10-2013

Automated Library Recommendation

Ferdian THUNG
Singapore Management University, ferdiant.2013@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Julia LAWALL

DOI: https://doi.org/10.1109/WCRE.2013.6671293

Citation

THUNG, Ferdian; LO, David; and LAWALL, Julia. Automated Library Recommendation. (2013). *2013 20th Working Conference on Reverse Engineering (WCRE 2013): Proceedings: Koblenz, Germany, 14-17 October 2013*. 182-191. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/2026

Automated Library Recommendation

Ferdian Thung[1], David Lo[1], and Julia Lawall[2]
[1]Singapore Management University, Singapore
[2]Inria/LIP6-Regal, France
{ferdiant.2013,davidlo}@smu.edu.sg, julia.lawall@lip6.fr

***Abstract*—Many third party libraries are available to be downloaded and used. Using such libraries can reduce development time and make the developed software more reliable. However, developers are often unaware of suitable libraries to be used for their projects and thus they miss out on these benefits. To help developers better take advantage of the available libraries, we propose a new technique that automatically recommends libraries to developers. Our technique takes as input the set of libraries that an application currently uses, and recommends other libraries that are likely to be relevant.**

We follow a hybrid approach that combines association rule mining and collaborative filtering. The association rule mining component recommends libraries based on a set of library usage patterns. The collaborative filtering component recommends libraries based on those that are used by other similar projects. We investigate the effectiveness of our hybrid approach on 500 software projects that use many third-party libraries. Our experiments show that our approach can recommend libraries with recall rate@5 of 0.852 and recall rate@10 of 0.894.

I. Introduction

Third-party libraries are an integral part of many software projects [3], [11], [26]. For example, we have investigated 1008 projects of substantial size from GitHub. We found that 93.3% of them use third-party libraries, at an average of 28 third-party libraries each. The use of such libraries allows the developer to write less code and to focus on the parts of the code that are specific to the project. The use of third-part libraries also reduces the need for testing.

Today, many third-party libraries are readily available to software developers, from repositories such as the Maven repository.[1] Still, effectively using these libraries remains a challenge for developers, because they may not become aware of these libraries as they are released. Developers may thus be led to "re-implement the wheel". An approach is needed to bridge the gap between the many third-party libraries that are available and the developers that need to use them.

In this work, we propose a technique that automatically recommends libraries for a particular project. Given the set of libraries that the project has used, our technique recommends other libraries that are potentially useful for it. Our technique follows a hybrid approach that combines association rule mining and collaborative filtering. The association rule mining component extracts libraries that are commonly used together. The component then rates each of the libraries based on their likelihood to appear together with the currently used libraries. The collaborative filtering component works on the assumption that similar projects are likely to share similar third party libraries. The component analyzes the libraries that are used by the n most similar projects. It then rates each of the libraries based on how many of the top-n most similar projects use it. Our technique finally aggregates the recommendations made by the association rule mining and collaborative filtering components.

A number of previous studies have proposed approaches to recommend library methods to be used in a particular context, e.g., [18], [37]. Our work differs from this previous work in terms of the level of granularity considered. While previous approaches recommend a particular method to be used in a particular context, we target the problem of recommending an entire library (e.g., an entire jar file in the case of a Java project). Past approaches assume that the set of relevant libraries is already known to the developer and it is only the methods in these libraries that are unknown. Our work does not make this assumption, and thus complements these existing studies. Indeed, our approach could be deployed first to recommend particular libraries that will interest developers. These results could then be fed to existing approaches to recommend particular methods to be used in different contexts.

