Search-based software library recommendation using multi-objective optimization

Ali Ouni [a,b,*], Raula Gaikovina Kula [a], Marouane Kessentini [c], Takashi Ishio [a], Daniel M. German [d], Katsuro Inoue [a]

[a] Department of Computer Science, IST, Osaka University, Osaka, Japan
[b] Department of Computer Science and Software Engineering, UAE University, UAE
[c] Department of Computer and Information Science, University of Michigan, MI, USA
[d] Department of Computer Science, University of Victoria, Victoria, Canada

ARTICLE INFO

ABSTRACT

Context: Software library reuse has significantly increased the productivity of software developers, reduced time-to-market and improved software quality and reusability. However, with the growing number of reusable software libraries in code repositories, finding and adopting a relevant software library becomes a fastidious and complex task for developers.

Objective: In this paper, we propose a novel approach called *LibFinder* to prevent missed reuse opportunities during software maintenance and evolution. The goal is to provide a decision support for developers to easily find "useful" third-party libraries to the implementation of their software systems.

Method: To this end, we used the non-dominated sorting genetic algorithm (NSGA-II), a multi-objective search-based algorithm, to find a trade-off between three objectives : 1) maximizing co-usage between a candidate library and the actual libraries used by a given system, 2) maximizing the semantic similarity between a candidate library and the source code of the system, and 3) minimizing the number of recommended libraries.

Results: We evaluated our approach on 6083 different libraries from Maven Central super repository that were used by 32,760 client systems obtained from Github super repository. Our results show that our approach outperforms three other existing search techniques and a state-of-the art approach, not based on heuristic search, and succeeds in recommending useful libraries at an accuracy score of 92%, precision of 51% and recall of 68%, while finding the best trade-off between the three considered objectives. Furthermore, we evaluate the usefulness of our approach in practice through an empirical study on two industrial Java systems with developers. Results show that the top 10 recommended libraries was rated by the original developers with an average of 3.25 out of 5.

Conclusion: This study suggests that (1) library usage history collected from different client systems and (2) library semantics/content embodied in library identifiers should be balanced together for an efficient library recommendation technique.

1. Introduction

Modern software systems build on a significant number of third-party software libraries to deliver feature-rich and high-quality software. Several studies have shown that software library reuse promotes efficient and effective software development. Consequently, library reuse leads to a significant increase in the productivity, reduction in time-to-market, improvement in the overall software quality, as well as reducing the inherent testing costs [1,2]. Reusing mature software modules can benefit from the collective experience of previous users of the module, as many bugs as well as deficiencies in the documentation have already been discovered [3,4].

Indeed, it is recognized that replacing legacy code with quality components and libraries typically reduces the amount of source code that must be maintained [5]. The benefits of replacing legacy

* Corresponding author.
E-mail address: ouniaali@gmail.com (A. Ouni).

code by external quality software components was best articulated by Seacord et al. [5]: "*replacing functional components may also provide additional capabilities and improve on such attributes of system quality as robustness or performance*". In fact, replacing legacy code by third-party libraries has recently attracted much attention in both academia and industry. One example is the refactoring of "synchronized" blocks in Java by replacing them with the utility library `java.util.concurrent` [6,7].

Today, software systems utilize online code repositories such as Maven Central repository[1] to access to a host of reusable Open Source Software (OSS) libraries. Indeed, reusable software libraries are often reused multiple times and are therefore proven solutions that can provide better quality characteristics compared to newly developed code [8]. Recent empirical studies have found that 93.3% of modern OSS projects use third-party libraries, with an average of 28 libraries per project [9]. On the other hand, recent work indicate that developers are still often reinvent the wheel and spend effort and time, on re-implementing functionality, that could be saved by reusing mature and well-tested libraries [10,11].

We conjure two key reasons for this occurrence. First, due to the magnitude of available libraries we consider that, most of the time, developers are unaware or overwhelmed by related libraries. Online sources[2] report that available libraries are growing at an exponential rate. Hence, searching relevant software libraries can be a fastidious task for software developers, which would have an impact their productivity. Second, in addition to different reasons of distrust [12], developers are wary of the inherent costs and risks of library incompatibilities [13] associated with integrating new and unknown libraries into their existing systems. With the motto '*if not broke don't fix*', systems as a consequence risk outdated libraries.

To help developers, most of existing library recommendation approaches are based on commonly used together library methods, e.g., API usage patterns, at the method level of granularity [14–18]. The most related work of recommendation at the library level of granularity is by Thung et al. [9]. The authors use collaborative filtering and association rule mining on historic software artifacts to determine commonly used libraries without considering the library content. However, a library usage history-based approach 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. Thus, the content of a library is an extremely important asset that should be more informative and explicit for an effective library recommendation method. This approach deal with library recommendation as a single objective problem based on usage history. We believe that library recommendation is rather a complex decision making problem where several considerations should be balanced. These complex multi-objective decision problems with competing and conflicting constraints are well suited to Search Based Software Engineering (SBSE) [19,20].

To address the library recommendation problem, we introduce a novel approach called *LibFinder* based on the following two heuristics: a candidate library L can be useful for a given system S if (i) L has been commonly used with one or more libraries adopted by S, and (ii) L uses identical or similar identifiers, i.e., belongs to the same application domain, as S. To this end, we used the history of library usage as a 'wisdom of the crowd' and semantic similarity embodied in library identifiers mined from large code repositories from the internet. Our multi-objective formulation aims at finding optimal solutions providing the best trade-off between the three following objectives: 1) maximize co-usage between a candidate library and the actual libraries used by a given

system, 2) maximize the semantic similarity between a candidate library's code and the system's code, and 3) minimize the total number of recommended libraries. To this end, we used the popular multi-objective search-based algorithm the non-dominated sorting genetic algorithm (NSGA-II) [21] to find the best trade-off between the three objectives. The complexity of the addressed library recommendation problem is combinatorial since our formulation consists of assigning libraries to different code fragments and the search is guided based on the above dependent evaluation functions.

To evaluate the efficiency of our approach, we used the history of 32,760 software projects mined from Github, that were clients for 6083 Maven libraries. The obtained results show that our approach is efficient in recommending relevant software libraries. We compare our approach with random search and two other popular search-based algorithms as well as a state-of-the-art approach. The statistical analysis shows better performance of our approach with 92% of accuracy, 51% of precision and 68% of recall. Furthermore, we evaluate the usefulness of our approach in practice through an empirical study on two industrial Java systems with developers. Results show that the top 10 recommended libraries was rated by the original developers of both systems with an average of 3.25 out of 5.

The main contributions of this paper can be summarized as follows:

1. We propose a new search-based approach called *LibFinder*, to detect and recommend third-party libraries that may be relevant to software systems that have already been implemented, and that it is intended for maintenance and evolution. To the best of our knowledge, this is the first attempt to use SBSE to address the library recommendation problem.
2. We collect a rich dataset by (i) mining the usage history, and (ii) extracting the identifiers of a large set of popular libraries from Maven Central Repository. The dataset is publicly available to encourage future research in the field of library recommendation[3].
3. We present an empirical evaluation of the performance of our approach using a 10-fold cross validation, along with statistical analysis of the obtained results. The obtained results show that our approach outperforms random search and two other search techniques at a confidence level of 95% and outperforms a state-of-the-art library recommendation approach [9] with an accuracy score of 92%, precision score of 51% and recall score of 68% while finding the best trade-off between the considered objectives. We present the results of a second empirical study to evaluate our approach in two industrial systems in real world setting where the recommended libraries were rated 3.25 out of 5 on average.

The rest of the paper is organized as follows. Section 2 presents the necessary background and a motivating example. Section 3 presents the basic concepts of our approach. Section 4 introduces our search-based approach for library recommendation *LibFinder*. Section 5 describes our empirical study and reports the obtained results, while Section 6 presents the threats to validity of the study. Section 7 presents the related work. Finally, Section 8 concludes and presents our future research directions.

2. Background and motivating example

In this section, we first describe the necessary background related to the proposed approach. We then present an example to help readers to better understand the motivation for library recommendation.

2.1. Search based software engineering (SBSE) and mining software repositories (MSR)

Our approach is largely inspired by contributions in SBSE and MSR. SBSE studies the application of meta-heuristic optimization techniques to software engineering problems [22]. Once a software engineering task is framed as a search problem, by formulating it in terms of solution representation, objective function, and solution change operators, there are a multitude of search algorithms that can be applied to solve that problem. SBSE aims at exploring large search spaces of possible solutions for a particular problem in order to discover near optimal solutions.

On the other hand, the Mining Software Repositories (MSR) field analyzes the rich data available in software repositories to uncover interesting and actionable information about software systems and projects. MSR transforms software repositories to gain empirical understanding of software development. This can be leveraged by software practitioners to estimate and manage various aspects of their projects [23,24].

In recent years, both fields are widely applied to solve several software maintenance and evolution problems including refactoring, testing, modularization, planning, and so on [22–24]. However, despite the innate link between both fields, SBSE and MSR communities are still not unified. Indeed, library recommendation is a complex task, and one of the non-obvious software engineering problems that can benefit from both SBSE and MSR techniques.

2.2. Recommendation systems

Recommendation systems support users and developers of various computer and software systems to overcome information overload, perform information discovery tasks and approximate computation, among others [25]. With the increasing size and complexity of software systems and software engineering data, recommendation systems play an important role in providing a decision support for software engineers. Indeed, recommendation systems have recently become popular in software engineering and have attracted a wide variety of application scenarios from business process modeling to source code maintenance and manipulation [9,26,27]. Recommendation systems use a number of different technologies [28,29], and can be classified into two broad classes.

- *Content-based systems:* These systems focus on properties of items. Similarity of items is determined by measuring the similarity in their properties and features.
- *Collaborative-filtering systems:* These systems focus on the relationship between users and items based on the usage history. Similarity of items is determined by the similarity of the utilization/ratings of those items by the users who have used/rated both items.

Each class has its own advantages and disadvantages. For instance, content-based approaches provide user independence, in contrast to collaborative filtering that needs other users' history to find the similarity between them and then give the recommendations. Content-based methods only need to analyze users and items features. Moreover, collaborative filtering methods provide recommendations for a user based on some unknown users who might have the same taste, while content-based methods provide recommendations based on what item's features the users like. On the other hand, unlike collaborative filtering, new items can be recommended by a content-based method before being used by a substantial number of users. Thus, both classes can be combined together in order to provide more effective recommendation systems.

Algorithm 1 High level pseudo code for NSGA-II.

1: Create an initial population P_0
2: Create an offspring population Q_0
3: $t = 0$
4: **while** stopping criteria not reached **do**
5:    $R_t = P_t \cup Q_t$
6:    F = fast-non-dominated-sort(R_t)
7:    $P_{t+1} = \emptyset$ *and* $i = 1$
8:    **while** $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ **do**
9:       Apply crowding-distance-assignment(F_i)
10:       $P_{t+1} = P_{t+1} \cup F_i$
11:       $i = i + 1$
12:    **end while**
13:    $Sort(F_i, \prec n)$
14:    $P_{t+1} = P_{t+1} \cup F_i[N- \mid P_{t+1} \mid]$
15:    Q_{t+1} = create-new-pop(P_{t+1})
16:    t = t+1
17: **end while**

2.3. Multi-objective search-based algorithms

Multi-objective problems contain several objectives, and the goal is to find solutions that are able to optimally satisfy each objective simultaneously. However, in real world problems, it is difficult (or even impossible) to find a solution that is concurrently perfect for each objective for a problem with two or more objectives due to conflicts that always exist among problem objectives. Thus, certain expenses and trade-offs always exist between the multiple objectives of a problem [30]. Many real-world problems involve simultaneous optimization of several incommensurable and often competing objectives. Often, there is no single optimal solution, but rather a set of alternative solutions. One of the most popular multi-objective search-based algorithms is the non-dominated sorting genetic algorithm (NSGA-II) [21] that has shown high performance in solving several software engineering problems [22].

A high-level view of NSGA-II is depicted in Algorithm 1. NSGA-II starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [21] is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives f_i, $i \in$ 1..n, to maximize, a solution x_1 dominates x_2:

$$iff \ \forall i, f_i(x_2) \leq f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1)$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0 Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [21] to make the selection (line 9). This

```java
public DateEdit(String l, Date date, ChangeListener listener) {
calendar.setTime(date);
setLayout(new GridLayout(2, 1, 0, -3)); setOpaque(false);
label = new JLabel(l);
JPanel datePanel = new JPanel();
datePanel.setLayout(new GridLayout(1, 3, 0, 0));
day = new JSpinner(new SpinnerNumberModel(calendar.get(Calendar.DAY_OF_MONTH),1,31,1));
month = new JSpinner(new SpinnerNumberModel(calendar.get(Calendar.MONTH)+1,1,12,1));
year = new JSpinner(new SpinnerNumberModel(calendar.get(Calendar.YEAR),
            calendar.get(Calendar.YEAR), calendar.get(Calendar.YEAR) + 10,1));
day.addChangeListener(listener);
month.addChangeListener(listener);
year.addChangeListener(listener);
datePanel.add(day); datePanel.add(month); datePanel.add(year);
add(label); add(datePanel);
}
```

Fig. 1. Code snippets of date edit panel from `JVacation`.

parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

For library recommendation, Pareto optimality means that we do not recommended to the developer a single solution. Instead, we want to provide a decision support tool, by showing a variety of solutions, allowing the developer to see a space of trade-offs between the considered objectives.

2.4. Motivating example

To illustrate the need for an approach to recommending useful libraries, let us consider the example of JVacation[4], an open-source stand alone travel-booking-client for travel-agencies. JVacation is implemented in java and its current version adopts only one third party library, `mysql-connector-java`[5]. It is clear that this software system does not effectively take advantage of reusing functionality provided by existing OSS libraries; instead, developers are trying to reinvent the wheel.

Indeed, several library reuse opportunities are missed. For instance, JVacation is implementing from scratch several functionalities such as entering/editing dates and integrating a date edit panel manually into their GUI as illustrated in the code fragment sketched in Fig. 1.

While using the standard APIs `java.util.Calendar` with `javax.swing.JPanel` provided by JDK is helpful, other existing libraries such as JCalendar[6] can be more relevant as it is more specialized for GUI-based software systems. The library JCalender provides an `IDateEditor` for direct date editing, and a button implementing `JDateChooser` for opening a `JCalendar` for graphically selecting date, as well as other calender services that can be useful for this travel-booking software system.

