does not make this assumption, and thus complements the existing studies. Indeed, our approach could be used as an early phase to infer sets of libraries consistently co-used together. Then existing approaches could be applied to learn how to use particular methods within the patterns' libraries.

4. The proposed approach

In this section, we present our approach, LibCUP, for mining library usage patterns. Before detailing the used algorithm, we provide a brief prerequisite, an overview of our approach and describe our visualization technique to explore the identified library usage patterns.

4.1. Prerequisites

Before delving into the details of our approach, we first provide the basic concepts and terminology underlying the proposed approach in this paper: library, client system, library dependency.

- *Software library:* A software library is a software system that provides its functionalities through a publicly defined API. In general, most of software libraries are hosted in software ecosystems that offer specialized dependency management systems such as Maven[4]and Nuget[5].
- *Client system:* We consider a client system as any software system that is hosted in a repository and has at least one external dependency with existing libraries. A library could be also considered as a client system itself if it has dependencies with other libraries. Note that not all libraries could be hosted in such ecosystems.

4.2. Approach overview

Our approach takes as input a set of popular libraries, and a wide variety of their client systems extracted from existing open-source repositories. The output is a set of library usage patterns, each pattern is a collection of libraries , organized within different layers according to their co-usage frequency.

We define a *library co-usage pattern* (LCUP) as a collection of libraries that are commonly used together. A LCUP represents an exclusive subset of libraries, distributed on different usage cohesion layers. A usage cohesion layer reflects the co-usage frequency between a set of libraries.

Indeed, similar client systems may share some domain specific libraries, but they may at the same time share some utility libraries which are more commonly used by a large number of systems. For this reason, we seek a technique that can capture co-usage relationships between libraries at different levels.

Our approach proceeds as follows. First, the input dataset is analyzed to identify the different client systems depending on each library. Then, the dependency information is encoded using usage vectors. Indeed, each library in the dataset is characterized with a usage vector which encodes information about (1) their client systems and (2) the other systems in the dataset that are not using it. Finally, we use hierarchical clustering technique based on DBSCAN to group the libraries that are most frequently co-used together by clients. All libraries that have no consistent usage through the client systems are isolated and considered as noisy data.

4.3. Clustering algorithm

Our clustering is based on the algorithm DBSCAN (Ester et al., 1996). DBSCAN is a density-based algorithm, i.e., the clusters are formed by recognizing dense regions of points in the search space. The main idea behind it, is that each point to be clustered must have at least a minimum number of points in its neighbourhood. This property of DBSCAN allows the clustering algorithm to filter out all points that are not located in a dense region of points in the search space. In other words, the algorithm clusters only relevant points and leaves out noisy points.

This specific property explains our choice of the clustering algorithm DBSCAN to detect usage patterns of libraries. Indeed, not all libraries of the dependency dataset are to be clustered because some are simply not co-used with specific subsets of the libraries, while others are co-used with almost all the subsets of libraries.

In our approach, each library is represented as a usage vector that has constant length l. The vector length is the number of all client programs which use the libraries in the dataset. Hence, we carry out the library usage analysis on a specific set of client programs in a particular snapshot in time. Fig. 1 shows that the considered dataset represents 8 client systems depending on 8 third-party libraries. For an external library, Lib_x, an entry of 1 (or 0) in the ith position of its usage vector, denotes that the client system corresponding to this position depends (or does not depend) on the considered library. Hence, summing the entries in the library's vector represents the number of its client program in the dataset. For instance, in Fig. 1, the usage vector of *Lib1* shows that the four client systems *C1, C2, C3* and *C6* depend on this library. We can also see that these systems depend on other libraries including *Lib2, Lib3* but none of them depends on *Lib5*.

DBSCAN constructs clusters of libraries by grouping libraries that are close to each other, thus forming a dense region (i.e., similar libraries) in terms of their co-usage frequency. For this purpose, we define the Usage Similarity, *USim* in Eq. (1), between two libraries Lib_i and Lib_j, using the Jaccard similarity coefficient with regards to the client programs, Cl_sys, of Lib_i and Lib_j.

The rationale behind this is that two libraries are close to each other (short distance) if they share a large subset of common client systems.

$$USim(Lib_i, Lib_j) = \frac{|Cl\_sys(Lib_i) \cap Cl\_sys(Lib_j)|}{|Cl\_sys(Lib_i) \cup Cl\_sys(Lib_j)|} \quad (1)$$

where $Cl\_sys(Lib)$ is the set of client programs depending on the library *Lib*. For example, the *USim* between the libraries *Lib1* and *Lib6* in Fig. 1 is $\frac{2}{4}$ since these libraries have in total 4 client programs, and 2 of them are common for *Lib1* and *Lib6*. The distance between the points in the search space corresponding to two libraries Lib_i and Lib_j is then computed as $Dist = 1 - USim(Lib_i, Lib_j)$.

DBSCAN requires two parameters to perform the clustering. The first parameter is the minimum number of points in a cluster, *minP*. We set this parameter at 2, so that a usage pattern must include at least two

[4] http://search.maven.org.
[5] https://www.nuget.org.

```
1:  ϵ-DBSCAN(DataSet, maxEpsilon, MinNbPts, epsilonStep){
2:  epsilon <– 0
3:  while epsilon < maxEpsilon do
4:      DBSCAN(DataSet, maxEpsilon, MinNbPts, epsilonStep)
5:      clusters <– DBSCAN.clusters
6:      noisyPoints <– DBSCAN.noisyPoints
7:      compositePoints <– constructPoints(clusters)
8:      Dataset <– noisyPoints ∪ compositePoints
9:      epsilon <– epsilon + epsilonStep
10: end while
11: }
12: constructPoints(clusters){
13: for each C in clusters do
14:     compositePoints <– ∪(all points of C)
15: end for
16: }
```

Algorithm 1. ϵ-DBSCAN: Hierarchical DBSCAN algorithm.

libraries of the studied dataset. The second parameter is, *epsilon*, the maximum distance within which two points can be considered as neighbor, each to other. In other words, *epsilon* value controls the minimal density that a clustered region can have. The shorter is the distance between libraries within a cluster the more dense is the cluster.

However, it is insufficient merely to apply a generic algorithm like DBSCAN 'out of the box'; we need to define problem-specific clustering to provide usage patterns distributed into different levels of usage cohesion.

In the next subsection, we describe our adaptation of the standard DBSCAN to support hierarchical clustering based on different values for *epsilon* to identify different usage cohesion level in the inferred patterns.

4.4. Multi-layer clustering

We have introduced a variant of DBSCAN (which we call ϵ-DBSCAN) specifically for the library usage patterns identification problem. In DBSCAN, the value of the *epsilon* parameter influences greatly the resulting clusters. A value of 0 for *epsilon*, means that each cluster must contain only libraries that are completely similar (i.e., distance among libraries belonging to the same cluster must be 0). Our idea is to "relax" the *epsilon* parameter that controls the constraints on the requested density within clusters.

On the one hand, if we set *epsilon* at fixed small value, e.g., $epsilon = 0$, this will produce patterns that are very dense. Consequently, the resulted usage patterns will include only libraries that exhibit a high co-usage score. On the other hand, an increase of the *epsilon* value, e.g., $epsilon = 0.3$, will result in an additional external layer of patterns that exhibit less co-usage score. Therefore, we iteratively apply the standard DBSCAN at different levels of *epsilon* in order to have library usage patterns organised as multi-layers, each representing a particular co-usage score.

As a result, our ϵ-DBSCAN, builds the clusters incrementally by relaxing the epsilon parameter, step by step. Algorithm 1 shows the pseudo-code of our incremental clustering technique, ϵ-DBSCAN. First, ϵ-DBSCAN takes as input a dataset containing all the libraries and their client systems within a specific format, then it clusters them using the standard DBSCAN algorithm with epsilon value of 0. This step results in clusters of libraries that are always used together, as well as multiple noisy points left out. For each produced cluster, we aggregate the usage vectors of its libraries using the logical disjunction in one usage vector. Then, a new dataset is formed which includes the aggregated usage vectors and the usage vectors of noisy libraries from the previous run. This dataset is then fed back to the DBSCAN algorithm for clustering, but with a slightly higher value of epsilon, i.e., $epsilon = epsilon + epsilonStep$. This procedure is repeated in each step until reaching *maxEpsilon* a maximum value for epsilon, given as a parameter.

For example, Fig. 2 shows the incremental clustering of the libraries in Fig. 1 using ϵ-DBSCAN. In this example, the initial dataset contains 8 libraries, *Lib1, ... , Lib8*. The *epsilon* parameter is incremented in each step by $epsilonStep = 0.25$ with the epsilon maximum value set to $maxEpsilon = 0.55$. The choice of the parameters' values is for the sake of the illustrative example. As shown in Fig. 2a, the first step produces two clusters at $epsilon = 0$. The two clusters include respectively (*Lib1, Lib2, Lib3*) and (*Lib4, Lib5*). These libraries are clustered at the most dense level since, in each cluster, these libraries were co-used together frequently (by the exact same clients). The second step is performed with $epsilon = 0.25$ as illustrated in Fig. 2b. For this step, there is no change in the dataset since the distances are larger than the current *epsilon* value. Finally at $epsilon = 0.5$, as illustrated in Fig. 2c a new cluster involving 2 density levels is generated. This cluster includes *Lib7* in addition to (*Lib1, Lib2, Lib3*) since they share 2 out of the 4 common client systems.