To evaluate the effectiveness of our approach, we have downloaded a few hundred Java projects of substantial size ($\geq$ 10,000 lines) from GitHub[2] and investigated the libraries that these projects use. We observe that these projects make use of a substantial number of third-party libraries and thus are appropriate subjects of our study. Evaluating our approach on projects that use many libraries ensures that it is able to recommend libraries to *real* projects that *need* third-party libraries. We then use ten-fold cross validation to evaluate our approach. For this, we divide the dataset into ten parts. Nine parts are used as training data, and one part is used as test data. The results over the ten iterations are aggregated. To evaluate our results, we use recall rate@5 and recall rate@10, which are often used as evaluation measures [22], [30], [36]. In our experiments we achieve recall rate@5 and recall rate@10 of 0.852 and 0.894, respectively.

The contributions of our work are as follows:

1) We identify a new problem of library recommendation: Given a set of libraries that a project uses, recommend other libraries that are potentially useful for it.
2) We propose a hybrid technique based on association rule mining and collaborative filtering to recommend libraries

[1] repo1.maven.org, http://search.maven.org

[2] https://github.com/

978-1-4799-2931-3/13 © 2013 IEEE

that a project can use based on the libraries that the project already uses.

3) To test the effectiveness of our approach, we investigate the third-party libraries used by a few hundred projects. Our experiment shows that our approach is effective and can achieve recall rate@5 and recall rate@10 of 0.852 and 0.894, respectively.

The structure of this paper is as follows. We formally define the problem and provide an illustrative example in Section II. We describe the base algorithms in Section III. We then present our proposed approach in Section IV. Next, we describe our experiments in Section V. Related work is presented in Section VI. We finally conclude and mention future work in Section VII.

II. Problem Definition & Illustrative Example

In this section, we define our problem and illustrate it by a motivating example.

Problem Definition. We define the library recommendation problem as follows. Let *allLibraries* be the set of all libraries that are available, *usedLibraries* be the set of libraries that are currently used in the software project, *usefulLibraries* be the set of libraries that are useful for the project, and *recLibraries* be the set of libraries that are to be recommended. The goal of our approach is to find the a set *recLibraries* that satisfies the following conditions:

1) $\mathit{recLibraries} \subseteq \mathit{allLibraries}$
2) $\mathit{recLibraries} \bigcap \mathit{usedLibraries} = \emptyset$
3) $\mathit{recLibraries} \bigcap \mathit{usefulLibraries} \neq \emptyset$

The recommended set of libraries needs to be a subset of all available libraries and should not contain any library that is currently used in the project (there is no need to recommend a library that is already used). Finally, we want to recommend libraries that are useful.

Illustrative Example. To illustrate the issues involved, we provide a motivating example. Consider a project named *openpipe* that uses the following libraries: *logback-classic*, *spring-context*, *lucene-core*, *commons-collections*, *commons-dbcp*. Now, consider the database shown in Table I which maps a set of projects to the libraries that they use. We want to recommend libraries to *openpipe* based on this database.

One approach to recommend libraries is to look for library usage patterns. From the database, one can see that most projects that use *commons-collections* also use *commons-lang* libraries - for projects *easysoa*, *red5-mavenized*, and *thucydides* this is the case. We can thus infer a *library usage pattern*: *commons-collections* → *commons-lang*. Based on this usage pattern, since *openpipe* uses *commons-collections* but does not use *commons-lang*, we would recommend *commons-lang* to *openpipe*.

Another approach to recommend libraries is to look for projects that are similar to *openpipe* in Table I and investigate the set of libraries that they use. We measure the similarity of two projects based on the set of libraries that are used in common between the two projects.[3] Comparing the complete set of libraries already used by *openpipe* with those of the projects in database, we find that there are two projects that use a similar set of librarie: *easysoa* and *fuse*. These projects share the following libraries with *openpipe*: *lucene-core*, *logback-classic*, and *spring-context*. Furthermore, *easysoa* and *fuse* also use a library that is not used in *openpipe*, namely *commons-httpclient*. Thus, we would recommend *commons-httpclient* to *openpipe*.

We propose to automate the above approaches for recommending libraries to a project. We automate the first approach by using association rule mining to extract library usage patterns. We automate the second approach by leveraging a collaborative filtering technique. We then combine these two approaches to recommend libraries.