One can notice that JCalendar share several identifiers/terms with JVacation including *Calendar, Date, Panel, Edit, Day, Month, Year*, and so on. This is an indication that they implement simi-

lar functionalities, and therefore JCalendar can be a candidate library that might be relevant for JVacation.

In such situations, recommending libraries based only on the library usage history, such as in [9], would not be enough. In fact, more informative knowledge about the library content is highly required for effective recommendations. On the other hand, using only library usage history might give no chance for new emerging libraries to be recommended and adopted.

The above observations tell us that effective libraries recommendation should make the content of library more explicit and informative as well as libraries usage history.

3. Basic concepts and terminology

This section defines the basic concepts and terminology underlying the proposed approach in this paper: library usage, co-usage, linked-usage, and semantic similarity.

3.1. System and library dependencies

We are concerned with mining large code repositories to extract the '*wisdom from the crowd*'. Specifically in regards to the dependence of third-party libraries in software systems.

Suppose $l \in \mathcal{R}$, where l is a library that belongs to a set of super library repositories $\mathcal{R}$. Examples of popular super repositories that host such libraries include Maven[7] and Nuget[8]. In this study, we are particularly interested in the frequency count of a set of different systems S that use a library l. At this stage we do not consider the granularity of library versions. We define the following terms:

- **Usage.** Usage refers to the frequency count of systems that have used a library l. For a specific system S_i, $S_i \rightarrow l$ shows that the system S_i uses a library l. Formally, let p the total number of available systems, for $l \in \mathcal{R}$, we define:

$$usage(l) = \sum_{i=1}^{p} [S_i \rightarrow l] \qquad (1)$$

- **Co-usage.** The co-usage refers to the frequency count of a pair of libraries used together in one system. Take l_1 and l_2 as two

libraries, then the co-usage is:

$$co\text{-}usage(l_1, l_2) = \sum_{i=1}^{p} [S_i \rightarrow l_1 \wedge S_i \rightarrow l_2] \qquad (2)$$

- **Linked-usage**. The linked-usage metric is a simple average of co-usage score in respect to the usage of different libraries l_1 and l_2. Formally:

$$linked\text{-}usage(l_1, l_2) = \frac{1}{2} \times \left(\frac{co\text{-}usage(l_1, l_2)}{usage(l_1)} + \frac{co\text{-}usage(l_1, l_2)}{usage(l_2)} \right) \qquad (3)$$

The linked-usage is used as a normalized measure of the co-usage between two libraries.

3.2. Semantic similarity

Inspired by information retrieval (IR) technique, our approach uses semantic similarity as a primary mechanism of capturing similar concepts between a given software system's code and a third-party library. Our assumption is that the identifiers/vocabulary of a software element are borrowed from its domain terminology, and thus identifiers can be used as an indicator of source code relatedness. This implies that two software elements could be in the same application domain, i.e., implement similar functionality, if they use similar vocabulary [31]. One of the widely used techniques in IR to calculate semantic similarity between documents is *cosine similarity*.

- **Cosine similarity**. To capture the semantic relatedness between two bags of words A and B, the cosine similarity is defined as the cosine of the angle between both vectors representing A and B in a vector space using *tf-idf* (term frequency-inverse document frequency) model. We interpret term sets as vectors in the n-dimensional vector space, such that each dimension corresponds to the weight of the term (tf-idf) and thus n is the overall number of terms. Formally, the semantic similarity (*Sim*) between A and B corresponds to the cosine similarity (*CS*) of their two weighted vectors $\vec{A}$ and $\vec{B}$ given by Eq. (4)

$$Sim(A, B) = CS(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \times \|\vec{B}\|} = \frac{\sum_{i=1}^{N} w_{a_i} \times w_{b_i}}{\sqrt{\sum_{i=1}^{N} w_{a_i}^2} \times \sqrt{\sum_{i=1}^{N} w_{b_i}^2}} \qquad (4)$$

where w_{a_i} and w_{b_i} are respectively the *tf-idf* weights of the terms a_i and b_i in the bags of words A and B, respectively.

4. Search-based software library recommendation

This section describes our approach that uses SBSE techniques to find and recommend useful third-party software libraries.

4.1. Framework of the approach

The proposed approach, *LibFinder*, is expected to be used with systems that are already implemented. The goal is to keep the developers updated with potentially useful libraries during the maintenance and evolution of their systems, especially with the exponentially growing number of software libraries in open-source repositories. *LibFinder*, is based on two main assumptions for a library could be potentially useful for a given software project (*i*) if it has been commonly used by the crowd with one or more libraries that the project is currently using, and (*ii*) if it uses identical or similar identifiers, i.e., belongs to the same application domain and thereby implements similar functionalities.

In order to realize an approach that meets the requirements stated above, we combine SBSE with MSR techniques. Instead of manually deriving a set of useful library for a software system with a variety of application domains, dedicated search algorithms are employed based on a given set of objectives and constraints. In fact, SBSE techniques allow us to address multi-objective problems as they aim to find the Pareto-optimal set of solutions, as opposed to trying to obtain a single optimal solution. For the library recommendation problem, this would mean that we are interested in a set of solutions where all objectives are compensated and optimized instead of being combined into a single metric, which may not achieve optimality [30].

The overall framework of *LibFinder* is depicted in Fig. 2. Our framework consists of two important steps: (1) data extraction and processing, and (2) data exploration and search process.

4.1.1. Step1: data extraction and processing

This step consists of collecting the necessary data for our recommendation system including systems and library dependencies. We first collected a large set of Java projects and software libraries from GitHub and Maven, respectively (c.f., Section 5.2.1).

System and library dependency. To mine the current usage of these libraries for our linked-usage metrics, our approach is based on `pom.xml` files that define explicitly all the project dependencies with external libraries. To this end, we developed a specific tool (PomWalker[9]) to automatically extract these dependency information from all versions of POM files in a project repository. Fig. 3 shows an xml snippet of the POM file from the *CRest*[10] system under the `CodeGist` project repository. The snippet shows the details including the *groupId, artefactId, version*, etc. of the system as well as the information about all external libraries the system depends on including the library *groupId, artefactId* and *version*.

Identifier-based semantic similarity. To calculate the semantic similarity, we extracted all identifiers for both libraries and systems. For library identifier extraction, as we deal with jar files, i.e., binary code, we used the `asm`[11] library to compile and resolve fully qualified identifier names. A library is regarded as a set of package, class and method names defined in the library. In the case of system identifier extraction, as we deal with systems source code directly, we used the `JavaParser`[12] library to facilitate identifiers extraction for AST generation without compilation. Note that all identifiers related to import and invoke external libraries were excluded during the system's identifiers extraction process.

After identifiers extraction, we performed a lexical analysis to pre-process all the extracted identifiers. Our lexical analysis consists of the four following steps:

1. *Tokenization*. All extracted identifiers are tokenized using a camel case splitter where each identifier is broken down into tokens/terms based on commonly used coding standards.
2. *Filtering*. We use a stop word list to cut-off and filter out all common English words[13] (e.g., *and, the, to*) and reserved words (e.g., *static, string, class*) from the extracted tokens. Typically, these words are irrelevant to the code concept. Such words carry a very low information value and can affect the semantic similarity process negatively as they have no direct relation to the application domain [32].
3. *Lemmatization*. This is a morphological process that transforms each word to its basic form, also called lemma. This process

---

9 https://github.com/raux/PomWalker.
10 https://github.com/codegist/crest.
11 http://asm.ow2.org.
12 http://javaparser.github.io/javaparser/.
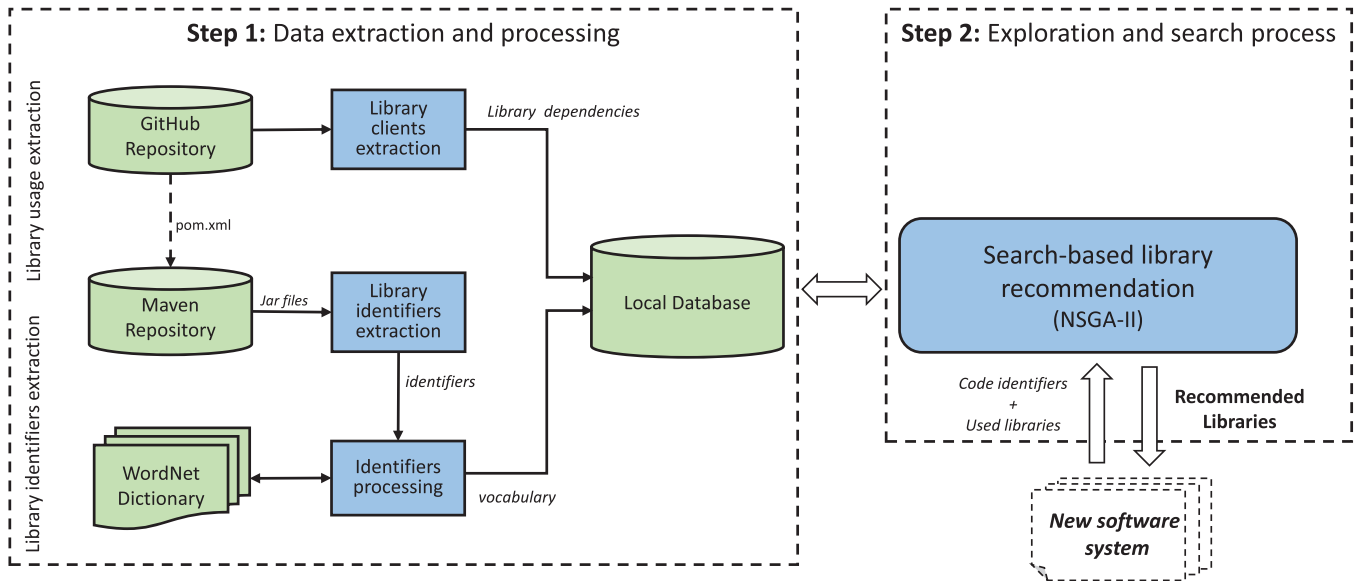13 http://www.textfixer.com/resources/common-english-words.txt.

Fig. 2. Our search-based library recommendation framework *LibFinder*.

aims at reducing a word to its basic form in order to group together the different inflected forms of a basic word so they can be analyzed as a same word. Hence, different forms of words that may have similar meanings are grouped together and handled as identical word. For example, the verb 'to walk' may appear as 'walk', 'walked', 'walks', 'walking'. The base form, 'walk' is then the lemma of all these words. To do so, we use Stanford's CoreNLP[14] to find the base forms of all extracted words.

4. *Vocabulary expansion*. To enhance the effectiveness of the semantic similarity calculation, our approach utilizes `WordNet`[15], a widely used lexical database that groups words into sets of cognitive synonyms (synsets), each representing a distinct concept. We use `WordNet` to enrich and add more informative vocabulary to the extracted bag of words for each library and system. For instance, the word *customer* might be used with different synonyms (e.g., *client, purchaser*, etc.), but pertaining to a common domain concepts.

4.1.2. Step2: data exploration and search process

The collected data from Maven and Github repositories along with all the processed identifiers represents a very large dataset. To facilitate exploring these data, we stored them within a single local data model, so that accessing them becomes easier and faster. Fig. 4 depicts the used data model. A repository consists of a set of systems. Each system depends on set of libraries that are defined through a `pom` file. Each system and each library in the dataset has its vocabulary which consists of a set of terms. A term can have a lemma and a set of synonyms. Formally, let $S = \{s_1, s_2, ...s_n\}$ denote the set of n systems and $L = \{l_1, l_2, ...l_m\}$ denote the set of m libraries. Let $R \subset S \times L$ denote relation that is defined in the pair of one system and one library. The relation R assumes that s_i R l_j is valid if the system s_i depends on the library l_j. The relation l_j R s_i means the library l_j is used by the system s_i. The inverse of the relation R is denoted by R^{-1} and l_j R^{-1} s_i is the same as s_i R l_j. Let S_1 a subset of S let $R(S_1)$ denote the set of those libraries that are dependent on all systems in S_1. Similarly, a subset $R^{-1}(L_1)$ denotes the set of systems that use all the libraries in L_1.

To explore our data model, efficient search techniques are needed. Instead of manually searching for common clients of a particular library and compare the used vocabulary in a particular scenario, dedicated search algorithms can be employed to do so based on a given set of objectives and constraints. One way to efficiently explore this huge search space (collected libraries, their client systems, and the extracted vocabulary, etc.), is to apply dedicated SBSE techniques. Hence, SBSE has proven to be efficient technique in solving several software engineering problems where the number of potential solutions is very large and even infinite [22].

The next section describes in more details how SBSE techniques are adopted for this problem.

4.2. NSGA-II adaptation

Complex decision problems with multiple variables and large search spaces, similarly to this, are well-matched to SBSE, which has proven good performance to provide decision support in several software engineering problems [22]. Our approach adopts SBSE [19], as it provides best practice to formulate software engineering problems in terms of (*i*) computational search algorithm, (*ii*) solution representation, (*iii*) fitness function, (*iv*) change operators, and so on. In the following we describe our SBSE formulation.

4.2.1. Search algorithm

As a search method, we employed a widely used multi-objective evolutionary algorithm (MOEA) namely NSGA-II [21]. NSGA-II tries to ensure diversity to avoid the situation where populations have been filled only with dominating solutions (because of the elitism mechanism, i.e., best solutions are preserved).

Identifying a Pareto front is useful as the software engineer can use the frontier to make a well-informed decision that balances the trade-offs between the different objectives. In our context, one could select recommended libraries achieving the highest semantic similarity, i.e., implementing similar functionality, the highest linked-usage, the lowest recommendation set size, or a compromise among these objectives. Using Pareto optimality, we can plot the set of solutions found to be non-dominating. In the case where there are three objectives, such as ours, this leads to a three dimensional Pareto surface where the developer can go through and make his decision (c.f., Section 5.4).