We can notice that *Lib6* is a rarely used library and *Lib8* is a utility library used with almost all the considered client systems, showing no particular usage trend. Thus at the last iteration of ϵ-DBSCAN illustrated in Fig. 2d, the libraries *Lib6* and *Lib8* are left out as noisy points since their distance from the clustered libraries is larger than the maximum epsilon value, which is 0.55.

5. Empirical study

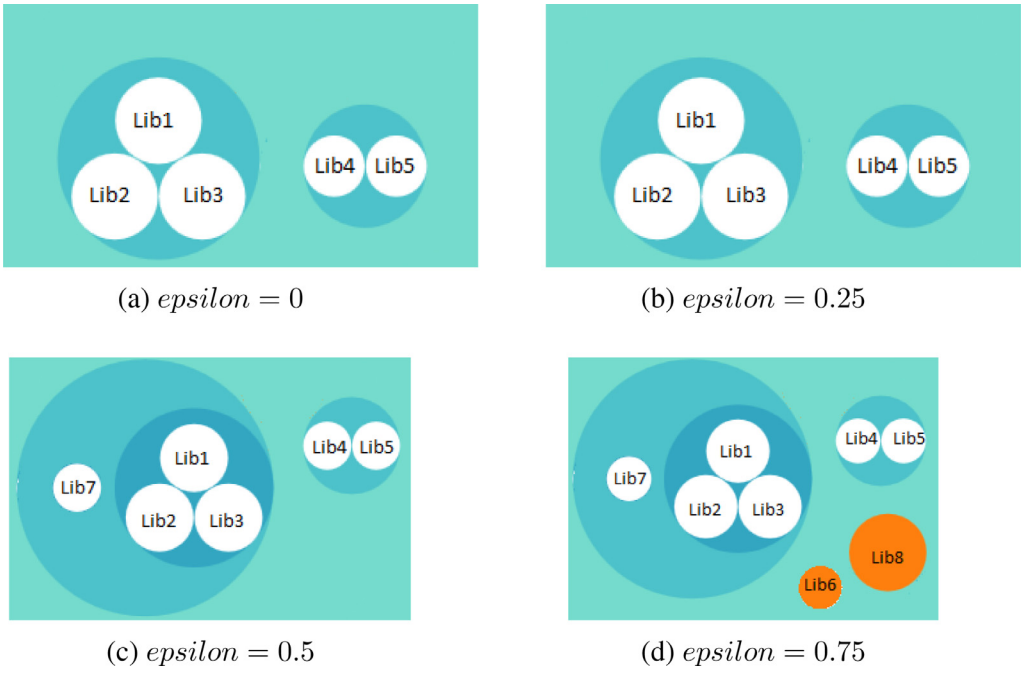
In this section, we present the results of our evaluation of the proposed approach, LibCUP. Our study aims at assessing whether LibCUP can detect *usage patterns of libraries* that are (*i*) *cohesive* enough to provide valuable information to discover relevant libraries, and (*ii*) *generalizable* for new client systems. We also compare the results of our technique LibCUP to the available state-of-the-art approach, LibRec (Thung et al., 2013). LibRec combines association rule mining and collaborative filtering to recommend libraries based on their client usage. For each experiment in this section, we present the research questions to answer, the research method to address them, followed by the obtained results.

All the material used to run our three experiments is publicly available in a comprehensive replication package[6].

[6] https://saiedmoh.github.io/LibCUP/.

	C1	C2	C3	C4	C5	C6	C7	C8
Lib1	1	1	1	0	0	1	0	0
Lib2	1	1	1	0	0	1	0	0
Lib3	1	1	1	0	0	1	0	0
Lib4	0	0	0	1	1	0	0	0
Lib5	0	0	0	1	1	0	0	0
Lib6	0	0	0	0	0	0	1	0
Lib7	0	0	1	0	0	1	0	0
Lib8	1	1	0	1	1	0	1	1

Fig. 1. The usage vector representing the dependency between eight client programs and eight libraries.



(a) $epsilon = 0$ (b) $epsilon = 0.25$

(c) $epsilon = 0.5$ (d) $epsilon = 0.75$

Fig. 2. Resulting clusters of applying the incremental algorithm ϵ-DBSCAN to the library dataset presented in Fig. 1.

5.1. Data collection

To evaluate the feasibility of our approach on real-world scenarios, we carried out our empirical study on a large dataset of Open Source Software (OSS) projects. As we described earlier, our study is based on widely used libraries collected from the popular library repository Maven and a large set of client systems collected from Github repository. Since Github is the host of varying projects, we performed the following filtering on the dataset, to limit the number of considered Github projects:

- *Commit size*: We only included java projects that had more than 1000 commits. To remove toy projects that exist on GitHub, we assume that real-world projects have more than 1000 commits over time.
- *Forks*: We only include projects that are unique and not forks of other projects.
- *Maven dependent project*: We only included projects that employ the maven build process (use `pom.xml` configuration file).

Our data collection consists of crawling both Github and Maven repositories. In particular, we track dependencies between a Maven libraries and each unique system within each project in Github (i.e., a project may contain multiple systems). Every project includes one or multiple Project Object Model files (i.e., pom.xml) that describe the project's configuration meta-data including its compile and run time library dependencies. We implemented this extraction method in our in-house tool called PomWalker[7] which is publicly available to the

[7] https://github.com/raux/PomWalker.

Table 1
Dataset used in the experiment.

	Dataset
Snapshot Date	15th January 2015
# of github systems	38,000
# of unique dependent libraries	6638

research community. Our method is similar to other works on software libraries (Kula et al., 2018; Raemaekers et al., 2014; 2012) which use the same schema of having (sub)systems in the repository depending on Maven libraries as managed through POM dependency management.

Note that for all data, we first downloaded an offline copy of the original software projects (the source code) from Github and the libraries (the jar files) from Maven before extraction. Thereafter, for each library, we selected the latest release. In the beginning we started with 40,936 dependent libraries. However, to remove noise, we filtered out libraries having less than 50 identifiers based on methods, attributes, and classes. This process removed libraries that we assume very small or partial copies of their original libraries and thus are not relevant and will not be a useful recommendation for other systems. As described in Table 1, our dataset resulted in 6638 Maven libraries extracted from unique 38,000 client systems from GitHub. Other filtering criteria and threshold values could have led to a different dataset. However, we believe that our assumption is a clear underestimation, which reduces the chance of having toy projects and very small libraries.

The dataset is a snapshot of the projects procured as of 15th January 2015. Our dataset is very diversified as it includes a multitude of libraries and software systems from different application domains and different sizes. Overall, the average number of used libraries per system is 10.56, the median is 6.

5.2. Sensitivity analysis

As a first experiment, we evaluated the sensitivity of the pattern's quality, identified by LibCUP, with respect to different settings including the dataset size and *maxEpsilon* values. We aim at addressing the following research question.

RQ1. *What is the impact of various experimental settings on the pattern's quality?*

5.2.1. Analysis method

To address (RQ1), we need to evaluate whether the detected patterns are cohesive enough to exhibit informative co-usage relationships between specific libraries. Hence, we use a cohesion metric namely, Pattern Usage Cohesion metric (PUC), to capture the cohesion of the identified patterns.

PUC is inspired from Perepletchikov et al. (2007) and was originally used to assess the usage cohesion of service interfaces. It evaluates the co-usage uniformity of an ensemble of entities, which corresponds, in our context, to a set of libraries in the form of a library usage pattern. PUC values are in the range [0,1]. The larger the value of PUC is, the better the usage cohesion, i.e., a usage pattern has an ideal usage cohesion (PUC = 1) if all the library patterns are always used together. Let p be a library usage pattern, then its PUC is defined as follows:

$$PUC(p) = \frac{\sum_{cp} ratio\_used\_Libs(p, cp)}{|C(p)|} \in [0, 1] \quad (2)$$

where *cp* denotes a client system of the pattern *p, ratio_used_Libs(p, cp)* is the ratio of libraries that belong to the pattern p and that are used by the client system *cp*, and $C(p)$ is the set of all client systems of all libraries in p.

To answer our first research question (RQ1), we perform two studies.

- **Study 1.A.** We apply LibCUP to our collected dataset described in Section 5.1. Then, we investigate the impact of different *maxEpsilon* values on the PUC results of the detected patterns.
- **Study 1.B.** We investigate the scalability of our technique. We fix the *maxEpsilon* value and we run LibCUP several times while varying the dataset size to observe the patterns cohesion and the time efficiency.

5.2.2. Results for RQ1

The obtained results are as follows.

Study 1.A: Sensitivity to maxEpsilon parameter. Figs. 3, 6, and 7 report the effect of different *maxEpsilon* values. Our experiments show that the *maxEpsilon* parameter influences different characteristics of the inferred patterns including the pattern usage cohesion, the number of inferred patterns, and the pattern size. Fig. 3 shows that the average PUC ranges from 1 to 0.5, while varying the *maxEpsilon* in the range [0,0.95]. We notice that even when the *maxEpsilon* reaches high values (0.95), the inferred patterns maintain cohesion values greater than 50.