III. Preliminaries

In this section, we describe several techniques that are used in our approach: *frequent itemset mining* [1], *association rule mining* [1], and *collaborative filtering* [32]. Frequent itemset mining is the first step of association rule mining.

A. Frequent Itemset Mining

Frequent itemset mining takes as input a transaction database (i.e., a multi-set of transactions), where each transaction is a set of items, and outputs sets of items (aka. itemsets) that appear frequently (i.e., each frequent itemset is a subset of many transactions) in the database. In our setting, a transaction is the set of third party libraries that are used by a project. We refer to the number of transactions that contain all of the elements of an itemset I as the *frequency* of I, denoted as $\mathit{freq}(I)$. The *support* of an itemset I is defined as:

$$\mathit{sup}(I) = \frac{\mathit{freq}(I)}{N}$$

where N is the number of transactions in the transaction database. An itemset is *frequent* if its support is no less than *minsup*, where *minsup* is a user-defined minimum support threshold.

Example 1. Consider a project database shown in Table I. Each project can be considered as a transaction. If *minsup* is e.g. 0.5, then $I = \{\mathit{commons\text{-}collections}, \mathit{commons\text{-}logging}\}$ is a frequent itemset in these transactions, computed as follows. I appears in 3 transactions (easysoa, red5-mavenized, thucydides), so $\mathit{freq}(I)$ is 3. Since the number of transactions in the database (N) is 5, $\mathit{sup}(I)$ is 0.6, which is greater than *minsup*.

B. Association Rule Mining

In addition to frequent itemsets, another kind of pattern can be extracted from the transaction database. For example, from the database shown in Table I, we can infer: "if a project uses $\mathit{commons\text{-}collections}$, then the project is likely to also use $\mathit{commons\text{-}lang}$" because all of the transactions

[3] McMillan et al. use API calls as semantic anchors to measure the similarity between projects [21]; we use a similar idea.

TABLE I
EXAMPLE PROJECT DATABASE

Project	**Libraries**
easysoa	lucene-core, commons-httpclient, logback-classic, spring-context, commons-collections, commons-lang
fuse	commons-httpclient, camel-core, lucene-core, logback-classic, aether-util, spring-context
red5-mavenized	groboutils-core, commons-collections, ehcache, jta, commons-lang, catalina
histone-java	mockito-all, reflections, spring-context, cglib, joda-time, xstream
thucydides	logback-classic, commons-collections, spring-context, commons-lang, opencsv, groovy

that contain *commons-collections* also contain *commons-lang*. This kind of pattern is referred to as an *association rule*. An association rule is an "if/then" rule that captures a relationship between two itemsets X and Y in the database. It can be written as:

$$X \rightarrow Y$$

where X is the pre-condition of the rule and Y is the post-condition of the rule. The pre-condition is a statement that must be satisfied for the rule to be applied, whereas the post-condition is the result if the pre-condition is met.

We are interested in association rules that apply to many transactions in the database. For this, we use a metric referred to as *support*. The support of an association rule $R = X \rightarrow Y$, denoted as $sup(R)$, is the proportion of transactions in the database that contain $X \bigcup Y$. We also need to measure the likelihood that a rule is true. For this purpose, we use a metric referred to as *confidence*. The confidence of a rule R with pre-condition X and post-condition Y (i.e., $R = X \rightarrow Y$) is defined as follows:

$$conf(R) = \frac{freq_{X \bigcup Y}}{freq_X}$$

Association rule mining extracts all rules that satisfy user-defined minimum support (*minsup*) and confidence (*minconf*) thresholds. We refer to such rules as *significant association rules*. Association rules can be generated from frequent itemsets. We can enumerate all possible pairs of frequent itemsets where one is a subset of another. Consider A and B where A and B are frequent itemsets and A is a subset of B. Then, the generated association rule R is of the form:

$$A \rightarrow B \setminus A$$

The support and confidence of the rule R can be computed from the supports of its constituent itemsets as follows:

$$sup(R) = sup(B)$$
$$conf(R) = \frac{sup(B)}{sup(A)}$$

Example 2. We refer to Table I. Given $minsup$ = 0.5, frequent itemsets $A = \{commons\text{-}collections\}$ and $B = \{commons\text{-}collections, commons\text{-}lang\}$. Frequent itemsets A and B form the rule $R = commons\text{-}collections \rightarrow commons\text{-}lang$. The support of R is the same as $sup(B)$, which is 0.6. Since $sup(A)$ is 0.6, the confidence of the rule is 1.0.

C. Collaborative Filtering

Collaborative filtering is an automatic technique to make predictions about an entity based on information collected about other similar entities. Collaborative filtering has been used in many real systems, including environmental sensing, financial services, electronic commerce, web applications, etc. [29]. One popular implementation of collaborative filtering in the context of web applications is the recommendation system developed by Amazon.com for recommending new items to the users in their website [14].

A basic method to perform collaborative filtering is by finding the nearest neighbors of the target entity. A target entity is compared with all other entities and a list of most similar entities based on a distance metric is produced. The similarities among the entities are used as a basis for making predictions about the entity. In our setting, an entity is a project, and the prediction task is the prediction of libraries that are useful for the project.

IV. PROPOSED APPROACH

In this section, we first provide a birds-eye-view description of our overall framework in Section IV-A. We then zoom in to the various components of our framework. We describe the association rule mining component in Section IV-B. We then present the collaborative filtering component in Section IV-C. The aggregator component which combines the result of the association rule mining and collaborative filtering components is described in Section IV-D.

A. Overall Framework

Figure 1 shows the overall framework of our recommendation system referred to as *LibRec*. It has two major components: $LibRec^{RULE}$ and $LibRec^{COLLAB}$. $LibRec^{RULE}$ recommends libraries by mining library usage patterns expressed as association rules. $LibRec^{COLLAB}$ recommends libraries by investigating the set of libraries that are used by similar projects, using a nearest-neighbor-based collaborative filtering approach. Our framework consists of two phases: *training* and *deployment*.

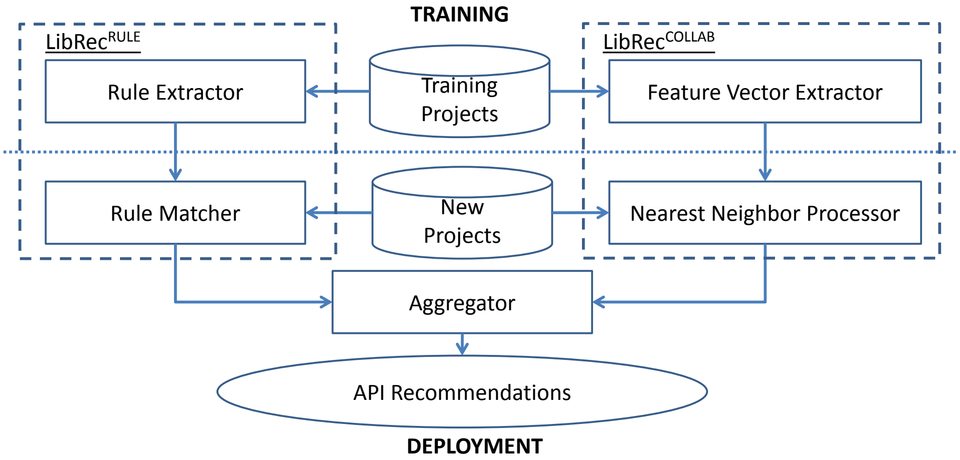

Fig. 1. Our Recommendation Framework *LibRec*

In the training phase, our system infers models needed for the deployment phase. These models are extracted from a training dataset ($TrainingProjects$). $TrainingProjects$ is a set of projects, along with the names of third-party libraries used by each of them. Models are extracted by the following sub-components:

1) $RuleExtractor$, a sub-component of $LibRec^{RULE}$, extracts association rules that capture library usage patterns from *Training Projects*.
2) $FeatureVectorExtractor$, a sub-component of $LibRec^{COLLAB}$, extracts a vector of feature values from the set of libraries that each project uses. Each feature corresponds to a library and its value is 1 if the library is used by the project and 0 otherwise.