---

[14] nlp.stanford.edu/software/corenlp.shtml.
[15] wordnet.princeton.edu.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.codegist</groupId>
        <artifactId>codegist-parent</artifactId>
        <version>1.0.2</version>
    </parent>

    <groupId>org.codegist.crest</groupId>
    <artifactId>crest-parent</artifactId>
    <version>2.0.0-RC2-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>CRest Global</name>
    <url>http://crest.codegist.org/</url>
...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.apache.httpcomponents</groupId>
                <artifactId>httpclient</artifactId>
                <version>4.0.1</version>
            </dependency>
            <dependency>
                <groupId>com.sun.xml.bind</groupId>
                <artifactId>jaxb-impl</artifactId>
                <version>2.1</version>
            </dependency>
            <dependency>
                <groupId>log4j</groupId>
                <artifactId>log4j</artifactId>
                <version>1.2.16</version>
            </dependency>
            <dependency>
...
        </dependencies>
</project>
```

Fig. 3. A POM snippet from the `CRest` system describing how Maven manage library dependencies.

4.2.2. Solution representation

A solution in general consists of a number of (decision) variables that are optimized by the respective SBSE algorithm, a number of constraints that need to be fulfilled in order for the solution to be valid, and a number of objective values, one for each of the objective dimensions evaluated by the defined fitness function.

Candidate solutions for our problem are encoded as chromosomes of length n, where each gene represents a candidate third-party library. The length n of a chromosome corresponds to the number of classes in the input software system for which we want to recommend relevant libraries. Note that each class could be assigned either a candidate library or a "NONE" element, i.e., no library is recommended for this specific class. Fig. 5 represents an example of a chromosome that consists of eight recommended libraries. The figure can be interpreted as follows: the library `httpclient` is recommended for the class C1, the library `bcel` is recommended for the classes C2 and C3, `log4j` is rec-

ommended for the class C4, and `guava` is recommended for the classes C5, C6, C7 and C8.

Additionally, a solution candidate may be subjected to a number of constraints in order for the solution to be valid. Depending on the algorithm, invalid solutions may be filtered out completely or may receive a low ranking in relation to the magnitude of the constraint violation. In our approach, each candidate solution should fulfill the two following constraints:

1. A candidate library for a class should be different from the ones already used by that class. To check this constraint, we are based on the *GroupId* (i.e., domain name), without considering the library version. Note that an already used library by a system could be recommended for new classes that are not using it.
2. A candidate solution should not contain similar libraries with different *GroupId*, e.g., `log4j` and `commons-logging`. To check this constraint, we are based on a library-to-library se-
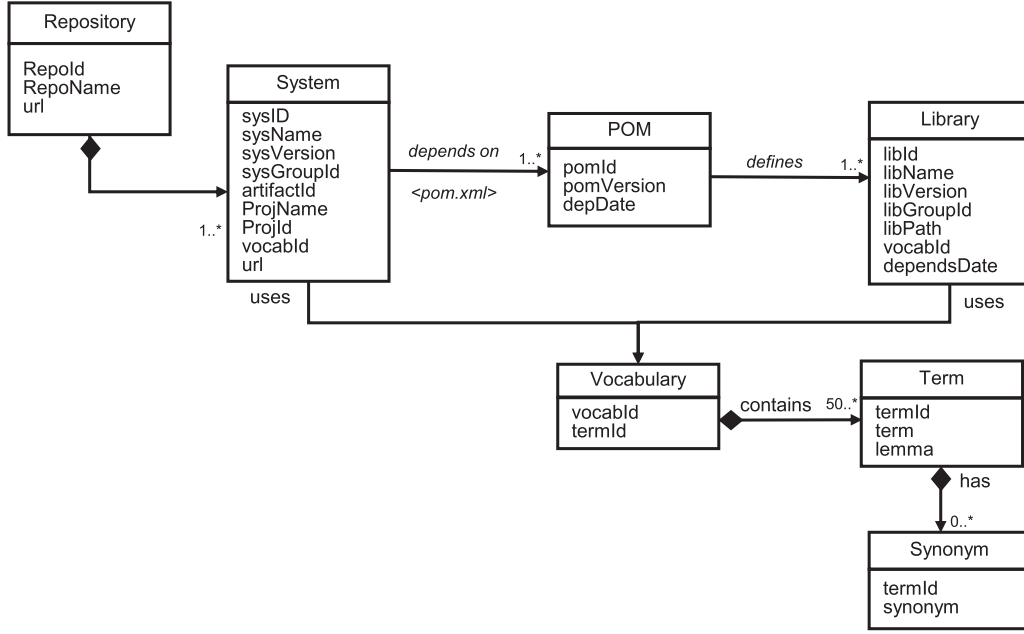
Fig. 4. The data model of the collected dataset.

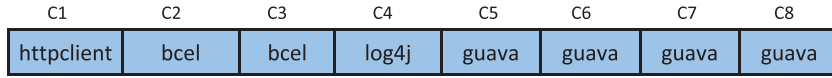| C1	C2	C3	C4	C5	C6	C7	C8
httpclient	bcel	bcel	log4j	guava	guava	guava	guava

Fig. 5. Chromosome representation for a candidate solution.

mantic similarity. Indeed, having similar libraries in a recommendation set does not add value from the developers' perspective as the recommendations will seem redundant and it takes longer for developers to explore [25]. Moreover, based on recent studies [33], some libraries are potentially copies of other libraries, which might lead to undesirable redundancies, security vulnerabilities, and license violations. This constraint assumes that two libraries are similar if the Jaccard similarity between their identifiers is equals or higher than 0.8. Thus, if a class uses `log4j`, this constraint will prevent *LibFinder* from recommending `commons-logging`.

Ranking. The recommended libraries are then ranked according to their frequency count in the chromosome. The more the number of classes for which a library is recommended, the more the library is useful for the system. For instance, for the chromosome depicted in Fig. 5, libraries can be ranked as follows: (1) guava, (2) bcel, (3) httpclient and log4j.

4.2.3. Objective functions

The quality of each candidate solution is defined by a fitness function that evaluates multiple objective and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" than another solution. In our approach, we optimize the following three objectives:

1. **Maximize library linked-usage (LU):** Let L a candidate solution, i.e., chromosome, that consists of a set of libraries $L = \{l_1, ..., l_n\}$ for a given system S that currently uses a set of libraries $L_S = \{l_{s_1}, ..., l_{s_m}\}$. The linked usage is calculated as follows:

$$LU(L) = \sum_{i=1}^{n}\sum_{j=1}^{m} linked\text{-}usage(ls_j, l_i) \times \frac{1}{m} \times \frac{1}{n} \qquad (5)$$

where the function *linked-usage*(ls_j, l_i) is given by Eq. (3).

2. **Maximize semantic similarity (SS):** Let L a candidate solution that consists of a set of libraries $L = \{l_1, ..., l_n\}$ for a given system S that contains n classes where $S = \{c_1, ..., c_n\}$. SS is calculated as follows:

$$SS(L) = \sum_{i=1}^{n} Sim(c_i, l_i) \times \frac{1}{n} \qquad (6)$$

where the function $Sim(c_i, l_i)$ calculates semantic similarity between a class c_i and a library l_i as described in Eq. (4), of course after identifiers tokenization, filtering, lemmatization and vocabulary expansion (c.f., Section 4.1.1).

3. **Minimize the recommendation set size (RSS):** This objective function aims at reducing the number of recommended libraries. Although the number of libraries in our solution representation is propositional to the number of classes in a system, we aim at reducing the number of different libraries in our recommendation set. This objective function is motivated by two reasons. First, we start from the assumption that, typically, developers are unlikely to go through a large recommendation set. Second, adopting a large set of libraries is a costly and error-prone task as it requires an extensive effort from the developer. Indeed, our goal is to get the most from a small set of relevant library recommendations. Formally, let L a candidate solution that consists of a set of libraries $L = \{l_1, ..., l_n\}$, then RSS is given by the following function:

$$RSS(L) = \sum_{i=1}^{n} Unique(l_i) \qquad (7)$$

where the function $Unique(l_i)$ returns 1 if the library l_i is distinct from the previous $i - 1$ libraries in L, 0 otherwise.

The search process is then guided by these three objective functions where *LU* and *SS* are to maximize, while *RSS* is to be minimized.
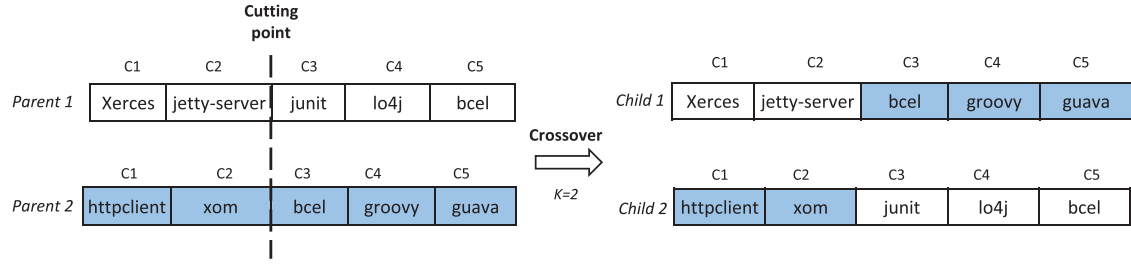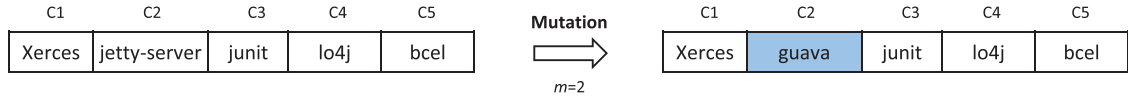
Fig. 6. Crossover operator.

Fig. 7. Mutation operator.

4.2.4. Genetic operators

Population-based search algorithms deploy crossover and mutation operators to improve the fitness of the solutions in the population in each iteration (the initial population is completely random). Change operators such as crossover and mutation aim to drive the search towards near-optimal solutions.

The *crossover* operator is responsible for creating new solutions based on already existing ones, e.g., re-combining solutions into ones. In our adaptation, we use a single, random cut-point crossover to construct offspring solutions. It starts by selecting and splitting at random two-parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. An example of crossover is depicted in Fig. 6.

The *mutation* operator is used to introduce slight, random changes into candidate solutions. This guides the algorithm into areas of the search space that would not be reachable through re-combination alone and avoids the convergence of the population towards a few elite solutions. With library recommendation, we use a mutation operator that picks at random one or more genes (i.e., libraries) from their chromosome and replaces them by other ones from our set of libraries extracted from Maven Central Repository (including the "NONE" element) as shown is Fig. 7.

5. Evaluation

This section first presents experiment design including: (1) research questions required to be addressed, (2) evaluation methods and metrics, and (3) the datasets used in our experiments. Moreover, we also describe the inferential statistical methods used for our experiments and the algorithms parameters tuning and setting.

5.1. Research questions

We design our experiments to address five research questions:

- **RQ1 (Sanity check):** How does the proposed approach performs, in exploring the search space, compared to random search and other existing search algorithms?
- **RQ2 (Accuracy):** How accurate is our proposed approach in recommending libraries to client software systems?
- **RQ3 (Impact of library usage history and semantic similarity):** What is the contribution of each of library usage history and semantic similarity heuristics on the overall performance of *LibFinder*?

- **RQ4 (State-of-the-art comparison):** How does *LibFinder* comparing to existing library recommendation approaches, LibRec [9]?
- **RQ5 (Usefulness):** Is *LibFinder* useful for software developers in a real-world setting?

5.2. Experimental design

We evaluate our approach from two perspectives. First, we evaluate our approach from SBSE perspective by following Harman's guidelines [19]. Then, we evaluate our approach from recommendation system perspective [25].

5.2.1. Dataset

To evaluate the feasibility of our approach on real world scenarios, we carried out an empirical study on real-world Open Source Software (OSS) projects. As we described in Section 4.1.1, our study is based on dataset collected from two popular code repositories Github and Maven. Since github is host to varying projects, to ensure validity of quality github projects, we performed the following filtering on the dataset:

- *Size*. We only included java projects that had more than 1000 commits.
- *Forks*. We only include projects that are unique and not forks of other projects.
- *Maven dependent project*. Our projects only include projects that employ the maven build process (use `pom.xml` configuration file).

Each github repository may contain multiple projects, each which may comprise of several systems. Each of these systems are dependent on a set of maven libraries, that are defined in `apom.xml` file within the project.

For client libraries, we selected the latest release of the library at that period of time. In the beginning we started with 40,936 dependent libraries. However, to remove noise, we filtered out libraries having less than 50 identifiers. This removed specialized libraries that we assume will not be a useful recommendation for other systems. Our dataset resulted in 6083 different maven libraries extracted from unique 32,760 client systems (from 4305 repositories) from Github.

The dataset is a snapshot of the projects procured as of 15th January 2015. As described in Table 1 and 2, our dataset is very diversified as it includes a multitude of libraries and software systems from different application domains and different sizes. We found in our dataset, that the average number of used libraries per system is 10.56 while having an average number of 4729 identifiers

Table 1
Github dataset used in the experiment.

	Dataset
Snapshot date	15th January 2015
# of github repositories	4305
# of github systems	32,760
# of unique dependent libraries	6083

Table 2
Statistics of the maven library dataset size used in the experiment.

	Min	Mean	Median	Max
# of dependent libraries per system	3.00	6.00	10.56	292.00
# of maven identifiers per library	51.0	731.0	4729.0	74,080.0

per library. Our dataset is available for future replications and to encourage future research in the field of automated library recommendation (http://sel.ist.osaka-u.ac.jp/~ali/libRecommendation/).

5.2.2. Evaluation method and metrics

To evaluate our approach, we have performed a ten-fold cross validation on our dataset. The basic idea is to randomly split the data into ten equal-sized parts. Each fold consists of a training data (nine parts) and a test data (the remaining part), then train the recommendation system on the training data and evaluate the recommendations using the test data. Note that only client systems are split into ten parts of size 3276 each.

To answer **RQ1**, we compare our NSGA-II formulation against random search (RS) [34] from two perspectives (1) search algorithm performance, and (2) recommendation system performance, i.e., the performance in solving the problem in hands. The goal is to make sure that there is a need for an intelligent method to explore our search space. Indeed, it is important to compare our search technique to random search, that is if an intelligent search method fails to outperform random search, then the proposed formulation is not adequate [19]. In addition, to justify the adoption of NSGA-II, we compared our approach against two other popular search algorithms namely MOEA/D [35] and IBEA [36]. RQ1 is a sanity check and standard 'baseline' question asked in any attempt at an SBSE formulation [19].