When *maxEpsilon* is set to 1 the pattern cohesion drops down to 0. This is because in the last step all the libraries are clustered into one usage pattern as depicted in Fig. 6. Moreover, we can clearly see from this figure that the number of inferred patterns increases to reach a peak of 1061 when *maxEpsilon* is set to 0.80.

To get a more qualitative sense, we illustrate , in Fig. 4, an example of patterns inferred using LibCUP. The example consists of a pattern having in his core layer some libraries of the Spring framework such as `spring-context`, `spring-beans`, and `spring-orm` . Then, in the second layer, LibCUP identified some libraries of the Hibernate framework such as the `hibernate- entitymanager` and `hibernate-annotations`, while in the third layer, LibCUP identified some json libraries such us `jackson- databind`. The pattern continues growing until adding some utility libraries of logging and testing such as `SLF4J` API and `JUnit` API. Thereafter, some other libraries for basic utilities are added to the pattern including `commons-pool`, `commons-collections`, `commons-lang`, and `commons-io`. Looking at the dataset, we observe that these libraries have been co-used in a set of hundreds of client systems.

For a more detailed analysis, we illustrate in Fig. 5 the evolution of the considered pattern through the different clustering iteration while relaxing *maxEpsilon*. We observe that:

- Before the peak of *maxEpsilon* = 0.80, some of the existing patterns are enriched with new libraries to add new external layers to the original usage pattern such as `hibernate-entitymanager`, `hibernate-annotations` and `jackson-databind`. Moreover, we noticed that some new patterns are enriched with libraries that were considered as noisy such as `commons-lang` and `commons-io`. This could be justified by the fact that our DBSCAN adaptation tolerates less density within clusters when *maxEpsilon* increases. We observe that this has an effect to increase the global number of inferred patterns.
- After the peak, some of the existing patterns are merged without losing their internal structure as can be seen in Fig. 5. This result, in turn has an effect to reduce the overall number of inferred patterns.

To get more qualitative sense of the obtained results, we noticed that for the low values of *maxEpsilon* and up to intermediate values (i.e., 0.5, 0.6), the inferred patterns tend to mainly cover domain specific libraries (e.g., program analyzers `jdepend` , graphics manipulation `batik` , etc.). Those patterns are characterized with an average number of client systems that do not exceed 50 clients per pattern. The more the *maxEpsilon* parameter is relaxed, the more the patterns are enriched with other libraries. Starting with specific libraries, the patterns reach a step in which they become enriched with utility or more generic libraries such as `JUnit` and `log4j` . For sake of simplicity, we do not present in Fig. 7 the last step where all the libraries are clustered into

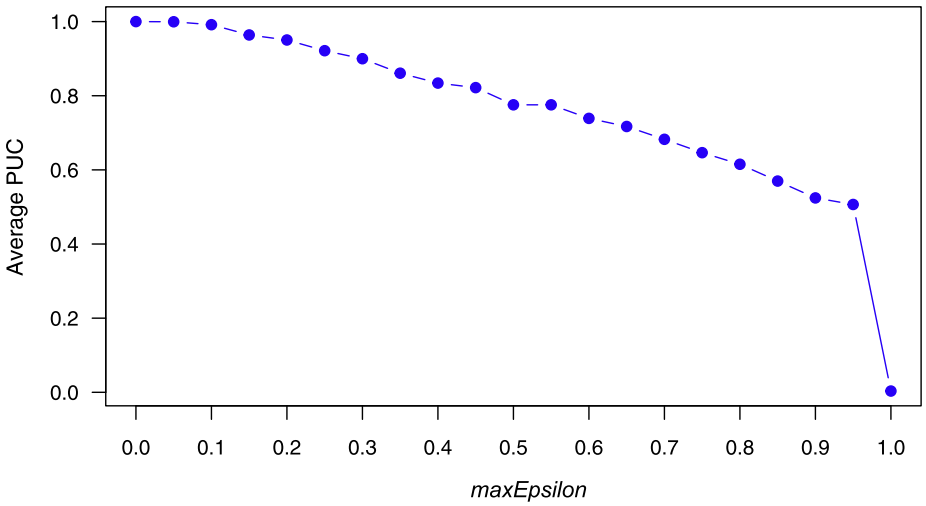


Fig. 3. Effect of varying *maxEpsilon* parameter on the average cohesion of the identified patterns.

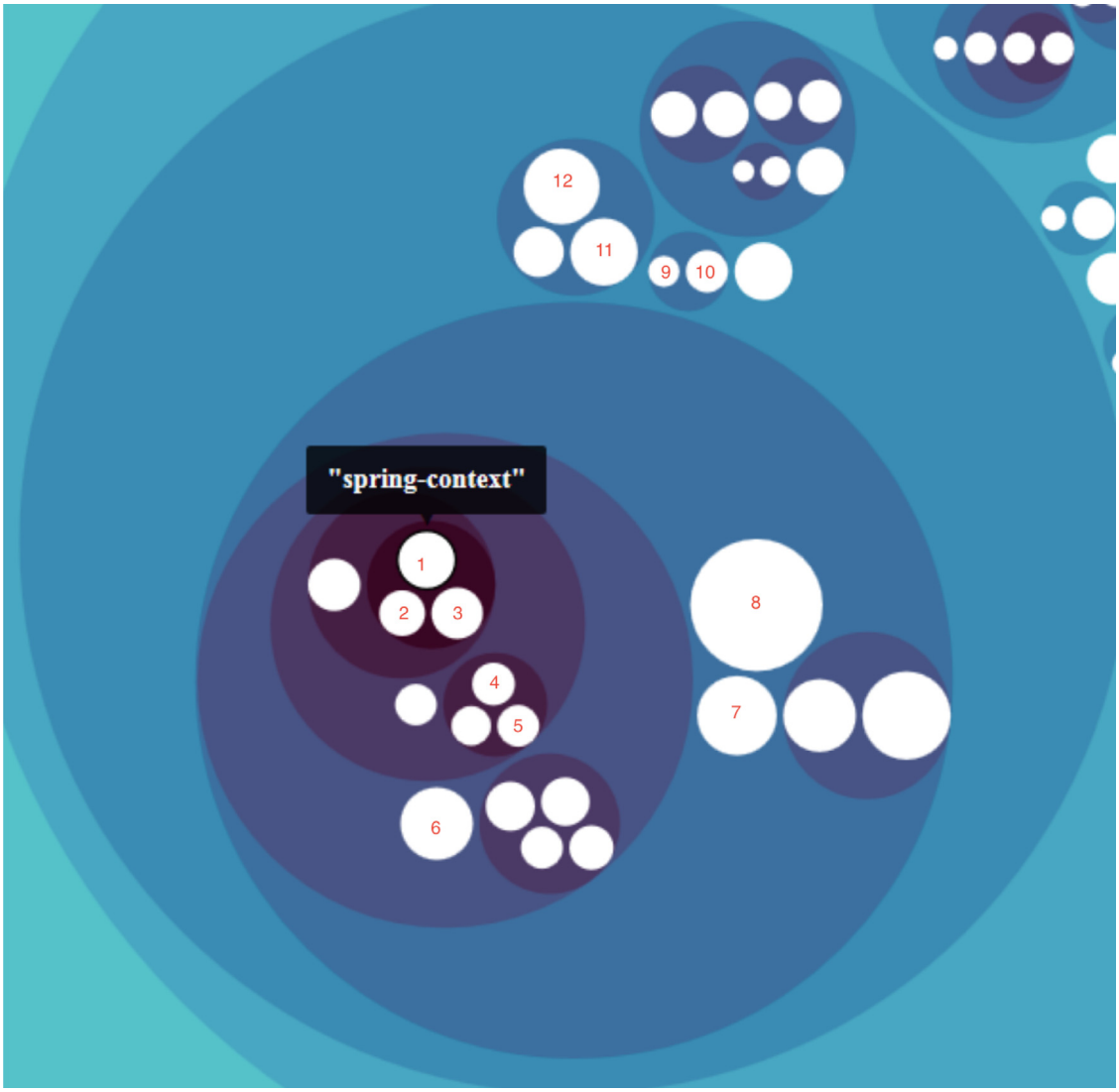


Fig. 4. Example of pattern visualization. The libraries index are: 1:`spring-context`, 2:`spring-beans`, 3:`spring-orm`, 4:`hibernate-entitymanager`, 5:`hibernate-annotations`, 6:`jackson-databind`, 7:`SLF4J`, 8:`JUnit`, 9:`commons-pool`, 10:`commons-collections`, 11:`commons-lang`, 12:`commons-io`.

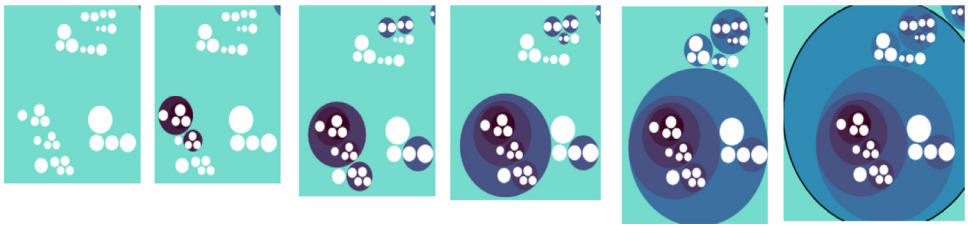


Fig. 5. Evolution of a pattern while relaxing maxEpsilon.

one single usage pattern with a larger number of client systems.