The association rules and vectors are models that are provided to the deployment phase.

In the deployment phase, our system recommends libraries to new projects ($NewProjects$) based on the models extracted from the training phase. $NewProjects$ is a set of new projects along with the names of third-party libraries already used by each of them. The following sub-components use the models to recommend libraries:

1) $RuleMatcher$, a sub-component of $LibRec^{RULE}$, takes each new project $p \in NewProjects$ and the association rules extracted in the training phase as inputs. It matches the libraries used in the project with the rules. It makes recommendations based on the post-conditions of the matching rules.
2) $NearestNeighborProcessor$, a sub-component of $LibRec^{COLLAB}$, takes each new project $p \in NewProjects$ and the feature vectors extracted in the training phase as inputs. It then constructs a feature vector for p and calculates the distance between this feature vector and each of the feature vectors extracted from the training phase. The most similar vectors and their corresponding projects (i.e., the nearest neighbors) are identified. Recommendations are made based on the libraries used by the nearest neighbors.

Both $RuleMatcher$ and $NearestNeighborProcessor$ output a list of recommended libraries along with their recommendation scores. The $Aggregator$ component combines these two lists into a new list with the final recommendation scores. The libraries with the highest scores will be recommended.

We now describe each of the components in more detail.

B. $LibRec^{RULE}$ Component

Our $LibRec^{RULE}$ component recommends libraries based on library usage patterns. It consists of the $RuleExtractor$ sub-component in the training phase and the $RuleMatcher$ sub-component in the deployment phase.

1) $RuleExtractor$: This sub-component employs association rule mining to mine library usage rules. In $TrainingProjects$, each project uses a set of third-party libraries. We treat each project as a transaction where the items in the transactions are the third-party libraries that it uses.

Traditional association rule mining (see Section III) extracts *all* rules that satisfy the minimum support and confidence thresholds from the set of *all* frequent itemsets. However, often, too many rules are extracted. Association rule mining can thus become very slow. To address this issue, we observe that not all of the rules are necessary; some of the rules can be combined to construct a compact set of association rules.

Example 3. Consider the following rules: $R1 = log4j \rightarrow commons\text{-}logging$, $R2 = log4j \rightarrow slf4j\text{-}api$, and $R3 = log4j \rightarrow commons\text{-}logging, slf4j\text{-}api$. All of these rules have support 0.6 and confidence 1.0. Rules $R1$ and $R2$, however, are covered by rule $R3$ and they have the same support and confidence, and thus they are not actually needed.

To construct a compact set of association rules, we use two special subsets of *all* frequent itemsets, referred to in the literature as *closed itemsets* and *generators* [24]. A closed itemset is a frequent itemset with no superset having the same support as itself. A generator is a frequent itemset with no subset having the same support as itself. Algorithms have been developed to mine these closed itemsets and generators directly [24], [13], without the need to mine all frequent itemsets.

Example 4. Consider two frequent itemsets: $I1 = \{\mathit{log4j}\}$ and $I2 = \{\mathit{log4j}, \mathit{commons\text{-}logging}, \mathit{slf4j\text{-}api}\}$. $I1$ has support of 3 and does not have any subset having the same support. Thus, $I1$ is a generator. $I2$ has support of 3 and does not have a superset having the same support. Thus, $I2$ is a closed itemset. Note that $I1$ is not a closed itemset and $I2$ is not a generator as $I1 \subset I2$ and $sup(I1) = sup(I2)$.

We construct the compact set of association rules from these two subsets following Bastide et al. [2]. We define a compact set of association rules in Definition 1.