For the search algorithm performance, we provide a quantitative assessment of the performance of each search algorithm in terms of search space exploration. Unlike mono-objective search algorithms, multi-objective evolutionary algorithms return as output a set of *non-dominated* (also called *Pareto optimal*) solutions obtained so far during the search process. To this end, we employ three common performance indicators for evaluating multi-objective optimization algorithms, namely *Hypervolume, Spread* and *Generational Distance* [37].

- *Hypervolume (HV)*: calculates the proportion of the volume covered by the non-dominated solution set returned by the algorithm. A higher HV value means better performance, as it indicates solutions closer to the optimal Pareto front. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity [30,37].
- *Spread (Δ)*: measures the distribution of solutions into a given front. The idea behind the spread indicator is to evaluate diversity among non-dominated solutions. An ideal distribution has zero value for this metric when the solutions are uniformly distributed. An algorithm that achieves a smaller value for Spread can get a better diverse set of non-dominated solutions [37].
- *Generational distance (GD)*: computes the average distance between the set of solutions, S, from the algorithm measured and the reference front RF (also called reference set). The distance

Table 3
Contingency table that accumulates the numbers of true/false positive/negative recommendations.

	Dropped	Non-dropped	Total
Recommended	TP	FP	TP+FP
Non-recommended	FN	TN	FN+TN
Total	TP+FN	FP+TN	N

between *S* and *RF* in an *n* objective space is computed as the average n-dimensional Euclidean distance between each point in S and its nearest neighbouring point in *RF*. GD is a value representing how "far" *S* is from RS (an error measure) [38]. The reference front refers to the set of non-dominated solutions found by the union of all algorithms compared [39]

For further details about the formulation of these performance indicators, interested readers could refer to [30,37] and [38]. Although Hypervolume, Spread and Generational distance are widely used performance indicators when comparing multi-objective algorithms, they ensure the effectiveness of *LibFinder* as a recommendation system by providing a reasonably diversified set of recommendations.

For the performance in solving the problem in hands, i.e., finding relevant library recommendations, we used the top-*k* accuracy, precision and recall on historical datasets. These metrics are commonly used for evaluating recommendation systems in software engineering [14,25,40]. Library ranking is based on the frequency count a library in the recommendation list as described in Section 4.2.2. We conduct a 10-fold cross validation by randomly splitting our datasets D (c.f. Section 5.2.1) into 10 equal parts of size n each. For each fold, we run our approach using a part $P_x \in D$, while training from the 9 other parts P_T where $P_T = D \setminus \{P_x\}$. For each system $S_i \in P_x$, we randomly drop half of the set of its currently used libraries L. Let $L_d \subset L$ the subset of dropped libraries where $|L_d| = \lfloor \frac{|L|}{2} \rfloor$. *LibFinder* will then try to retrieve the dropped libraries in its recommendation set.

- *Top-k accuracy* of our recommendation for a part P_x is calculated as follows.

$$\text{Top}-k\text{accuracy}(P_x) = \frac{\sum_{i=0}^{n-1} isFound(S_i, l \in L_d)}{|P_x|} \times 100\% \quad (8)$$

where the function *isFound(S_i, $l \in L_d$)* returns 1 if at least one dropped library $l \in L_d$ is part of the returned recommendation set, and returns 0 otherwise. For instance, a top-10 accuracy value of 75% indicates that for 75% of the systems, at least one correct dropped library was returned in the top 10 results.
The overall accuracy of our approach corresponds to the average accuracy of the 10 folds.
- *Top-k precision* is calculated from the number of libraries that are either dropped or non-dropped contained in the recommendation set or not. The ground truth is the set of dropped libraries. All possible scores can be arranged in a contingency table (also called the *confusion matrix*) (see Table 3). Once these scores are defined, precision can be calculated. Precision corresponds to *true positive accuracy* and is calculated as the ratio of recommended libraries that are dropped over the total number of recommended libraries as described in Eq. (9).

$$\text{Top}-k\text{ precision} = \frac{TP}{TP+FP} \quad (9)$$

- *Top-k recall* is calculated similarly to precision from the contingency table (Table 3. Recall in corresponds to *true positive rate* and is calculated as the ratio of recommended libraries that

Table 4
Studied industrial systems.

System	Release	#Classes	#Libraries	KLOC
JDI-Ford	v5.8	638	11	247
DROI-Ford	v6.4	786	19	264

are dropped over the total number of dropped libraries as described in Eq. (10).

$$\text{Top}-k \text{ recall} = \frac{TP}{TP+FN} \qquad (10)$$

We thus used accuracy, precision and recall to compare the performance of NSGA-II against IBEA, MOEA/D and RS with top-10 recommendations.

To answer **RQ2**, we evaluated *LibFinder* using our three evaluation measure defined above, accuracy, precision and recall. To better investigate the behavior of *LibFinder*, we conduct the experiment with different k values 1, 2, 4, 6, 8 and 10.

To answer **RQ3**, we assess the effect of our two heuristics for a library to be useful of a client application if (*i*) if it has been commonly used by the crowd with one or more libraries that the project is currently using, and (*ii*) it uses similar identifiers, i.e., implements similar functionalities. We thus investigate the potential of combining both heuristics formulated respectively through the objective functions libraries linked-usage and semantic similarity. To this end, we assess the accuracy, precision and recall results while excluding 1) LU objective function, and 2) SS objective function from *LibFinder*. If our approach is demonstrated to outperform each individual heuristic, then we can claim that our formulation is appropriate. To this end, we used NSGA-II with same adaptation described in Section 4.2 to investigate each combination LU with RSS and SS with RSS objectives functions.

To answer **RQ4**, we compare our approach with a recent state-of-art approach called *LibRec* [9]. *LibRec* is library recommendation approach that combines association rule mining and collaborative filtering. The association rule mining component recommends libraries based on library usage history. The collaborative filtering component recommends libraries based on those that are used by other similar projects. However, the library content is not considered. To the best of our knowledge, this is the only existing approach for library recommendation. To conduct a comparative study, we replicated the *LibRec* on the same collected dataset (c.f., Section 5.2.1) using the same metrics top-k accuracy, *precision* and *recall*.

To answer **RQ5**, we conducted a qualitative evaluation to better evaluate *LibFinder* with developers in practice. Indeed, it is important to qualitatively evaluate the relevance and usefulness of the recommended libraries from developer's perspective. To this end, we performed a qualitative evaluation with two large industrial projects provided by our industrial partner, the Ford Motor Company as described in Table 4. The first project is a marketing return on investment tool, called MROI, used by the marketing department of Ford to predict the sales of cars based on the demand, dealers' information, advertisements, etc. The tool can collect, analyze and synthesize a variety of data types and sources related to customers and dealers. It was implemented over a period of more than eight years and frequently changed to add and remove new/redundant features. The second project is a Java-based software system, namely JDI, that helps the Ford Motor Company to create the best schedule of orders from the dealers based on thousands of business constraints. This system is also used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to match the expectations of their customers. JDI is highly structured and software developers have developed several versions of it at Ford over the past 10 years. Due to the

importance of the application and the high number of updates performed on both systems, it is critical to ensure that they reuse and adopt high quality software libraries so to reduce the time required by developers to introduce new features in the future and maintain high quality software.

Our experiment is based on a survey to collect the feedback of Ford developers about *LibFinder*'s recommendations. One of the advantages of this industrial validation is the participation of the original/current developers of a system in the evaluation of the recommended libraries. The experiment is conducted as follows. We executed *LibFinder* on each of JDI-Ford and DROI-Ford using our library dataset (cf. Section 5.2.1) and selected the top 10 recommended libraries. Thereafter, the software engineers from Ford were asked to manually evaluate each of the recommended libraries by answering the following question:

Is the recommended library useful for the implementation of your system?

For each library, the participants were asked to assign a score using a five-point Likert scale [41] to express their level of agreement: 1: *Not useful at all*; 2: *some what useful*; 3: *Useful*; and 4: *Very useful*; and 5: *Extremely useful*. Furthermore, the developers where asked to comment on their ratings by an additional (optional) question "*If so, are you willing to adopt this library in your code?*"

Subjects. Our study involved 8 industrial developers from the Ford Motor Company. Prior the study, the participants were invited to fill a pre-questionnaire about their experience, with Java programming, software library reuse and their experience with the subject systems. The eight subjects claimed they are familiar with Java programming, software maintenance activities with an experience ranging from 8 to 17 years. All claimed to frequently using third-party software libraries in their projects. The eight participants were selected based on having similar development skills, their motivations to participate in the survey and their availability. We organized the participants into two equal groups, each consists of four developers. The first group contains developers who are part of the development team of JDI, while the second group consists of developers who are current developers of the DROI system, and three of them are already part of the original developers' team. The first group evaluated the recommendations for JDI, while the second group assessed the DROI system. The survey is completed anonymously thus ensuring the participants confidentiality.

In a first meeting with the participants, we explained the overall purpose of the study, without giving any details, concerning the libraries and the method used for the recommendation. Following, we gave to each participant a document that contained the following information (a) each recommended libraries, (b) a brief description of the main features of the library and its website url and Maven link, and (c) the list of classes for which the library is recommended. In a second meeting with each of the developers, we collected the documents and we analyzed the developers' feedback.

5.2.3. Inferential statistical test methods used

Due to the stochastic nature of the used algorithms, they may produce slightly different results when applied to the same problem instance over different runs. To cope with this stochastic nature, the use of a rigorous statistical testing is essential to provide support to the conclusions derived from analyzing such data [42]. To this end, we used the Wilcoxon Signed Rank test in a pairwise fashion [43] in order to detect significant performance differences between the algorithms under comparison. 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 themselves. We set the confidence limit, α, at 0.05. In these settings,

each experiment is repeated 30 times, for each algorithm and for each fold. The obtained results are subsequently statistically analyzed with the aim to compare our NSGA-II approach to MOEA/D, IBEA, as well as random search (RS).

While Wilcoxon Signed Rank test allows verifying whether the results are statistically different or not, it does not give any idea about the difference magnitude. To this end, we assess the effect size based on Cohen's d [43]. The effect size is considered: (1) small if $0.2 \leq d < 0.5$, (2) medium if $0.5 \leq d < 0.8$, or (3) high if $d \geq 0.8$.

5.2.4. Parameter tuning and setting

An important aspect for metaheuristic search algorithms lies in the parameters tuning and selection, which is necessary to ensure not only fair comparison, but also for potential replication. The initial population/solution of NSGA-II, MOEA/D, IBEA, and RS are completely random. The stopping criterion is when the maximum number of fitness evaluations, set to 350,000, is reached. After several trial runs of the simulation, the parameter values of the four algorithms are fixed to 100 solutions per population and 3500 iterations. There are no general rules to determine these parameters, and thus, we set the combination of parameter values by trial-and-error method, which is commonly used in the SBSE community [44,45].