Study 1.B: Sensitivity to the dataset size. To carry out this experiment, we set the *maxEpsilon* value to 0.5. This is a proactive choice to ensure that libraries appearing in the same pattern are used more frequently together than separately. Thereafter, we run LibCUP with different dataset sizes. In each run, we augmented the previously used dataset with 1000 libraries, and we observed the average cohesion of patterns as well as the execution time taken to infer them. All experiments were carried out on a computer with an Intel core i7-4770 CPU 3.40 GHz, with 32 GB RAM.

Fig. 8 depicts the obtained results for this experiment. We noticed from the figure, that the shape of the graph is consistent for the different dataset size. The PUC score slightly increases from 0.79 to stabilize at 0.82 for the last three runs. In more details, we found that there is an increase in terms of the number of inferred patterns from 62 in the first run with an average size of 3 libraries per pattern, to reach the bar of 500 patterns at the last run with an average size of 5.5 libraries per pattern. These results confirm that when considering more libraries, LibCUP is able to enrich the inferred patterns with new libraries while detecting new patterns.

Our algorithm executes DBSCAN multiple times; thus we have the same order of complexity of the original DBSCAN. The nearby neighbors' identification step mainly influences the time complexity of DBSCAN. DBSCAN have an overall average runtime complexity of $O(n \log n)$ and the worst case run time complexity is $O(n^2)$ (Schubert et al., 2017). As a proof of concept, we are not using an indexing structure for the identification of nearby neighbors, and we are using distance matrices of size $(n^2 - n)/2$ thus we fall in the worst case quadratic runtime and memory complexity. More details on the complexity of DBSCAN could be found in Schubert et al. (2017). To get a more quantitative sense, we assess the scalability of our approach with respect to time efficiency.

Fig. 9 depicts the influence of the dataset size on the execution time. As it can be seen on the figure, the execution time of LibCUP is sensitive to the dataset size, as expected. At the first run, LibCUP took less than 7 min to mine a set of 1000 libraries, while reaching 159 min of execution time to mine the large set of 6000 libraries with their 38,000 client systems. However, it is worth saying that even with 159 min of execution time, LibCUP can be considered time efficient, since the inference process is done off-line once, then the identified patterns can be easily explored using our interactive visualization tool.

In summary, the obtained PUC results of the identified usage patterns provide evidence that LibCUP exhibits consistent cohesion using our adopted ϵ-DBSCAN technique. Using a default *maxEpsilon* = 0.5, we found that at least 50%, and up to 100%, of the patterns' libraries are co-used together with high PUC. Moreover, our technique is stable and time efficient when varying the size of the mined library set.

5.3. Evaluation of patterns cohesion

As a second experiment, we conduct a comparative study to evaluate the cohesiveness of the identified library usage patterns against the state-of-the-art approach LibRec (Thung et al., 2013). To the best of our knowledge, LibRec (Thung et al., 2013) is the only existing approach that has addressed this problem. We aim at addressing the following research question.

RQ2. *To which extent are the identified library usage patterns cohesive as compared to those inferred with LibRec?*

5.3.1. Analysis method

To address our second research question (RQ2) we conducted a comparative evaluation of our approach with LibRec in order to better position our approach and characterize the obtained results.

To infer usage patterns, LibRec is based on mining association rules obtained from closed itemsets and generators using the Zart algorithm (Szathmary et al., 2006). We applied both LibCUP and LibRec to all the selected libraries of our dataset (cf. Section 5.1). Then, we compare the identified usage patterns of both approaches in terms of PUC. More specifically, we compare the average PUC values for all detected

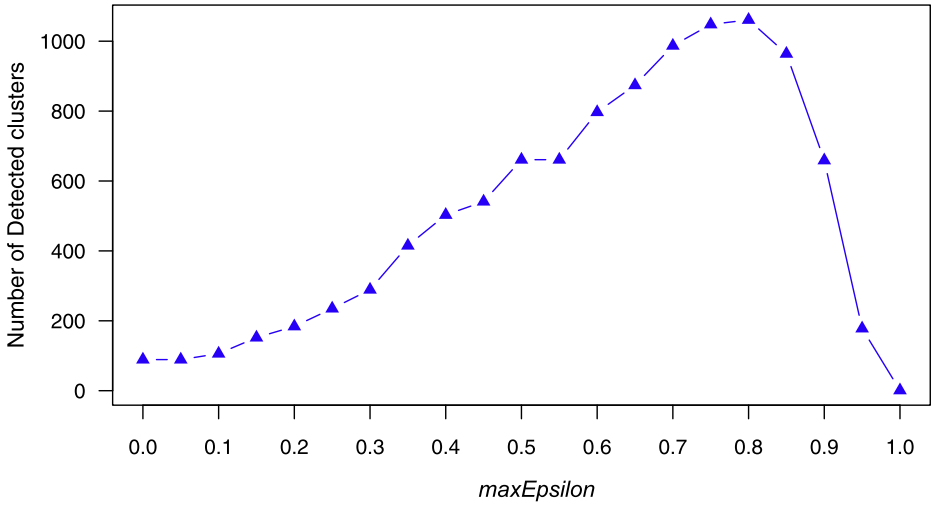


Fig. 6. Effect of varying *maxEpsilon* parameter on the number of identified patterns.

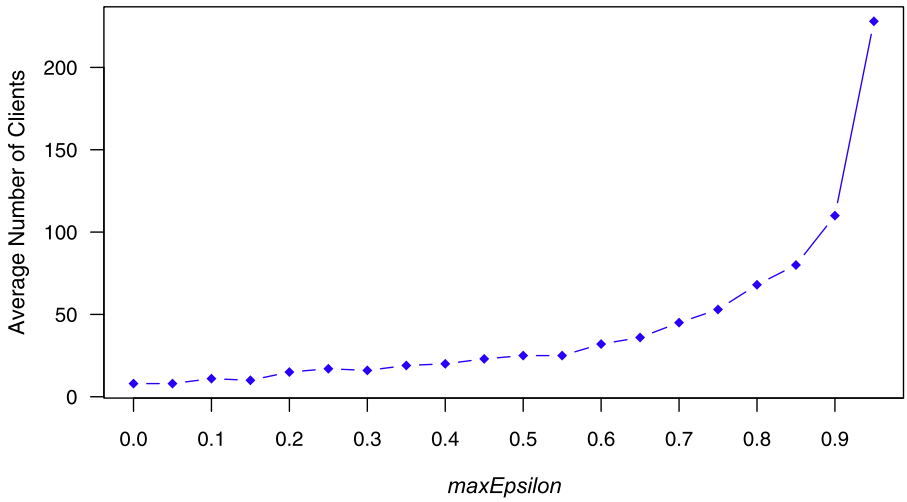


Fig. 7. Effect of varying *maxEpsilon* parameter on the average number of clients per pattern.

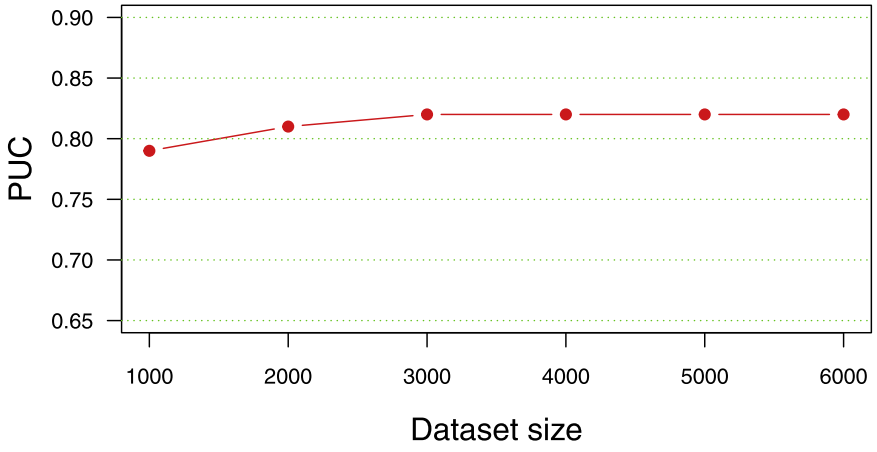


Fig. 8. Effect of varying the dataset size on the average pattern cohesion *maxEpsilon* = 0.5.

patterns of each approach. For LibCUP, we fixed the *maxEpsilon* value to 0.5 as explained earlier. For LibRec we fixed the *Minconf* to 0.8 , the *Minsup* to 0.002 and the *Number ofNearestNeighbors* to 25 (Thung et al., 2013).

5.3.2. Results for RQ2

Table 2 reports the obtained results for **RQ2**. On average, LibCUP achieves an average PUC score of 0.82 which outperforms LibRec that was only able to achieve 0.72 of PUC. We also notice that the standard deviation values are very low. The achieved PUC values by LibCUP reflect high co-usage relationships between the pattern's libraries making them more cohesive. The box plots of Fig. 11 confirm this observation.

In terms of number of inferred patterns, we observe that our multi-layer clustering technique allows detecting a reasonable number of patterns of 531, with a medium size of libraries distributed on the

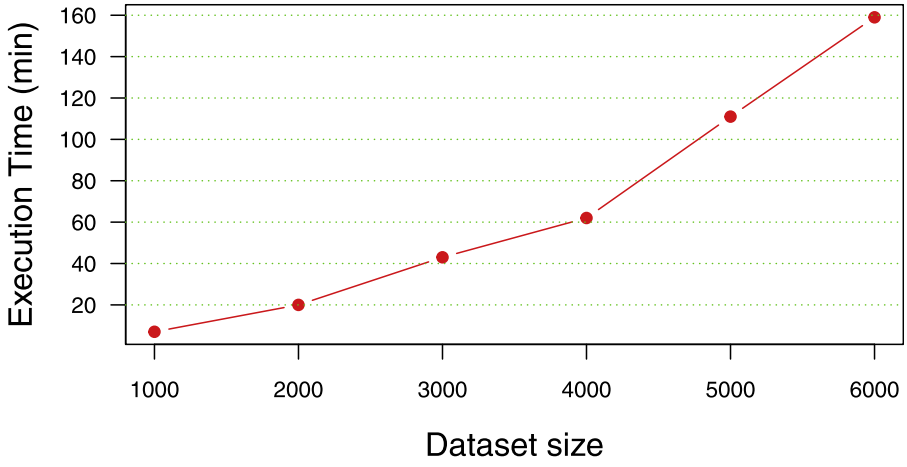


Fig. 9. Effect of varying the dataset size on the time efficiency with *maxEpsilon* = 0.5.

Table 2
Average cohesion and overview of the inferred usage patterns for LibCUP and LibRec.

	LibCUP	LibRec
Avg PUC	0.82	0.72
StdDev	0.07	0.09
Nb Patterns	531	3952
Avg pattern size	5.5	2.0
Nb Clients per Pattern	30	2269
Execution time (minutes)	159	93

different layers (i.e., 5.5). On the other hand, LibRec inferred an abundant number of patterns up to 3,952, even though it relies on closed itemsets and generators to construct a compact set of association rules (Thung et al., 2013). Indeed, the set of patterns obtained from the closed itemsets and generators is supposed to be much smaller than the complete set of rules. However, in practice the inferred patterns with LibRec tend to be many but with smaller size (on average, it generates 2 libraries per pattern). We believe that this large number of small size library patterns will in turn limit the practical adoption and usefulness of the LibRec approach.

As we can see in Table 2, LibRec tends to be more efficient than LibCUP in term of execution time. To push further the execution time comparison, we illustrate in Fig. 10 the influence of the dataset size on the execution time of LibRec. As it can be seen in Fig. 10, the execution time of LibRec is also sensitive to the dataset size. Based on Figs. 9 and 10, we observe that for up to 2000 considered libraries, LibCUP is more efficient than LibRec. Thereafter, the order is reversed and LibRec become more efficient. In terms of execution time, LibCUP requires 159 min of execution time to mine the 6000 libraries and their 38,000 client systems, whereas LibRec only need 93 min. We believe that LibCUP is struggling with large datasets because in our proof of concept we are not using an indexing structure for the identification of nearby neighbors. Indeed, while a faster version of DBSCAN exists to solve this problem, the problem of mining library usage patterns is not a real-time problem. Indeed, the patterns are identified once off-line and shows the patterns to the user. A re-execution of the algorithm is required only when the dataset of clients and libraries changes.

To get a more qualitative sense, we studied the number of clients per pattern. We observe from the results of Table 2, that the patterns inferred by LibCUP are used on average with 30 client systems. By manually investigating these client systems, we found that they generally share common domain specific features. An example of those patterns, can fulfil the requirements of the case scenario discussed in Section 2.1 that provide useful libraries for potential extensions of the system. The developer would use `scheduler-api` and `mailsender-api` rather than the `quartz` and `commons-email`. This pattern has

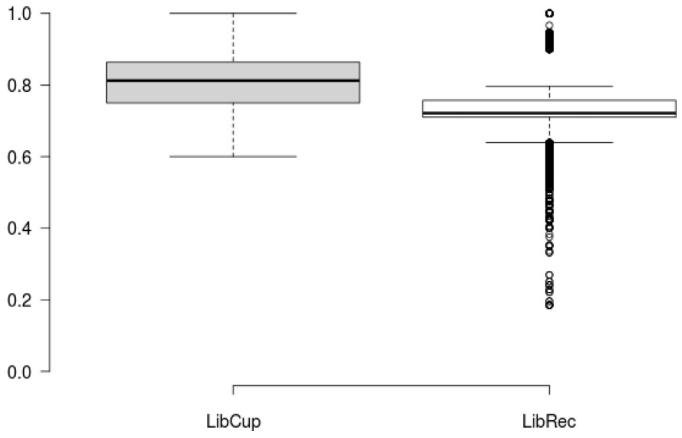


Fig. 11. PUC results of the identified library usage patterns.

different cohesion layers, and provides at the first layer, the libraries `sakai -calendar-api` and `sakai-presence-api` . In the second layer, we find three libraries that are added to the pattern, namely `portal-chat` , `messageforums-tool` and `mailsender-api` . At the external usage cohesion layer the `scheduler-api` is added to the pattern. Indeed, these libraries have been frequently co-used in a set of 18 client systems at least in our dataset. It is worth noticing that we found a trade-off between the usage cohesion of the detected patterns and their generalization. Indeed, an example of more generalizable patterns that was inferred when maxEpsilon parameter reached a relatively high value is the one based on the Spring framework presented in Study 1.A.

For LibRec, the inferred patterns are used within an excessive number of client systems that share pairs or triplets of libraries which are, in most of the cases, utility libraries such as `JUnit`, `log4j`, `slf4j-api`, `commons-lang` and several others. These libraries are likely to be used by several unrelated client systems. Examples of inferred patterns with LibRec are: { `commons-collections`, `commons-lang`} {`log4j`, `slf4j -api`} {`log4j`, `commons -logging`, `slf4j -api`}.

LibCUP is then able to infer both domain-specific patterns as well as more generalizable patterns for standard application. Whereas LibRec is meant for mining frequent patterns and thus is only able to infer generic utility pattern which in most of the cases do not add much to non-novice developers.

5.4. Evaluation of patterns generalization

In this study, we aim at evaluating whether the identified library usage patterns with LibCUP can be generalizable in comparison with

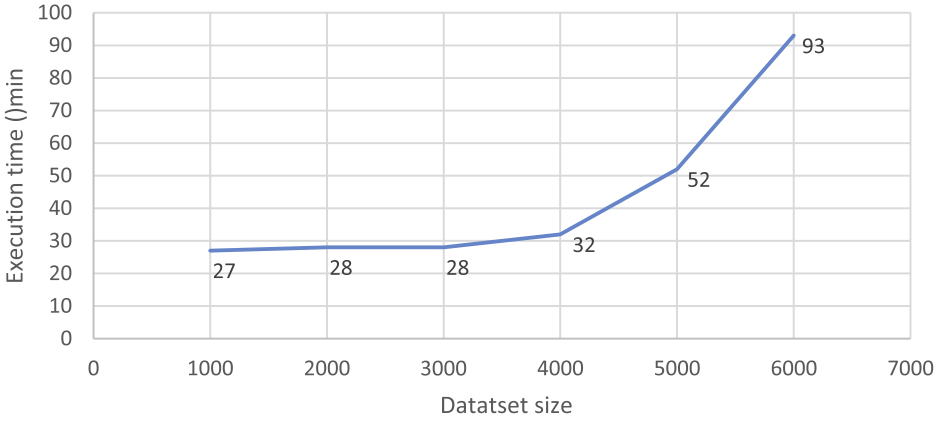


Fig. 10. Effect of varying the dataset size on the time efficiency of LibRec.

those of LibRec. We aim at addressing the following research question.

RQ3. *To which extent are the detected usage patterns generalizable to other "new" client systems, that are not considered in the training dataset?*

5.4.1. Analysis method

To answer RQ3, we investigate whether the detected patterns will have similar PUC values in the context of new client systems. We assume that *detected patterns are said "generalizable" if they remain characterized by a high usage cohesion degree in the contexts of various client systems.*

To evaluate the generalizability of the detected patterns, we perform a ten-fold cross-validation on all the client systems in the dataset. We randomly distribute the dataset into ten equal-sized parts. Then, we perform ten independent runs of both approaches, LibCUP and LibRec. Each run uses nine parts as training client systems to detect possible patterns, and leaves away the remaining part as a validation dataset.

The results are sorted in ten runs, where each run has its associated patterns, and its corresponding training and validation client systems. Then, we address (RQ3) through two experimental studies as follows.

Study 3.A. We evaluate the cohesion of the detected patterns (as measured by PUC) in the context of validation datasets. In a given run, it is possible that some detected patterns contain only libraries that are never used in the validation client systems. Consequently, to evaluate the generalizability of the detected patterns in each run, we consider only the patterns that contain at least one library that is actually used by the run's validation client systems.

We call such patterns the '*eligible patterns*' for the validation client systems. An eligible pattern will have a low PUC if only a small subset of its libraries is used by the validation client systems, while the other libraries have not been used. As a consequence, it will be considered as "non-generalizable". This study aims at comparing the PUC results obtained for the training client systems context and validation client systems context for both LibCUP and LibRec.