For the variation operators, we set crossover probability at 0.9 and a mutation one at 0.4. We used a high mutation rate since we are employing an elitist schema with diverse library contents and co-usage. In fact, as noted by Cohen [43], elitism may encourage premature convergence to occur, e.g., reaching the last iteration with solutions having high *LU* and *SS*, but belongs to only one application domain. For instance, returning all recommended libraries related to xml parsers, which seems redundant and do not add value from the users perspectives. In order to avoid such a problem, in each generation, we emphasize the diversity of the population by means of the high mutation rate.

Note that, while for RQ1, we compare our resulted Pareto solutions of each algorithm, for RQs 2–5, we need only one solution for our automated evaluation. As NSGA-II returns a set of optimal solutions instead of single one, the developer can choose one of them according to his preference; however at least for our evaluation as we seek a fair comparison, we need to automatically pick up one single solution from the Pareto front and then compare it with LibRec. To address this issue, we proposed a technique that aims at automatically selecting the nearest solution to a 'reference' point from the Pareto front that represents the desired values of the objective function. Ideally, this reference point has the optimum values of each objective function, i.e., a score of 1 for the linked-usage (LU), 1 for the semantic similarity (SS), and 1 for the recommendation set size (RSS), after normalization within the interval [0,1] (c.f. Fig. 12). Our recommended solution, *RecSol*, corresponds to the nearest point p_i from the Pareto front *PF* to the 'reference' point in terms of Euclidean distance. *RecSol* is calculated as follows:

$$RecSol = \min_{\forall p_i \in PF} \sqrt{(1 - LU(p_i))^2 + (1 - SS(p_i))^2 + (1 - RSS(p_i))^2}$$

(11)

5.3. Results

In this section, we present the results of our empirical evaluation with respect to our research questions RQs 1–4 set out in Section 5.1.

Results for RQ1. For RQ1, we compared NSGA-II against RS, MOEA/D, and IBEA, using the same objective functions, solution representation, and change operators. We describe the obtained results in terms of Hypervolume (HV), Spread (Δ), Generational Distance (GD), Accuracy@10, Precision@10 and Recall@10.

Fig. 8 and Table 5 present the results of the significance and effect size tests through 30 independent runs (3 runs for each fold) of each search algorithm. The higher the HV, accuracy@10, precision@10, recall@10, and the lower the Δ and GD values, the more likely the recommendation results are better. We observe that RS results are generally poor, whereas MOEA/D, IBEA and NSGA-II obtain higher results. This provides evidence that there is a need for an intelligent search technique to provide better library recommendation results. Furthermore, for all six quality indicators (HV, Δ, GD, accuracy, precision and recall), the Wilcoxon test results showed that NSGA-II achieves significantly better performance than MOEA/D, IBEA and RS with high effect size. This provides evidence that NSGA-II is effective in finding a well-converged and diversified set of Pareto-optimal solutions, i.e., recommended libraries. For the Δ, it is also desired that a multi-objective evolutionary algorithm maintains a good spread of returned solutions. This gives more options to the developer on which library can be useful for his code. The Wilcoxon test results showed that for 17 out of 18 experiments (6 performance indicators, and 3 pairs of algorithms), the quality indicators achieved by NSGA-II were significantly better than those of random search with a Cohen effect size "high". Only for the Spread indicator, a Cohen effect size of "meduim" is achieved against MOEA/D.

As part of our sanity check, we also studied the extent to which the linked usage (LU) and semantic similarity objective (SS) functions are conflicting. Indeed, if the two objectives are not conflicting, i.e., correlated, then the problem should be formulated as a single objective optimization problem. Fig. 9 presents the results of studying the conflict relation between our LU and SS objectives. To this end, we execute a mono-objective GA maximizing one of the objectives and we study the behavior of the second objective by recording its values over 100 independent runs. From Fig. 9a, we observe that the maximization of LU objective function does not cause any maximization or minimization of the SS objective function as their correlation is less than -0.21. Similar phenomenon could be seen in Fig. 9b where a genetic algorithm was used to maximize the SS objective function while recording the behavior of the LU values. The observed correlation was less than 0.18. Based on this finding, we conjuncture that both LU and SS objectives are conflicting.

To sum up, we conclude that there is a compelling evidence that our multi-objective formulation using NSGA-II is adequate (this answers RQ1).

Results for RQ2. Table 6 reports the top-1, top-2, top-4, top-6, top-8 and top-10 accuracy, recall and precision results for each of the ten folds obtained over 30 runs of *LibFinder* using NSGA-II.

For the top-k accuracy, we observe that *LibFinder* achieves the top-10 accuracy with 91.8% on average over the ten folds. The lowest top-10 accuracy score is 91.26% while the best one is 92.45%. This provides evidence that the accuracy of our approach is relatively stable over the ten folds for 30 independent runs. The average lowest accuracy score was in the top-1 with a score of 63.34%. This indicates that the chance to have at least one of the dropped libraries in the first rank is 0.63. Furthermore, we found in most cases where our approach fails in finding at least one of the dropped libraries that it relies on libraries from the same category, i.e., application domain. For instance, instead of recommending/retrieving the dropped library `commons-logging`, most of the cases our recommendation set includes the competitor logging library `log4j`. This may due to the popularity of `log4j` manifested by its growing usage score `commons-logging` and `log4j` have respectively 3587 and 6254 usage scores in Maven Central Repository, as of November 24th, 2015. in Maven while having similar identifiers.

For precision and recall, *LibFinder* achieves also good results. *LibFinder* achieves the top-10 precision of 51.06% on average over
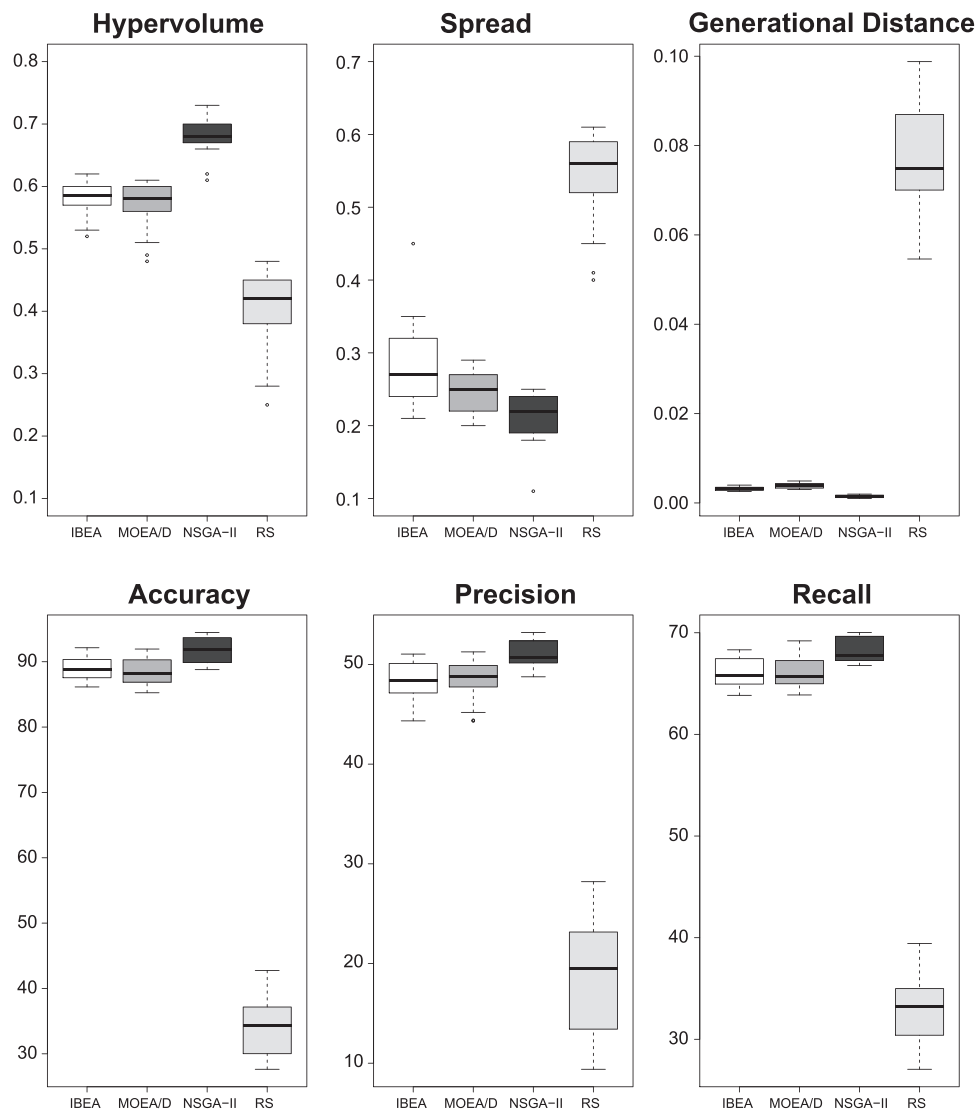
Fig. 8. Boxplots for the Hypervolume, Spread, Generational distance, accuracy@10, precision@10 and recall@10 performance indicators for NSGA-II, IBEA, MOEA/D and RS.

Table 5
Statistical significance p-value ($\alpha = 0,05$) and effect size comparison results of NSGA-II against MOEA/D, IBEA and RS for the hypervolume, spread and generational distance, accuracy@10, precision@10 and recall@10.

Metric	Stat.	NSGA-II vs MOEA/D	NSGA-II vs IBEA	NSGA-II vs RS
Hypervolume	*p*-value	7.27e−10	8.071e−10	7.68e−10
	effect size	high	high	high
Spread	*p*-value	1.239e−06	4.933e−08	7.656e−10
	effect size	medium	high	high
Genergational Distance	*p*-value	9.313e−10	9.313e−10	9.313e−10
	effect size	high	high	high
Accuracy@10	*p*-value	6.918e−06	5.145e−06	1.863e−09
	effect size	high	high	high
Precision@10	*p*-value	9.22e−06	2.716e−05	1.863e−09
	effect size	high	high	high
Recall@10	*p*-value	1.419e−06	1.563e−05	1.863e−09
	effect size	high	high	high

the ten folds. The highest precision score was achieved by the top-1 with a precision of 61.34%. From the recall side, *LibFinder* achieves better results with a top-10 recall of 68,13% and a top-1 recall of 47.94%. However, still precision and recall scores relatively lower than the accuracy. In fact, an important point to highlight is that these two metrics implies that recommended libraries that

the system has not used (i.e., not in the dropped set) are uninteresting or useless. This assumption is not always appropriate in practice. That is, a system might not use a specific library for many reasons mainly when (*i*) the system's developers are not aware by such a library especially with the exponentially growing number of
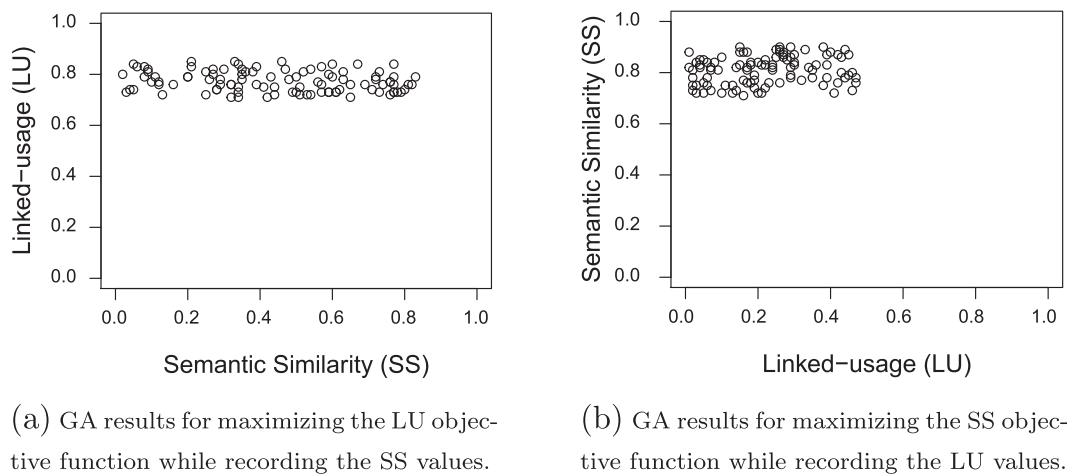
(a) GA results for maximizing the LU objective function while recording the SS values.

(b) GA results for maximizing the SS objective function while recording the LU values.

Fig. 9. Relation study between the Linked Usage (LU) and Semantic Similarity (SS) objective functions.

Table 6
Obtained top-k accuracy, Precision and Recall median results for of the 10-fold cross-validation obtained through 30 independent runs of NSGA-II.

Fold	Accuracy@k						Precision@k						Recall@k					
	top-1	top-2	top-4	top-6	top-8	top-10	top-1	top-2	top-4	top-6	top-8	top-10	top-1	top-2	top-4	top-6	top-8	top-10
1	62,52	69,69	81,17	87,56	90,16	92,45	62,52	58,23	58,16	56,33	53,63	51,87	48,16	57,47	56,02	61,58	63,01	67,16
2	63,19	70,98	82,36	86,15	89,16	91,45	59,19	58,26	56,29	55,03	53,18	50,98	49,03	56,87	55,16	60,33	62,47	69,56
3	62,15	69,02	80,21	87,24	90,45	91,46	62,15	60,48	58,36	56,17	52,33	49,87	48,44	55,13	56,31	61,04	63,09	68,36
4	60,43	70,78	81,06	88,62	91,03	92,13	60,43	59,15	57,33	56,04	53,1	51,22	48,17	54,18	56,09	60,63	62,18	67,12
5	63,25	70,21	81,71	88,36	90,43	92,36	59,25	58,03	58,04	56,14	53,36	50,13	49,11	53,24	55,89	59,49	61,34	68,03
6	65,47	69,8	81,53	87,56	90,56	91,26	62,47	61,62	57,98	55,32	53,24	51,86	47,64	54,48	56,55	60,44	62,07	68,17
7	63,41	68,48	82,09	87,28	89,98	91,36	60,41	59,22	58,68	56,87	52,11	49,89	48,21	51,17	55,92	61,3	62,29	68,43
8	64,87	70,47	81,39	87,14	90,26	91,89	62,87	60,39	59,74	57,33	53,49	51,76	47,29	51,89	56,67	60,39	64,33	67,89
9	62,45	70,33	81,69	88,47	91,03	92,02	62,45	59,11	57,71	56,09	52,78	51,44	46,33	52,03	56,58	59,11	63,08	68,11
10	65,66	69,15	80,36	88,36	91,27	92,16	61,66	59,17	58,6	56,54	52,07	51,6	47,04	54,86	55,77	60,03	61,37	68,51
Average	63,34	69,89	81,35	87,67	90,43	**91,85**	**61,34**	59,36	58,08	56,186	52,929	51,06	47,94	54,13	56,09	60,43	62,52	**68,13**

Table 7
Effect of each individual heuristic on the accuracy, precision and recall results with top-10 rank.

Approach	Accuracy@10	Precision@10	Recall@10
Linked-usage	63%	38%	34%
Semantic Similarity	61%	41%	37%
LibFinder	92%	51%	68%

available libraries in code repositories, or (*ii*) due to some resource constraints, e.g., budget limitations or deadline pressure.

To conclude, the obtained accuracy, precision and recall results indicate that leveraging a history of library usage from large source code repositories, with semantic similarity embodied in library identifiers can accurately recommend useful software libraries for client software systems.

Results for RQ3. Table 7 reports the median accuracy, precision and recall values of each of linked-usage and semantic similarity heuristics with top-10 recommendations. The aim is to investigate the effectiveness of our two heuristics that are formulated through LU and SS objective functions and how well they work separately. As shown in the table, the linked-usage reaches an accuracy score of 63.1%, whereas the semantic similarity reaches 61.26%. However, using each heuristic separately is still far from the accuracy score of *LibFinder* (92%). Similarly, our approach reaches a precision score of 51% while having only 38% and 41% for each individual heuristic, linked usage and semantic similarity respectively. Recall is also following the same trend as the accuracy by a score of 68% for *LibFinder* while only 34% and 37% are achieved by linked usage and semantic similarity respectively. This finding suggests that both linked-usage and context similarity should be balanced together for an effective library recommendation.

Results for RQ4. Table 8 presents the results of top-1, top-2, top-4, top-8 and top-10 accuracy, precision and recall of our approach, *LibFinder*, and a state-of-the-art approach *LibRec,Thung2013librec. LibFinder* achieves much better results for all 18 experiments (3 metrics and 6 top-k values). For instance, *LibFinder* achieves the top-10 with an accuracy of 92% and the top-1 with an accuracy of 63% while *LibRec* achieves only 73% and 12% for top-10 and top-1 respectively. Similarly for the precision and recall scores, *LinFinder* achieves the top-10 with 51% of precision and 68% of recall while only 23% of precision and 46% of recall was achieved by *LibRec*. This indicates that leveraging only a history of library usage is not enough for recommending relevant libraries. Thus a content-based recommendation should be balanced with usage history for better recommendations. In fact, a history-based approach (e.g., collaborative filtering method and association rule mining methods) provides recommendations for a user based on some unknown users who might have the same taste, while a content-based method provide recommendations based on what item's features the users need. Furthermore, unlike pure history-based approaches such as *LibRec*, new libraries can be recommended by a content-based method before being used by a substantial number of software projects. Indeed, *LibRef* is not able to recommend libraries to projects that only use a small number of li-

Table 8
Comparison results of our approach *LibFinder* against *LibRec* in terms of accuracy, precision and recall scores.

rank@k	Approach	accuracy@k	precision@k	recall@k
top-1	LibFinder	63	61	48
	LibRec	12	41	28
top-2	LibFinder	70	59	54
	LibRec	24	36	31
top-4	LibFinder	81	58	56
	LibRec	51	31	39
top-6	LibFinder	88	56	60
	LibRec	58	29	40
top-8	LibFinder	90	53	63
	LibRec	64	27	42
top-10	LibFinder	92	51	68
	LibRec	73	23	46
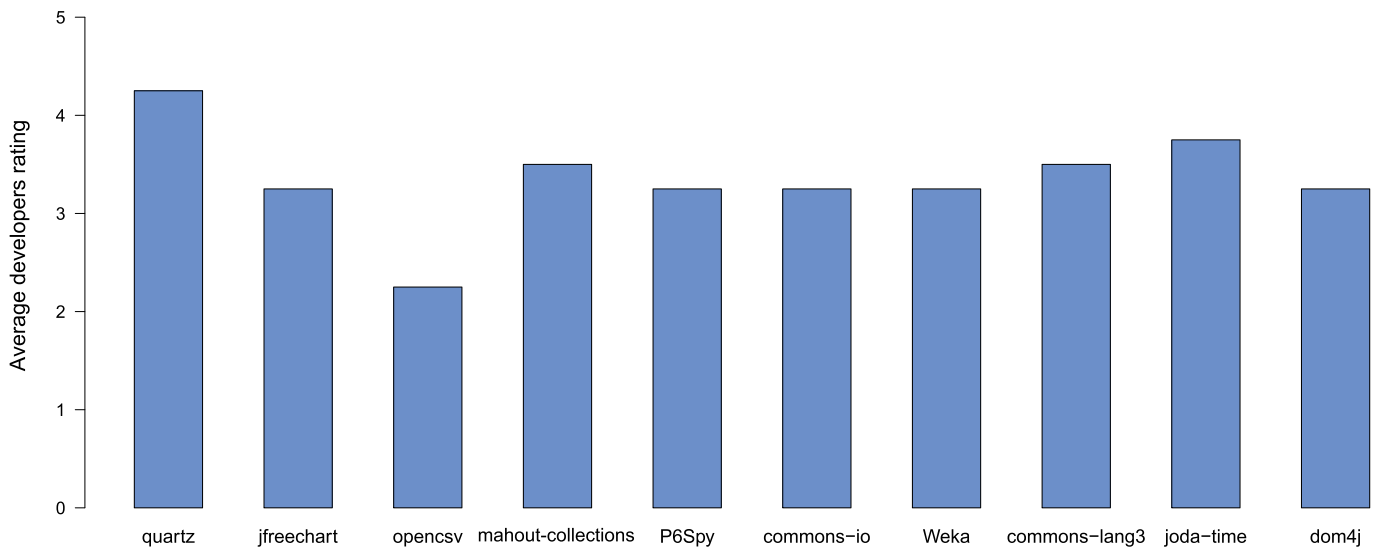
Fig. 10. Average developers ratings for the 10 recommended libraries by *LibFinder* for the system JDI-Ford.

braries or do not use any libraries at all. Thus, we believe that finding a trade-off between usage history and content-based method, such as *LibFinder* is a suitable way in formulation library recommendation problem.