Study 3.B. In this study, we push further the comparison, as LibRec is specifically designed for library recommendation. We attempt to evaluate whether our approach is also useful in a recommendation context. To this end, we define for the library patterns inferred by LibCUP an ad-hoc ranking score based on the pattern cohesion and the library usage similarity.

For each fold, we identify a recommendation set of useful libraries for the validation client systems. For each system, we drop half of its libraries and use them as the ground truth. The remaining half is used as input to the recommendation process. This methodology was also used in Thung et al. (2013) and mimics the scenario where a developer knows some of the useful libraries but needs assistance to find other relevant libraries.

For each system that should receive library recommendation, we first identified potentially useful patterns containing at least one library from the ground truth set. Thereafter, we rank the libraries of these patterns according to their recommendation score as defined below:

$$RecScore(L) = \max_i \{USim(L, Lib_i)/Lib_i \in GT\} \quad (3)$$

where *USim* is the Usage Similarity in Eq. (1), and *GT* is the set of libraries conserved as ground truth of the client system that should receive library recommendations.

We evaluate the ranking for both LibCUP and LibRec using two metrics commonly used in recommendation systems for software engineering (Avazpour et al., 2014; Tantithamthavorn et al., 2013b; 2013a): (*i*) the recall rate@*k*, and (*ii*) the Mean reciprocal rank (MRR) as follows. To measure the recall@*k*, we consider *N* target systems *S*that should receive library recommendations. For each system $S_i \in S$, if any of the dropped libraries is found in the top-*k* list of recommended libraries, we count it as a hit. The recall rate@*k* is measured by the ratio of the number of hits over the total number *N* of considered systems. Inspired by the previous studies (Avazpour et al., 2014; Tantithamthavorn et al., 2013b; 2013a), we choose the *k* value to be 1, 3, 5, 7, and 10. Formally, the recall rate@*k* is defined as follows:

$$Recall\ rate@k(S) = \frac{\sum_{\forall S_i \in S} isCorrect(S_i, \text{Top-}k)}{N} \quad (4)$$

where the function $isCorrect(S_i, \text{Top-}k)$ returns a value of 1 if at least one of the top-*k* recommended libraries is in the ground truth set, or 0 otherwise.

The MRR is a statistic measure that is commonly used to evaluate recommendation systems. Let *N* be the number of systems that should receive recommendations, and $rank_i$ is the rank of the first relevant recommendation for the i^{th} system. MRR reflects the overall ranking performance, it shows on average the rank of the first relevant recommendation. A score of 1 indicates that the recommendation system provides a relevant recommendation in the first rank, whereas a score of 0.5 indicates that on average the first relevant recommendation is in the second rank. The MRR score is calculated as follows:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i} \quad (5)$$

5.4.2. Results for RQ3

The obtained results are as follows.

Study 3.A: Patterns generalizability. To assess the PUC score variation between the training and validation client systems, we first analyze the average value of their corresponding scores collected from all cross-validation runs. Then, we analyze the distribution of the collected values using the box plots. The results of this study are summarized in Table 3 and Fig. 12.

Table 3 summarizes the PUC results of the detected patterns in the contexts of training and validation client systems. In the training context, we notice that the average values are high for both LibCUP and LibRec with respectively 77% and 72% of the patterns libraries that are co-used together. For LibCUP, a slight degradation of the PUC value is observed in the context of validation client systems. We also notice that the standard deviation values are very low (0.01 and 0.05). These results reflect that, overall, the detected patterns had good PUC in the context of both validation and training client systems. However, for LibRec the achieved average PUC values are significantly lower in the validation context compared to the training context.

In more details, the distribution of PUC values for all detected usage patterns in Fig. 12 confirms the above-mentioned finding. Indeed, the medians and lower quartiles in the context of validation clients remain larger than 66%. Fig. 12 also provides evidence that the degradation of cohesion values for each inferred pattern is much more visible for LibRec.

Moreover, to compare the distributions of LibCUP and LibRec, we used the Wilcoxon Signed Rank test in a pairwise fashion (Cohen, 1988) in order to detect significant performance differences between the validation and training client systems in both approaches. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values ranks instead of operating on the values

Table 3
Average Training and Validation Cohesion of identified usage patterns for LibCUP and LibRec.

	LibCUP		LibRec	
PUC	Training context	Validation context	Training context	Validation context
Avg	0.77	0.69	0.72	0.54
Max	0.78	0.78	0.88	0.89
StdDev	0.01	0.05	0.06	0.27

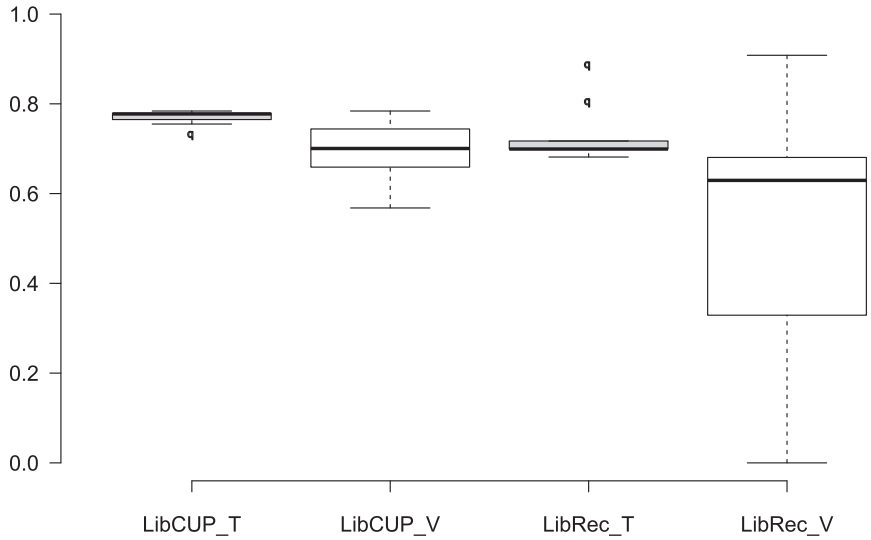


Fig. 12. PUC results of the identified library usage patterns in the contexts of training (T) and validation (V) clients achieved by each of LibCUP and LibRec.

themselves. We set the confidence limit, α, at 0.01. The Wilcoxon statistical tests show that LibCUP has no statistical difference ($\alpha = 0.009$) when comparing the validation and training client systems. These results indicate that LibCUP preserves the generalizability. Whereas, the statistical test show that LibRec exhibits a statistical difference with $\alpha = 0.027$ when comparing the validation and training client systems which indicate that it does not preserve the generalizability.

In summary, we can say that almost all detected usage patterns achieved by LibCUP retain their informative criteria. Precisely, 75% of detected usage patterns, according to the boxplot's lower quartile, are characterized with a high usage cohesion in both training and validation contexts.

Study 3.B: Library recommendation. Table 4 reports the recall rate@k for both LibCUP and LibRec while varying the value of $k \in \{1, 3, 5, 7, 10\}$. We notice that, as expected, larger k values achieve higher recall rates for both approaches. More specifically, we can see that when comparing the recall rate, LibCUP performs clearly better in terms of recall@1 and recall@3. However, the starting from $k = 5$, LibRec tends to achieve better results. This indicates that LibCUP is more efficient in recommending correct libraries in the top ranks within the recommendation list when using LibCUP. The MRR results support this observation with a score of 0.15 for LibCUP.

Conversely, good recommendations are achieved for more targeted client systems when using LibRec. This is mainly due to the fact that LibRec's patterns are mainly composed of utility libraries, unlike the LibCUP's patterns which are mainly composed of domain specific libraries. However, one can notice that recommending utility libraries that are commonly used is less useful in practice. It is also worth mentioning that for each fold, due to the large number of systems (38,000 validation client systems) that require library recommendations, the recall values achieved by both LibCUP and LibRec are still low as reported in Table 4.

Table 4
Recommendation recall rate results achieved by both LibCUP and LibRec.

	LibCUP	LibRec
Recall@1	0.12	0.01
Recall@3	0.14	0.11
Recall@5	0.15	0.19
Recall@7	0.17	0.27
Recall@10	0.22	0.34

6. Discussion and threats to validity

We applied our approach to over 6000 popular third-party libraries in order to detect possible library usage patterns. The detected patterns should be informative to help developers in automatically discovering complementary libraries sets and therefore relieve the developers from the burden of doing so manually.

The evaluation of our approach took into account the potential generalization of the identified patterns to other client systems and showed that these usage patterns remain informative for other clients. However, several factors can bias the validity of our studies, Some threats to validity may arise from our evaluation metrics. Although these metrics are well known measures that are widely used in evaluating recommendation systems, we believe that there is a little bias towards using these measures. That is, a common assumption when evaluating recommendation systems using such metrics is that items that the user has not selected are uninteresting, or useless, to other users. Hence, in the library recommendation problem, a system might not adopt a specific library for many reasons mainly if the systems developers are not aware of such a library. We are thus planning to further evaluate our technique with developers in an industrial setting to assess its effect on code quality as well as developers productivity.

Although we studies libraries hosted in Maven ecosystem which is the largest Java library ecosystem, it is worth notice that Maven is not inclusive for all existing Java libraries. That is, it is possible that some libraries might be used/imported directly in a client system without using the Maven dependency system (pom file). In our data collection, when a used library is not referenced in the pom file, we are unable to consider its dependency which might be a possible threat to validity. Moreover, to better generalize the results of our approach, it is important to consider other programming languages and other library ecosystems.

Third-party libraries that are declared without using Maven are not the only missed dependencies. There may be cases where the tool PomWalker suffers from the limitation of static detection of dependencies. For instance, PomWalker is able to collect managed dependencies (i.e., inherited) within systems, but is unable to summarize all subsystems within a repository, and thus may fail to resolve some dependencies. To mitigate this issue for each repository, we extract systems and subsystems with their dependencies.

As each Github repository may contain multiple projects, each having potentially several systems. Each of these systems is dependent on a set of maven libraries, which are defined in a pom.xml file within

the project. When the pom files are stored hierarchically, and referencing the same library, this will count as different client dependency to the considered library. We are unable to resolve this duplicated dependency which is a potential threat to validity.

One of the key contributions of this work is the adaptation of our variant ϵ-DBSCAN algorithm for mining library usage patterns. We have opted for this DBSCAN-based technique rather than a standard clustering technique since DBSCAN has the notion of noise, and it is widely considered robust to outliers. Our variant ϵ-DBSCAN also shows high scalability and performance even with the large dataset used in this work.

The application of our technique to detect library usage patterns requires the setting of thresholds that may impact its output. For instance, the *maxEpsilon* parameter in the clustering algorithm controls the cohesion (PUC) strength of the detected patterns. A small value leads to highly cohesive clusters which means that the detected patterns are more informative. Hence, decreasing the value of this parameter would result in an improvement in cohesion of the detected patterns. However, in this case the number and the consistency (generality) of the detected patterns could decrease because the highly cohesive detected patterns may not be shared by a large number of clients. To avoid bothering potential users of our approach with tuning *epsilonStep* and *maxEpsilon* parameters, we set default values as follows. The value of *epsilonStep* is set to 0.05. The lower the *epsilonStep* value is, the more precise the distribution of libraries across the different layers will be. However, this value may impact the execution time of the algorithm, since very low values may increase the number of clustering iteration in which the patterns are not evolved. We also set the *maxEpsilon* parameter to a default value of 0.5 which ensures that the libraries within patterns are at least used more frequently together than separately. Note that in our approach, the patterns were mined with a *maxEpsilon* value that is set to 1, which allows capturing all possible co-usage levels. Thus, the user can fix epsilon at low values and investigate only highly cohesive patterns. He can also fix epsilon at high values and examine less cohesive patterns.

Our approach mines the 'wisdom of the crowd', to discover library co-usage patterns which allows developers and especially non-experienced ones to have higher confidence in their choice for potential libraries. They are already sure that the chosen libraries have been commonly used together in other projects. Hence, developers will have more confidence regarding the integrality and compatibility of the selected libraries. However, the practical value of our approach needs to be evaluated through a user study, to assess its usefulness for developers to make better choices than without the tool, and when compared to the baseline technique LibRec. Although this is part of our future work, we attempted to mitigate this limitation in RQ3 Study 3.B to provide insights about the usefulness of our technique in a recommendation context where we mimic a developer who knows subset of useful libraries but needs assistance to find other relevant libraries.

In our visualization tool, we considered the popularity of libraries as a criterion to help the developer in exploring the jungle of libraries. However, different other characteristics could be considered by developers when selecting the most appropriate libraries for their needs. When a developer ignores aspects such as the quality of the library and the activity of the library development team, he will take the risk of using unstable and poor quality libraries. In our future work, we will consider other relevant characteristics for library selection. Nevertheless, we believe that our visualization tool can still mitigate this issue. The user of our tool can filter libraries according to their popularity and focus on a subset of libraries and patterns of popular libraries. Since we are mining the 'wisdom of the crowd', the popularity of libraries could be seen as a proxy heuristic for other quality characteristics unless the popularity of the library in the dataset snapshot was only a trend effect.

We are currently based on a specific set of client systems in a particular snapshot in time to infer library usage patterns. However, a potential threat to validity can be related to the robustness of our analysis to changes in the snapshot date as library usage trends is by nature dynamic and changes over time. To mitigate this issue, we will consider periodical updates of the snapshots through a Web service, and also provide the patterns evolution through different snapshots to better support developers in selecting appropriate libraries.

Another important issue is whether a tool like LibCUP is a kind of "self-fulfilling prophecy" where more popular libraries are more likely to be recommended by the tool. In addition, due to the phenomenon of "preferential attachment," popular libraries will become more used by developers and thus more popular with a higher chance to be recommended in the future. In such a way, less popular libraries, that may even be more appropriate, are at risk of not being recommended. We can avoid this phenomenon, thanks to the used metrics for pattern inference and library recommendation. The pattern inference is not based on the libraries popularity; It is rather based on the analysis of the joint versus separate use of the libraries. We can infer patterns also for non-popular libraries. When a set of libraries are not popular (are used just in few clients) but they are used together in their clients, these libraries could be considered as a pattern. Moreover, the library recommendation is not based on the library popularity. It is rather based on the patterns' cohesion and the library usage similarity. The popularity of libraries is only used in our visualization tool as a filtering criterion to help the developer in exploring the jungle of libraries. For these reasons, the preferential attachment phenomenon (Wang et al., 2008) is a negligible risk for our technique. However, the users of the tool should be aware of this issue to avoid neglecting patterns of less popular libraries when exploring the jungle of libraries and selecting the appropriate ones. In the future work, we will also investigate the possibility of recommending and showing equivalent and alternative libraries based on the provided functionality and their used vocabulary.

7. Conclusions and future work

Third-party library reuse has become vital in modern software development. The number of libraries provided on the Internet is exponentially growing which would provide several reuse opportunities. In this paper, we introduced an automated approach to detect multi-level library usage patterns – a collection of libraries that are commonly used together by client systems, distributed through multiple levels of cohesion. To this end, we adopted a variant of the standard clustering algorithm DBSCAN, namely ϵ-DBSCAN especially for the library usage patterns detection. We evaluated our approach on large dataset of 6638 popular libraries from Maven repository, and a large population of 38,000 client systems from Github, and we compared its results to those of a state-of-the-art approach. The results indicate that our approach gives a comprehensive overview on third-party library usage patterns. The obtained usage patterns exhibit high usage cohesion with an average of 77% , and could be generalizable to other other systems. Automatically detecting library usage patterns would support developers in enhancing the library space discovery, and attract their attention the missed reuse opportunities.

As future work, we are planning to consider the library usage over time while mining usage pattern. We plan to conduct an empirical study to investigate how usage patterns evolve when they are mined through different snapshots in time. We will also examine the evolution of libraries popularity over time and how this information could be incorporated in an automated recommendation system. This will also be an occasion to consider more recent data. Furthermore, we are interested in exploring how other techniques that have been applied as an alternative to hierarchical clustering in the domain of software engineering, could support the library usage patterns inference. We are specifically interested in adapting Formal Concept Analysis (Ganter and Wille, 2012) for our problem. Considering that concept lattices are hierarchies, we believe the structure of the lattice could have a potential for analyzing libraries co-usage at multiple levels.

In addition, we are interested in exploring how other techniques that have been applied as an alternative to hierarchical clustering in the domain of software engineering, could support the library usage patterns inference. We are specifically interested in adapting Formal Concept Analysis (Ganter and Wille, 2012) for our problem. Considering that concept lattices are hierarchies, we believe the structure of the lattice could have a potential for analyzing libraries co-usage at multiple levels.

As future work, we will consider other sources of information for libraries pattern inference and recommendation. In fact, a co-usage based approach such as LibCUP would not be able to recommend libraries to projects that only use a small number of libraries or do not use any libraries at all. In such situation, vocabulary based semantic similarity of libraries and client systems could be considered. We specifically plan to investigate Relational Topic Modeling (RTM) (Chang and Blei, 2009) which can be used to summarize a network of documents and can provide clusterings of nodes based on their vocabulary.

Our visualization tool is meant for the early stage of the project lifecycle when the developer needs to decide on the required libraries for his project. Furthermore, we are working towards an on-the-fly interactive library recommendation tool as an Eclipse Plug-in that recommends useful libraries to the developer while he is writing his code. Such tool would be interesting during advanced stages of the project lifecycle. We will also evaluate the usability of the tools through a controlled user study. Furthermore, we are planing to unify our library-level usage pattern detection with method-level usage pattern detection techniques in order to provide a comprehensive package for developers supporting them in understanding and reusing third-party libraries.

References

Antoniol, G., Di Penta, M., Neteler, M., 2003. Moving to smaller libraries via clustering and genetic algorithms. European Conference on Software Maintenance and Reengineering. pp. 307–316.

Antoniol, G., Penta, M.D., 2003. Library miniaturization using static and dynamic information. Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on. IEEE, pp. 235–244.

Asaduzzaman, M., Roy, C.K., Schneider, K., Hou, D., 2014. Cscc: Simple, efficient, context sensitive code completion. International Conference on Software Maintenance and Evolution (ICSME). pp. 71–80.