Results for RQ5. Fig. 10 reports the developer's rating results of the recommended libraries for the system JDI-Ford. For the ten recommended libraries, the developers ratings were ranging from 2.25 to 4.25. The highly rated library was *Quartz*, a richly-featured, enterprise-category library, that solves complex and small schedules, and get an average rating of 4.25 from the 4 developers. Features in Quartz include JTA clustering and transaction, real time management and monitoring. The four developers claimed that *Quartz* is a relevant library that could be useful for their current implementation, especially for creating and scheduling the orders with their dealers. On the other hand, while *opencsv*, a simple library for reading and writing CSV, was ranked lowest (an average of 2.25) as it was only recommended for few classes that are not related to core functionalities in JDI-Ford. Indeed, developers found it not as useful comparing to their current in-house code. Another recommended library was *Joda-time* which also get a high rating of 3.75 as it provides a quality replacement for the Java date and time classes with more useful and efficient features, relieving the developers from the burden of implementing date related features from scratch or based on the basic features provided by Java Development Kit (JDK).

Similarly, most recommended libraries for DROI-Ford get high ratings as sketched in Fig. 11. The library *mahout-math*, a high per-

formance scientific and technical computing data structures and methods library, gets the highest rating reaching 3.75 on average by the four developers. Same rating was obtained for *Guava*, a suite of core and expanded libraries that include utility classes, google's collections, io classes, and much much more useful utilities, get a high ranking of 3.75. However, *slf4j*, which a simple logging facade library which serves as a simple facade or abstraction for various logging frameworks, get the lowest score of 1.5 for this system. Overall, the eight developers who participated in the survey was generally satisfied with most recommended libraries based on their average score and their feedback. In both systems, we have two cases, *opencsv* for JDI-Ford and *slf4j* for DROI-Ford for which some of the developers was not satisfied and rated them by a score 1 (*Not useful at all*).

To better investigate the usefulness of the recommended libraries, we asked the eight developers if they are willing to consider adopting the libraries that they judged as 'useful' in the next releases of their system. Most of the developers (5 out of 8) expressed a high interest to adopt at least 4 libraries including *mahout-math, quartz, guava*, and *pdfbox*, for the coming releases. Their main reservations about such adoption are 1) the quality of the system after using the library and how to make sure that such a library will improve their code quality and 2) the time required to understand the entire library. Indeed, while *Joda-time* was highly rated, the developers did not express a high interest to adopt it for the short term as it might require additional efforts to incorporate several changes with several classes in the code. However, one
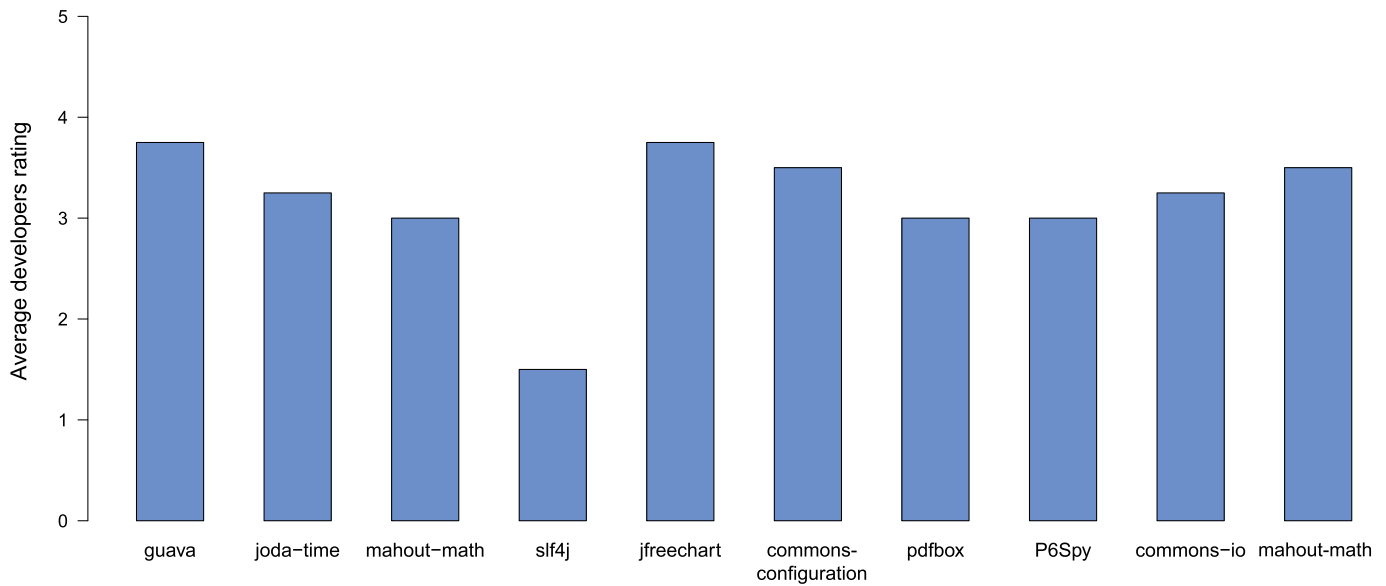
Fig. 11. Average developers ratings for the 10 recommended libraries by *LibFinder* for the system DROI-Ford.

of the developers claimed that they can keep their current code, but use *Joda-time* for the new features and changes in the next releases. Most of developers from both JDI-Ford and DROI-Ford claimed that this is a valuable reuse opportunity that they missed in the initial design of their system.

Another situation highlighted in some comments was that most classes for which *Joda-time* and *opencsv* was recommended have very low change frequency during the last two years. Thus, developers would adopt libraries for more frequently changed and active classes in the project. This gives us interesting insights to incorporate the change frequency of classes in *LibFinder* to improve its recommendation usefulness in practice.

5.4. Discussions

In this section, we further discuss related aspects to our approach including library selection in terms of trade-off, .

5.4.1. Library selection

To help developers on selecting a library from the set of recommendations, *LibFinder* provides a user friendly manner to select recommended libraries through 3D visualizations of the Pareto front, as shown in Fig. 12. This visualization shows the trade offs between the three objectives linked-usage, semantic similarity, and the recommendation set size. Each point in the Pareto surface in Fig. 12b represents a recommended solution, each solution comprises a set of libraries that will be sorted according to the frequency of each library in the solution (c.f., Section 4.2.2).

The distributions of these recommended solutions on the Pareto from gives more insights. Hence, if a developer is interested in new emerging libraries that implement similar functionality to his system, then he should focus his attention on the back corner of the Pareto surface, i.e., high semantic similarity score. For the developer who seeks to reduce the library integration costs and incompatibility risks, he needs rather to focus on the top corner of the Pareto surface as it recommends libraries that are commonly used with his current system's libraries, i.e., high linked-usage score. If the developer seeks to check a small set of recommended libraries, the he should focus on the left side of the Pareto surface, or the right side for a variety of recommended libraries, i.e., high RSS score, otherwise. If the developer seeks a kind of trade-off between all objectives, he should then focus his attention on the middle

part of the Pareto surface. Hence, as the three objectives are conflicting, maximizing semantic similarity could be possible but with cost of scarifying by some linked-usage.

For instance, following our motivating example JVacation which is using only the mysql-connector-java library, the recommended solutions comprises {JCalendar, glazedlists, mail, cxf-api, jetty-http, berkeleydb, javaosc-core, webapp-runner}. Most of them are likely to be useful for JVacation. For instance, JCalendar is likely to be good candidate as described in Section 2.4. Glazedlists is an event-driven lists for dynamically filtered and sorted tables. It is typically co-used with mysql-connector-java to process the results of SQL queries. In addition, mail can be useful for JVacation to directly contact customers, send invoices and so on. Berkeley is a high performance, transactional storage engine for Java applications. This makes sense as JVacation is a database related software system. Other recommendations such as cxf-api, jetty-http and webapp-runner are more related to web servers, building and developing services.

5.4.2. Developer insights on third-party library recommendation

The decision to keep in house code or reuse third-party libraries could be a delicate decision to be made by the developer in her individual context. It is important to get some insights from a developer's perspective on the situations where it is useful to adopt a recommended third-party library. To this end, we conducted a think-aloud survey with 17 developers from Ford including the 8 developers who participated in our experiment conducted in RQ5. The participants were first asked to answer the following question:

When do you think it is appropriate to apply a recommended third-party library to your code?

Fig. 13 shows the different answers collected from the survey. A total of 5 out of 17 developers (29%) answered that they would be interested in adopting a recommended library as replacement of buggy code. Four developers answered that if the overall quality of the system will be improved when the recommended library is adopted, then it is worth. Similarly, a total of four developers answered that they are willing to adopt a recommended library if it provides additional features that are useful for the implementation of their system, especially if their current solution is not extendible
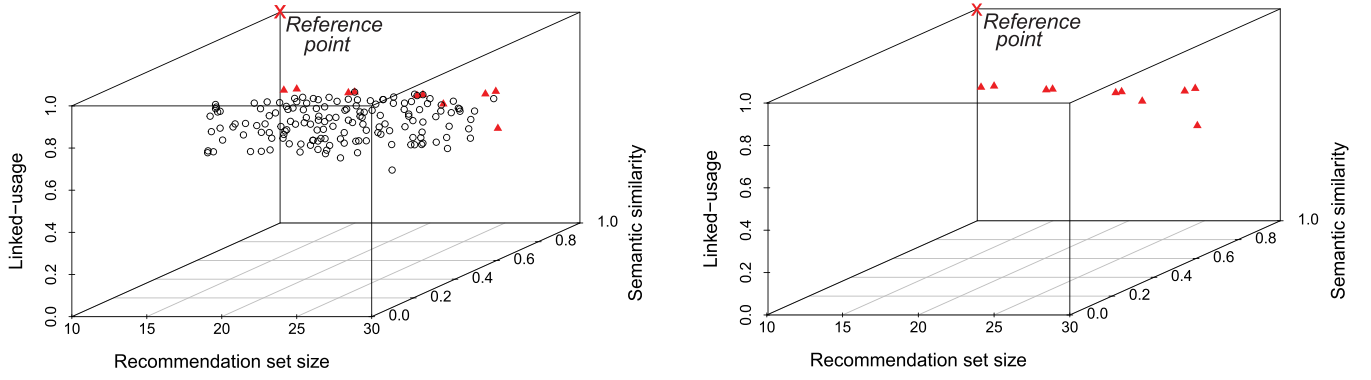
(a) Last iteration solutions (depicted by the circles) and Pareto surface (depicted by the red tingles) obtained by NSGA-II.

(b) Pareto surface (depicted by the tingles) obtained by NSGA-II.

Fig. 12. Example of Pareto optimal solutions obtained by *LibFinder* for JVacation.
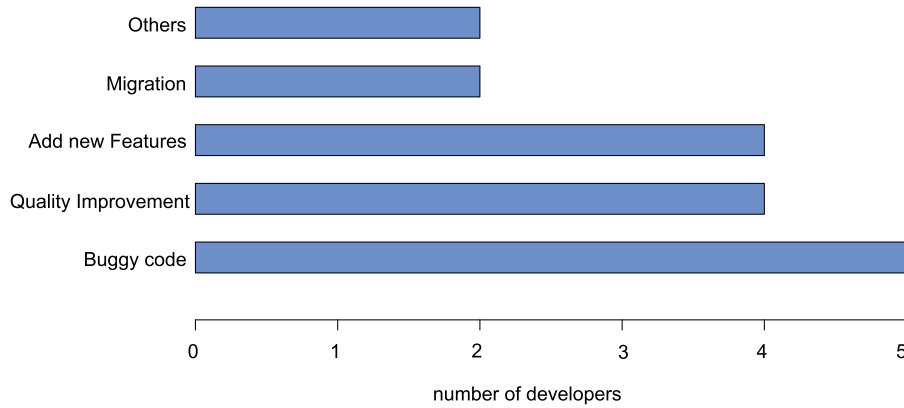

Fig. 13. Answers to the question: *when do you think it is appropriate to apply a recommended third-party library to your code?*

enough. Furthermore, three out of the 17 developers answered that they would adopt a library if it will support software migration to provide portable and reusable code. Some other answers were also related to the library popularity and its API stability/quality. Overall, all developers expressed a hight interest to adopt third-party libraries in their code for different motivations and reasons as described in Fig. 13. This justifies the exponentially increasing trend in adopting third-party software libraries from open source repositories.

These results have actionable insights. For example, it would be interesting to consider a profile-based library recommendation system to focus the recommendations on buggy classes, or smelly/low quality classes, or on some packages related to particular features that the developer is interested in.

Moreover, as a second part of the survey, we wanted to assess how realistic is our problem formulation for the library recommendation, i.e., do the three defined objective functions match the developers high-level expectations. Our second part of the survey consists of three questions about the meaningfulness of our objective functions. It consists of the three following questions:

- **Q1.** When searching for a relevant library, do you prefer to select/find libraries that are related/commonly used with your current adopted libraries?
- **Q2.** When searching for a relevant library, do you prefer to select/find libraries that belong to the same application domain as your implemented features, e.g., have textual similarity?
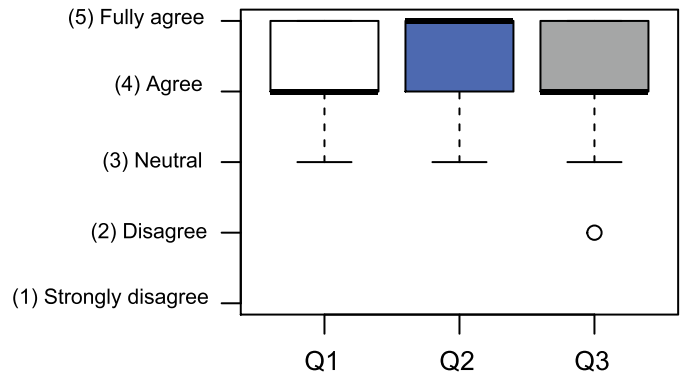

Fig. 14. Developers answers for the three questions Q1, Q2 and Q3.

- **Q3.** When searching for a relevant library, do you look for a minimum number of libraries to adapt such as libraries relevant for multiple tasks/features to integrate?

Participants were asked to answer each of the three questions by following a five-point Likert scale to express their level of agreement: 1: *Strongly disagree*, 2: *Disagree*, 3: *Neutral*, 4: *Agree*, 5: *Fully agree*.

Fig. 14 reports the obtained results from the survey. On average, a score of 4.17, 4.52, and 4.11 were obtained for Q1, Q2 and
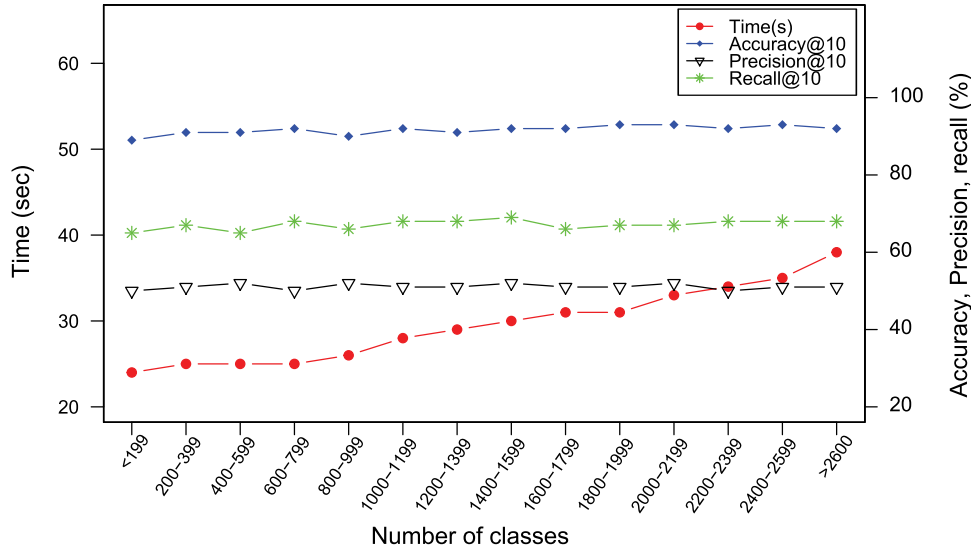
Fig. 15. Scalability of *LibFinder* with respect to the systems size.

Q3, respectively. As shown in the figure, the ratings distribution is within the range 3–5 with a median of 4 for Q1 and Q3, and 5 for Q3. Indeed, since library recommendation is a subjective decision process, it is normal that not all the developers have the same opinion. Thus, it is important to study the level of agreement between developers. To address this issue, we evaluated the level of agreement using Fleiss's Kappa coefficient κ [46], which measures to what extent the developers agree when answering to the three questions about each of our defined objectives. The Kappa coefficient assessments is 0.63, which is characterized as "*substantial agreement*" by Landis and Koch [47]. This obtained score makes us more confident that our defined objectives for library recommendation makes sense from software developer's perspective. However, this finding does not necessary mean that these three objectives are the only objectives that should be considered for library recommendation. Indeed, some of the developers suggested to involve library popularity and quality in the recommendation process. As part of our future work, we plan to personalize *LibFinder* with the defined objectives based on the developer's profile/preferences and make it a more interactive process to learn from the developer decision in which situations he accept or reject a recommended library.

It is worth notice that *LibFinder* allows the developers to give more importance to one objective over the other objectives using the pareto visualization as described in Section 5.4.1. This visualization supports to the developers to select their libraries based on their preferences between all objectives. Furthermore, *LibFinder* allows to easily remove or add new objectives to guide the search space based on the developer requirements/preferences.

5.4.3. LibFinder scalability

It is important to assess the scalability of our approach, as scalability is widely considered as one of the key issues for software engineering research and development [48]. Indeed, there is a pressing need for scalable solutions to Software Engineering problems. To evaluate the scalability of the performance of our approach for systems of increasing size, we report the results of *LibFinder* per system size in terms of number of classes. The scalability assessment of *LibFinder* is only concerned with the exploration and search process (Step 1 in Fig. 2 is not taken into account for the scalability). The results of the experiment are depicted in Fig. 15. We noticed that when the size of the system increase, the CPU time is still relatively stable as it ranges from 24 s to 35 s for

Table 9
Standard deviation of the obtained accuracy@k, precision@k and recall@k results over the 10 folds.

Top-k	Standard deviation		
	Accuracy@k	Precision@k	Recall@k
Top-1	1.537	1.329	0.83
Top-2	0.771	1.088	1.982
Top-4	0.651	0.859	0.433
Top-6	0.738	0.635	0.727
Top-8	0.579	0.545	0.851
Top-10	0.415	0.768	0.660

systems with less than 200 classes up to systems with more than 2600 classes, receptively. We also noticed that accuracy, precision and recall are not significantly affected by the systems' size as it tends to be stable with a standard deviation score of 1.15, 073 and 1.23 respectively.

In fact, we expected that the CPU time will slightly increase when the system size increase as the solution size depends on the number of classes in the system (c.f., Section 4.2.2. To further investigate this behavior we use the JVM Monitor[16] profiler for Java for tracking which operations are the most expensive. The results obtained from this profiling show that over 46% of the execution time is spent on finding new candidate libraries by evaluating our two constraints: (*i*) recommended libraries should be different than the ones used by the system, and (*ii*) recommended libraries should not be similar as described in Section 4.2.2. These operations are executed when constructing initial solutions or during crossover and mutation operators. Consequently, execution time is slightly increased with larger solutions that require in turn more constraints to be checked. Overall, we can say that *LibFinder* is scalable with respect to system size since it provides high precision and recall values, and reasonable execution time.

Another important point to highlight is about to stability of *LibFinder* over different folds. Stable results over different folds reflect the suitability of both the data and the exploration technique based on NSGA-II. To this end, we studied the standard deviation of the obtained accuracy, precision and recall for the different top-k results. The result for this experiment is shown in Table 9. We notice that the highest standard deviation of the accuracy score

---

16 http://jvmmonitor.org/, version 3.8.1

was 1.53 with the accuracy@1, while the lowest score was 0.41 with the accuracy@10. Similarly precision and recall scores does not vary significantly as their standard deviation was lower than 1.32 and 1.98 respectively. This indicates that *LibFinder* is relatively stable over different systems as they was randomly split over the different folds.

6. Threats to validity

Several factors can bias the validity of empirical studies. In this subsection, we elaborate on several factors that may threat the validity of our results.

Internal threats can be related to the stochastic nature of search algorithms employed[42]. To mitigate this, we conducted non-parametric statistical testing. We used the Wilcoxon Signed Rank test [43] over 30 independent runs with a 95% ($\alpha < 0.05$) confidence level to test if significant differences exist between the measurements for different treatments along with Cohen effect size for measuring the difference magnitude. This test makes no assumption on the data distribution and is suitable for ordinal data. We are, thus, confident that the observed statistical relationships are significant. Other threats to internal validity refers to experimenter bias. Most of our experimental process is automated and randomized. Thus we believe there is little experimenter bias.

Construct threats to validity may arise from our evaluation metrics accuracy@k, precision@k and recall@k to measure the effectiveness of our approach. Although these metrics are well known measure that are widely used in evaluating recommendation systems is software engineering [14,25,49], 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 [25]. Hence, in the library recommendation problem, a system might not adopt a specific library for many reasons mainly if the system's developers are not aware by such a library especially with the exponentially growing number of available libraries in code repositories, or due to other constraints such as budget limitations or deadline pressure, etc. We are thus planning to further evaluate *LibFinder* with developers in an industrial setting. Another potential threat to validity can be related to extracted identifiers. In fact, identifiers in a system may be influenced by class names and method names from its dependent library. Usually library import statements involve identifiers (package/class names) from the original library, and similarly for method calls which involve method names from the library. The presence of library identifiers in the projects code might bias our semantic similarity measure to retrieve the removed library. In other words, the library dependency is removed, but some of the library identifiers are still in the projects code. To mitigate this potential threat to validity, we excluded all identifiers related to library import and invoke from the projects code.

Moreover, another possible threat could be related to the dependencies extraction from pom files. Having a dependency declared in the pom.xml does not necessary grantee that the project is actually using it (i.e., invoking it). To address this issue, we compiled and inspected a random set of over 300 projects using `jcabi-aether`[17] library and `JavaCompiler` (ver.1.6) eclipse compiler[18] to automatically log all loaded classes. In our effort, less than 4% of library dependencies are not used by their projects. The percentage is insignificant and we believe it does not affect the results of dependency analysis.

External threats to validity refers to the generalizability of our findings. We considered a large set of open-source Java projects collection from Github that use libraries from Maven. Although Maven is a large and popular tool for the development community, in practice only a subset of developers are based in Maven to help manage their build process. To mitigate this threat, we have tested *LibFinder* with 32,760 projects with different sizes and from different application domains. We are also planning to conduct an industrial evaluation of our recommendation system to assess its effect on code quality as well as developers productivity.

7. Related work

In recent years, there has been much interest in academia as well as in industry on providing tools to help developers on understanding and using library APIs. Most of the related work relies on library method recommendation and API usage patterns.

Library method recommendation. Heinemann et al. [15] proposed an approach for recommending library methods using data mining for learning term-method associations based on identifiers similarity, similarly to our approach. *Rascal* [50] uses collaborative filtering to suggest API methods based on a set of already employed methods within a class. Similarly, *Javawock,tsunoda2005javawock* uses same technique to recommend API classes instead of methods. Thung et al. [14,16] have proposed a technique for recommending API methods based on textual description of feature requests and method history usage. While most of existing approaches suppose that developer has already find his library and he needs support on how to use it, our approach recommends whole library. *LibRec* [9] is the approach most related to ours. *LibRec* uses association rule mining and collaborative filtering on historic software artefacts to determine commonly used together libraries. The main limitation of *LibRec* is that a library is regarded as a black box where recommendations are based on how other client systems previously use it. Consequently, the proposed approach was not able to recommend libraries to projects that only use a small number of libraries or do not use any libraries at all. Unlike *LibRec*, our approach opts a content-based recommendation combined with library usage history. As shown in our empirical evaluation, effective library recommendation should be driven, most importantly, by the content of the library. Furthermore, our approach is based on a larger dataset of 32,760 client systems, while *LibRec* uses only 500.

API usage patterns. Several approaches have been proposed to mine API usage patterns to help developers on using their library APIs. Most of them propose temporal [17], unordered [51,52] and sequential [53,54] usage patterns, based on clients usage. For instance, MAPO [53] mines API usage patterns from existing source code. The patterns are given by methods that are frequently called together and that follow certain sequential rules. We believe that these approaches would be complementary to ours, as they provide support on how to use the recommended libraries.

Use of code identifiers. Source code vocabulary has been widely used in several purposes in software engineering [15,16,55–60]. Bajracharya et al. [61] used structural semantic indexing (SSI) to associate words to source code entities based on API usage similarities. Their goal is to improve the retrieval of API usage examples from code repositories. Mudablue [62] is a tool that categorizes software systems based on their code identifiers. Ouni et al. [31,63] proposed a vocabulary-based approach to recommend refactoring in order to preserving the semantic coherence of the code based on the semantic information embodied in code identifiers.

Combining *SBSE* and *MSR*. Software practitioners and researchers are recognizing the benefits of SBSE and MSR techniques to support the maintenance and evolution of software systems, improve software design/reuse, and empirically validate novel ideas

---

[17] Aether adapter for maven plugins, http://aether.jcabi.com/index.html, accessed 08-09-2016.

[18] Javacompiler tool, http://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.CompilationTask.html, accessed 08-09-2016.

and techniques [22–24]. In fact, there is recently an increase in the interactions between these two fields. Harman et al. [64] used Genetic Improvement for Adaptive Software Engineering where genetic programming is used as a means of program improvement based on a dataset of code fragments collected from software repositories. The research area has come to be known as 'genetic improvement'. You et al. [65] have proposed an SBSE approach to optimize non-functional properties of a system such as JIT compilation, and hardware dependent algorithm using libraries. In the area of bug prediction, Canfora et al. [66] used a multi-objective optimization approach, named MODEP, to train from 10 datasets from the PRedictOrModels In Software Engineering Software (PROMISE) repository. The proposed approach allows software engineers to choose predictors achieving a specific compromise between the number of likely defect-prone classes or the number of defects that the analysis would likely discover, and lines of code to be analysed/tested. Minku et al. [67] formulated the problem of software effort estimation as a multiobjective learning problem to understand the trade-off among different performance measures. The conducted study was based on five data sets from the PROMISE repository.

Another software engineering problem where SBSE has been used to mine software repositories is software product lines (SPL) recommendation and configuration focusing on feature model selection [68,69]. A Software Product Line represents a set of software products that share features in order to satisfy a specific market segment where a feature represents a functionality that is visible for the user. Sayyad et al. [70] studied the use of search-based algorithms for SPL feature selection as a multi-objective problem. They make explicit the link between search based software engineering for requirements selection and search based optimization of choices pertaining to feature models. Guo et al. [71,72] introduced a genetic algorithm to find SPL feature sets while considering the cost and value objective (value-per-unit-cost). Muller [73] also formulated the choice of products to be built from an SPL as a cost-value trade off, using the simulated annealing algorithm to find suggested choices of features that would form products that balance these trade offs. They focus on differing customer segments (stakeholder groups), observing that not all such groups can necessary be satisfied by the products offered (due to budgetary constraints). Cruz et al. [74] use a hybrid approach, which combines fuzzy inference systems and the well-known multi-objective genetic algorithm, NSGA-II,to help decision makers manage product lines by generating portfolios of products. These portfolios are based on user segments and the development cost of SPL products.

Indeed, we expect more adoption and unification of both SBSE and MSR techniques to solve several other software engineering problems in the future.

8. Conclusion and future work

In this paper, we have introduced *LibFinder*, a novel approach for third-party library recommendation. *LibFinder* unifies SBSE and MSR techniques, by exploring a large dataset collected from library usage history and identifiers mined from code in large repositories. The goal is to prevent missed reuse opportunities during software maintenance and evolution, by attracting the attention of developers to potentially useful third-party libraries to their software systems. We empirically evaluated our approach, we mined the usage history of 6083 libraries and 32,760 client systems from Maven and Github repositories, respectively. The obtained results show that our approach is efficient in recommending useful libraries comparing to random search and two other popular search algorithms with more than 92% of accuracy, 51% of precision and 68% of recall with top-10 recommendations. Furthermore, we have shown that *LibFinder* is significantly better than a state-of-the-art technique which is based on the usage history libraries. Furthermore, we evaluated the usefulness of our approach in practice through an empirical study on two industrial Java systems with developers. Results show that the top 10 recommended libraries was rated by the original developers with an average of 3.25 out of 5.