Avazpour, I., Pitakrat, T., Grunske, L., Grundy, J., 2014. In: Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T. (Eds.), Recommendation Systems in Software Engineering. Springer Berlin Heidelberg, pp. 245–273. https://doi.org/10.1007/978-3-642-45135-5_10. chapter 10.

Bruch, M., Monperrus, M., Mezini, M., 2009. Learning from examples to improve code completion systems. Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. pp. 213–222.

Buse, R.P.L., Weimer, W., 2012. Synthesizing API usage examples. International Conference on Software Engineering (ICSE). pp. 782–792.

Chang, J., Blei, D., 2009. Relational topic models for document networks. Artificial Intelligence and Statistics. pp. 81–88.

Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences. 2nd.

De Roover, C., Lammel, R., Pek, E., 2013. Multi-dimensional exploration of API usage. International Conference on Program Comprehension. pp. 152–161.

Duala-Ekoko, E., Robillard, M.P., 2011. Using structure-based recommendations to facilitate discoverability in APIs. European Conference on Object-oriented Programming. pp. 79–104.

Endrikat, S., Hanenberg, S., Robbes, R., Stefik, A., 2014. How do API documentation and static typing affect API usability? International Conf. on Software Engineering. ACM, pp. 632–642.

Ester, M., peter Kriegel, H., Xu, X., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. International Conference on Knowledge Discovery and Data Mining. pp. 226–231.

Ganter, B., Wille, R., 2012. Formal Concept Analysis: Mathematical Foundations. Springer Science & Business Media.

Hou, D., Li, L., 2011. Obstacles in using frameworks and APIs: an exploratory study of programmers' newsgroup discussions. International Conference on Program Comprehension. pp. 91–100.

Hou, D., Pletcher, D.M., 2011. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. International Conference on Software Maintenance. pp. 233–242.

Huppe, S., Saied, M.A., Sahraoui, H., 2017. Mining complex temporal API usage patterns: an evolutionary approach. Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on. IEEE, pp. 274–276.

Kula, R.G., De Roover, C., German, D., Ishio, T., Inoue, K., 2014. Visualizing the evolution of systems and their library dependencies. Working Conference on Software Visualization (VISSOFT). IEEE, pp. 127–136.

Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K., 2018. Do developers update their library dependencies? Emp. Softw. Eng. 23 (1), 384–417.

Li, Z., Zhou, Y., 2005. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. ACM SIGSOFT Software Engineering Notes. Vol. 30. ACM, pp. 306–315.

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C., 2011. Portfolio: Finding relevant functions and their usage. International Conference on Software Engineering (ICSE). pp. 111–120.

Montandon, J.E., Borges, H., Felix, D., Valente, M.T., 2013. Documenting APIs with examples: lessons learned with the apiminer platform. Working Conference on Reverse Engineering. pp. 401–408.

Moritz, E., Linares-Vasquez, M., Poshyvanyk, D., Grechanik, M., McMillan, C., Gethers, M., 2013. Export: Detecting and visualizing API usages in large source code repositories. Automated Software Engineering. pp. 646–651.

Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J., Nguyen, T.N., 2012. Graph-based pattern-oriented, context-sensitive source code completion. International Conference on Software Engineering (ICSE). pp. 69–79.

Ouni, A., Kula, R.G., Kessentini, M., Ishio, T., German, D.M., Inoue, K., 2017. Search-based software library recommendation using multi-objective optimization. Inf. Softw. Technol. 83, 55–75.

Parnin, C., Treude, C., Grammel, L., Storey, M.-A., 2012. Crowd Documentation: Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow. Tech. Rep. Georgia Institute of Technology.

Penta, M.D., Neteler, M., Antoniol, G., Merlo, E., 2002. Knowledge-based library re-factoring for an open source project. Working Conference on Reverse Engineering (WCRE). IEEE, pp. 319–328.

Perepletchikov, M., Ryan, C., Frampton, K., 2007. Cohesion metrics for predicting maintainability of service-oriented software. International Conference on Quality Software. pp. 328–335.

Raemaekers, S., van Deursen, A., Visser, J., 2012. Measuring software library stability through historical version analysis. Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE, pp. 378–387.

Raemaekers, S., Van Deursen, A., Visser, J., 2014. Semantic versioning versus breaking changes: A study of the maven repository. Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on. IEEE, pp. 215–224.

Robillard, M.P., 2009. What makes APIs hard to learn? answers from developers. IEEE Softw. 26 (6), 27–34.

Saied, M.A., Abdeen, H., Benomar, O., Sahraoui, H., 2015. Could we infer unordered API usage patterns only using the library source code? International Conference on Program Comprehension (ICPC). IEEE Press, pp. 71–81.

Saied, M.A., Abdeen, H., Benomar, O., Sahraoui, H., 2015. Could we infer unordered API usage patterns only using the library source code? Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. IEEE Press, pp. 71–81.

Saied, M.A., Benomar, O., Abdeen, H., Sahraoui, H., 2015. Mining multi-level API usage patterns. International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 23–32.

Saied, M.A., Benomar, O., Sahraoui, H., 2015. Visualization based API usage patterns refining. Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on. IEEE, pp. 155–159.

Saied, M.A., Sahraoui, H., 2016. A cooperative approach for combining client-based and library-based API usage pattern mining. Program Comprehension (ICPC), 2016 IEEE 24th International Conference on. IEEE, pp. 1–10.

Saied, M.A., Sahraoui, H., Dufour, B., 2015. An observational study on API usage constraints and their documentation. International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 33–42.

Schubert, E., Sander, J., Ester, M., Kriegel, H.P., Xu, X., 2017. DBSCAN Revisited, revisited: why and how you should (still) use DBSCAN. ACM Trans. Database Syst. (TODS) 42 (3), 19.

Szathmary, L., Napoli, A., Kuznetsov, S. O., 2006. Zart: A multifunctional itemset mining algorithm.

Tantithamthavorn, C., Ihara, A., Matsumoto, K.-i., 2013. Using co-change histories to improve bug localization performance. ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). IEEE, pp. 543–548.

Tantithamthavorn, C., Teekavanich, R., Ihara, A., Matsumoto, K.-i., 2013. Mining a change history to quickly identify bug locations: A case study of the Eclipse project. International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, pp. 108–113.

Thung, F., Lo, D., Lawall, J., 2013. Automated library recommendation, 182–191.

Uddin, G., Dagenais, B., Robillard, M.P., 2012. Temporal analysis of API usage concepts. International Conf. on Software Engineering. pp. 804–814.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D., 2013. Mining succinct and high-coverage API usage patterns from source code. Working Conference on Mining Software Repositories (MSR). pp. 319–328.

Wang, L., Fang, L., Wang, L., Li, G., Xie, B., Yang, F., 2011. APIExample: An effective web search based usage example recommendation system for java APIs. International Conference on Automated Software Engineering (ICSE). pp. 592–595.

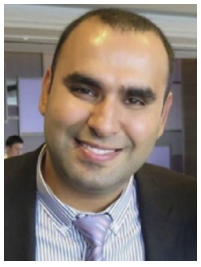
Wang, M., Yu, G., Yu, D., 2008. Measuring the preferential attachment mechanism in citation networks. Physica A 387 (18), 4692–4698.

Wang, W., Godfrey, M.W., 2013. Detecting API usage obstacles: A study of iOS and Android developer questions. Working Conference on Mining Software Repositories. pp. 61–64.

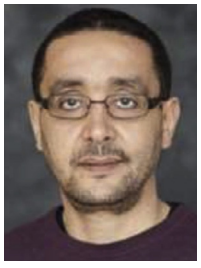
Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H., 2009. MAPO: Mining and recommending

API usage patterns. European Conference on Object-Oriented Programming. pp. 318–343.

Mohamed Aymen Saied is a postdoctoral fellow at the Department of Electrical and Computer Engineering Concordia University. He received his Ph.D. degree in computer science from University of Montreal. His research interests are at the intersection of software engineering, software systems, and software data analytics. His current research interests include software reuse, microservices architecture and mobile applications.



Ali Ouni is an associate professor at the department of Software Engineering and IT at ETS Montreal, University of Quebec. He received his Ph.D. degree in computer science from University of Montreal in 2014. His research interests are in software engineering including software maintenance and evolution, refactoring of software systems, software quality, service-oriented computing, and the application of artificial intelligence techniques to software engineering. His work has received several nominations and Best Paper Awards.



Houari A. Sahraoui is professor at the department of computer science and operations research (GEODES, software engineering group) of University of Montreal. His research interests include software engineering automation, model-driven engineering, software visualization, and search-based software engineering. He has served as a program committee member in several IEEE and ACM conferences, as a member of the editorial boards of four journals, and as an organization member of many conferences and workshops.

Raula Gaikovina Kula is an assistant professor at Nara Institute of Science and Technology. He received the Ph.D degree from Nara Institute of Science and Technology in 2013. He was a Research Assistant Professor at Osaka University from Sept. 2013 till April 2017. His current research interests include software ecosystems, software reuse and code reviews.

Katsuro Inoue is a professor of Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. His current research interest includes software ecosystem modeling, software provenance analysis, and code clone detection.

David Lo received his Ph.D. degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has more than 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009, and a number of international research awards including several ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).