As part of our future work, we plan to conduct an industrial evaluation of *LibFinder* with developers to better understand the impact of adopting external libraries on the quality of their systems as well as their productivity. We also plan to consider more software artifacts from other popular code repositories to better validate and generalize our results. More importantly, we will extend *LibFinder* by formulating the library recommendation problem as an interactive optimization problem to integrate the developer in the loop when recommending libraries. Another interesting extension of *LibFinder* can be to consider the change history of a system, so that library recommendation can be addressed to classes that are actively changed and maintained by developers. Moreover, we are planning to integrate *LibFinder* as an Eclipse plugin and try to provide 'on-the-fly' recommendations in such a way that the developer will be automatically notified by relevant libraries while he is writing his code. Yet another direction is to consider the library version and the internal quality of the recommended library to ensure high quality software systems.

References

[1] C.W. Krueger, Software reuse, ACM Comput. Surv. (CSUR) 24 (2) (1992) 131–183.
[2] W.C. Lim, Effects of reuse on quality, productivity, and economics, IEEE Softw. 11 (5) (1994) 23–30.
[3] E.-A. Karlsson (Ed.), Software Reuse: A Holistic Approach, John Wiley & Sons, Inc., New York, NY, USA, 1995.
[4] W.C. Lim, Effects of reuse on quality, productivity, and economics, IEEE Softw. 11 (5) (1994) 23–30.
[5] R.C. Seacord, D. Plakosh, G.A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[6] M. Schäfer, M. Sridharan, J. Dolby, F. Tip, Refactoring java programs for flexible locking, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE), 2011, pp. 71–80.
[7] K. Ishizaki, S. Daijavad, T. Nakatani, Refactoring java programs using concurrent libraries, in: Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), 2011, pp. 35–44.
[8] B.W. Boehm, Software Engineering Economics, 1st, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
[9] F. Thung, D. Lo, J. Lawall, Automated library recommendation, in: 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 182–191.
[10] D. Kawrykow, M.P. Robillard, Improving api usage through automatic detection of redundant code, in: IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society, 2009, pp. 111–122.
[11] E. Juergens, F. Deissenboeck, B. Hummel, Code similarities beyond copy & paste, in: 14th European Conference on Software Maintenance and Reengineering (CSMR), 2010, pp. 78–87.
[12] R.G. Kula, D.M. German, T. Ishio, K. Inoue, Trusting a library: a study of the latency to adopt the latest maven release, in: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 520–524.
[13] S. Raemaekers, A.V. Deursen, J. Visser, Semantic versioning versus breaking changes : A study of the maven repositor, 2014.
[14] F. Thung, S. Wang, D. Lo, J. Lawall, Automatic recommendation of api methods from feature requests, in: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013, pp. 290–300.
[15] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, B. Hummel, Identifier-based context-dependent api method recommendation, in: 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, pp. 31–40.

[16] W.-K. Chan, H. Cheng, D. Lo, Searching connected api subgraph via text phrases, in: ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE), 2012, p. 10.
[17] G. Uddin, B. Dagenais, M.P. Robillard, Temporal analysis of api usage concepts, in: 34th International Conference on Software Engineering (ICSE), 2012, pp. 804–814.
[18] M. Tsunoda, T. Kakimoto, N. Ohsugi, A. Monden, K.-i. Matsumoto, Javawock: a java class recommender system based on collaborative filtering, in: 17th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2005, pp. 491–497.
[19] M. Harman, B.F. Jones, Search-based software engineering, Inf. Softw. Technol. 43 (14) (2001) 833–839.
[20] F. Ferrucci, M. Harman, J. Ren, F. Sarro, Not going to take this anymore: multi-objective overtime planning for software engineering projects, in: International Conference on Software Engineering (ICSE), IEEE Press, 2013, pp. 462–471.
[21] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, IEEE Trans. Evolut. Comput. 6 (2) (2002) 182–197.
[22] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: trends, techniques and applications, ACM Comput. Surv. (CSUR) 45 (1) (2012) 11.
[23] A.E. Hassan, The road ahead for mining software repositories, in: Frontiers of Software Maintenance (FoSM), IEEE, 2008, pp. 48–57.
[24] H. Kagdi, M.L. Collard, J.I. Maletic, A survey and taxonomy of approaches for mining software repositories in the context of software evolution, J. Softw. Maintenance Evolut.: Res. Pract. 19 (2) (2007) 77–131.
[25] I. Avazpour, T. Pitakrat, L. Grunske, J. Grundy, Dimensions and metrics for evaluating recommendation systems, in: Recommendation Systems in Software Engineering, Springer, 2014, pp. 245–273.
[26] A. Ouni, A Mono-and Multi-objective Approach for Recommending Software Refactoring, Ph.D. thesis, University of Montreal, 2014.
[27] T. Patanamon, T. Chakkrit, G.K. Raula, Y. Norihiro, I. Hajimu, M. Ken-ichi, Who should review my code? a file location-based code-reviewer recommendation approach for modern code review, in: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015.
[28] G. Adomavicius, A. Tuzhilin, Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions, IEEE Trans. Knowl. Data Eng. 17 (6) (2005) 734–749.
[29] F. Ricci, L. Rokach, B. Shapira, Introduction to Recommender Systems Handbook, Springer, 2011.
[30] K. Deb, Multi-objective optimization using evolutionary algorithms, 16, John Wiley & Sons, 2001.
[31] A. Ouni, M. Kessentini, H. Sahraoui, M.S. Hamdi, Search-based refactoring: Towards semantics preservation, in: 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 347–356.
[32] C.D. Manning, P. Raghavan, H. Schütze, et al., Introduction to information retrieval, 1, Cambridge university press Cambridge, 2008.
[33] T. Ishio, R.G. Kula, T. Kanda, D.M. German, K. Inoue, Software ingredients: detection of third-party component reuse in java software release, in: IEEE Working Conference on Mining Software Repositories (MSR), 2016, p. toappear.
[34] D.C. Karnopp, Random search techniques for optimization problems, Automatica 1 (2) (1963) 111–121.
[35] H. Li, Q. Zhang, Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii, IEEE Trans. Evolut. Comput. 13 (2) (2009) 284–302.
[36] E. Zitzler, S. Künzli, Indicator-based selection in multiobjective search, in: Parallel Problem Solving from Nature-PPSN VIII, Springer, 2004, pp. 832–842.
[37] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. Da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Trans. Evolut. Comput. 7 (2) (2003) 117–132.
[38] D. Van Veldhuizen, G.B. Lamont, et al., On measuring multiobjective evolutionary algorithm performance, in: Congress on Evolutionary Computation (CEC), 1, 2000, pp. 204–211.
[39] C.M. Fonseca, J.D. Knowles, L. Thiele, E. Zitzler, A tutorial on the performance assessment of stochastic multiobjective optimizers, in: International Conference on Evolutionary Multi-Criterion Optimization (EMO), 216, 2005, p. 240.
[40] D. Jannach, M. Zanker, A. Felfernig, G. Friedrich, Recommender Systems: An Introduction, Cambridge University Press, 2010.
[41] P.M. Chisnall, Questionnaire design, interviewing and attitude measurement, J. Mark. Res. Soc. 35 (4) (1993) 392–393.
[42] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: 33rd International Conference on Software Engineering (ICSE), IEEE, 2011, pp. 1–10.
[43] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, Academic press, 1988.
[44] A. Arcuri, G. Fraser, On parameter tuning in search based software engineering, in: Search Based Software Engineering, Springer, 2011, pp. 33–47.
[45] A.E. Eiben, S.K. Smit, Parameter tuning for configuring and analyzing evolutionary algorithms, Swarm and Evolut. Comput. 1 (1) (2011) 19–31.
[46] J.L. Fleiss, Measuring nominal scale agreement among many raters., Psychol. Bull. 76 (5) (1971) 378.

[47] J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data, Biometrics (1977) 159–174.
[48] I. Sommerville, Software Engineering, 6, Pearson Education Ltd, 2001.
[49] X. Wang, L. Zhang, T. Xie, J. Anvik, J. Sun, An approach to detecting duplicate bug reports using natural language and execution information, in: 30th International Conference on Software Engineering (ICSE), ACM, 2008, pp. 461–470.
[50] F. Mccarey, M.Ó. Cinnéide, N. Kushmerick, Rascal: a recommender agent for agile reuse, Artif. Intell. Rev. 24 (3-4) (2005) 253–276.
[51] S. Mohamed Aymen, B. Omar, A. Hani, S. Houari, Mining multi-level api usage patterns, in: International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 23–32.
[52] Z. Li, Y. Zhou, Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code, in: ACM SIGSOFT Software Engineering Notes, 30, ACM, 2005, pp. 306–315.
[53] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, Mapo: Mining and recommending api usage patterns, in: S. Drossopoulou (Ed.), European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, 5653, Springer Berlin Heidelberg, 2009, pp. 318–343.
[54] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, D. Zhang, Mining succinct and high-coverage api usage patterns from source code, in: IEEE Working Conference on Mining Software Repositories (MSR), 2013, pp. 319–328.
[55] C. McMillan, M. Grechanik, D. Poshyvanyk, Detecting similar software applications, in: 34th International Conference on Software Engineering (ICSE), 2012, pp. 364–374.
[56] A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: 34th International Conference on Software Engineering (ICSE), 2012, pp. 837–847.
[57] L. Inozemtseva, S. Subramanian, R. Holmes, Integrating software project resources using source code identifiers, in: 36th International Conference on Software Engineering (ICSE), 2014, pp. 400–403.
[58] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? A study of identifiers, in: Proceedings of the 14th International Conference on Program Comprehension (ICPC), 2006, pp. 3–12.
[59] A. Ouni, R.K. Gaikovina, K. Inoue, Search-based peer reviewers recommendation in modern code review, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016.
[60] A. Ouni, Z. Salem, K. Inoue, M. Soui, SIM: an automated approach to improve web service interface modularization, in: IEEE International Conference on Web Services (ICWS), 2016, pp. 91–98.
[61] S.K. Bajracharya, J. Ossher, C.V. Lopes, Leveraging usage similarity for effective retrieval of examples in code repositories, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2010, pp. 157–166.
[62] S. Kawaguchi, P.K. Garg, M. Matsushita, K. Inoue, Mudablue: an automatic categorization system for open source repositories, J. Syst. Softw. 79 (7) (2006) 939–953.
[63] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: an industrial case study, ACM Trans. Softw. Eng. Methodol. 25 (3) (2016) 23:1–23:53.
[64] M. Harman, Y. Jia, W.B. Langdon, J. Petke, I.H. Moghadam, S. Yoo, F. Wu, Genetic improvement for adaptive software engineering (keynote), in: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, in: SEAMS 2014, 2014, pp. 1–4.
[65] S. Yoo, Amortised Optimisation of Non-functional Properties in Production Environments, pp. 31–46.
[66] G. Canfora, A.D. Lucia, M.D. Penta, R. Oliveto, A. Panichella, S. Panichella, Defect prediction as a multiobjective optimization problem, Softw. Test., Verif. Reliab. 25 (4) (2015) 426–459.
[67] L.L. Minku, X. Yao, Software effort estimation as a multiobjective learning problem, ACM Trans. Softw. Eng. Methodol. 22 (4) (2013) 35:1–35:32.
[68] M. Harman, Y. Jia, J. Krinke, W.B. Langdon, J. Petke, Y. Zhang, Search based software engineering for software product line engineering: a survey and directions for future work, in: 18th International Software Product Line Conference-Volume 1, 2014, pp. 5–18.
[69] R.E. Lopez-Herrejon, L. Linsbauer, A. Egyed, A systematic mapping study of search-based software engineering for software product lines, Inf. Softw. Technol. 61 (2015) 33–51.
[70] A.S. Sayyad, T. Menzies, H. Ammar, On the value of user preferences in search-based software engineering: a case study in software product lines, in: International Conference on Software Engineering (ICSE), 2013, pp. 492–501.
[71] J. Guo, J. White, G. Wang, J. Li, Y. Wang, A genetic algorithm for optimized feature selection with resource constraints in software product lines, J. Syst. Softw. 84 (12) (2011) 2208–2221.
[72] J. Li, X. Liu, Y. Wang, J. Guo, Formalizing feature selection problem in software product lines using 0-1 programming, in: Practical Applications of Intelligent Systems, Springer, 2011, pp. 459–465.
[73] J. Muller, Value-based portfolio optimization for software product lines, in: 15th International Software Product Line Conference (SPLC), 2011, pp. 15–24.
[74] J. Cruz, P.S. Neto, R. Britto, R. Rabelo, W. Ayala, T. Soares, M. Mota, Toward a hybrid approach to generate software product line portfolios, in: IEEE Congress on Evolutionary Computation, 2013, pp. 2229–2236.