*Definition 1 (***Compact Assoc. Rules (ARules^{Compact})):** Consider a set of generators GEN, a set of closed itemsets $CLOSED$, a minimum support $minsup$ and a minimum confidence $minconf$. The compact set of association rules is the following set:

$$\begin{aligned}\{r = pre \rightarrow post | (pre \in GEN) \wedge \\ (pre \bigcup post \in CLOSED) \wedge \\ sup(r) \geq minsup \wedge conf(r) \geq minconf\}\end{aligned}$$

The set of compact rules is the set of frequent rules whose pre-condition is a generator and where the difference of the pre- and post-conditions is a closed itemset.

Example 5. Referring to Example 4, we can construct a compact rule from $I1$ and $I2$. The pre-condition of the compact rule is $I1 = \{\mathit{log4j}\}$ and the post-condition of the compact rule is $I2 \backslash I1 = \{commons - logging, slf4j - api\}$. Thus, we can the construct compact rule $C = log4j \rightarrow commons - logging, slf4j - api$.

The set of association rules obtained from closed itemsets and generators is potentially much smaller than the complete set of rules. The algorithm presented in Figure 2 constructs this compact set of association rules. The algorithm first mines the set of closed itemsets and generators (lines 7-8). We use the algorithm Zart [31] to mine these closed patterns and generators.[4] Next, the algorithm iterates over all of the closed itemsets and generators (lines 9-10). If the generator (*Gen*) is the subset of the closed itemset (*Item*), a rule is constructed (lines 11-22). *Gen* is the pre-condition of the rule and $Item \setminus Gen$ is its post-condition (lines 14-15). We then compute the support and confidence of the rule (lines 16-17). The constructed rule is added to the compact set of association rules if it satisfies the $minconf$ threshold (lines 18-19). There is no need to check the $minsup$ threshold as the closed itemsets are frequent. At the end, all the generated rules are stored for use in the deployment phase.

```
 1: Input:
 2:    $TrainingProjects$ = set of projects with third party
                              libraries used by each of them
 3:    $minconf$ = minimum confidence threshold
 4: Output:
 5:    Compact set of association rules
 6: Method:
 7: Let $ClosedItems$ = set of closed itemsets mined from
                              $TrainingProjects$
 8: Let $Generators$ = set of generators mined from
                              $TrainingProjects$
 9: Let $CompactRules = \{\}$
10: for all $Item \in ClosedItems$ do
11:    for all $Gen \in Generators$ do
12:       if $Gen \subset Item$ then
13:          Let $Rule = \{\}$
14:          $Rule.PreCondition = Gen$
15:          $Rule.PostCondition = Item \setminus Gen$
16:          $Rule.Support = Item.Support$
17:          $Rule.Conf = Item.Support / Gen.Support$
18:          if $Rule.Conf \geq minconf$ then
19:             add $Rule$ to $CompactRules$
20:          end if
21:       end if
22:    end for
23: end for
24: return $CompactRules$
```

Fig. 2. Mining a Compact Set of Association Rules

2) RuleMatcher: In the deployment phase, for each new project p to receive library recommendations, *RuleMatcher* gets the list *currentLib* of the libraries that it currently uses, and then matches this list against the association rules *libRules* generated by the *RuleExtractor* component. A rule *matches currentLib* if its precondition is a subset of *currentLib*. *RuleMatcher* then recommends libraries, based on the post-conditions of the matching rules.

We assign a score to assess the suitability of a library to a new project p. Informally, the *rule-based recommendation score* for a library A is the highest confidence of any matching rule whose post-condition contains A. This score is computed by the following formula:

$$\begin{aligned}RecScore^{RULE}(A) = MAX(0, MAX_{R \in RMatched(A)}.conf(R)) \\ RMatched(A) = \{(R = X \rightarrow Y) \in libRules \mid \\ X \subseteq currentLib \wedge A \in Y\}\end{aligned}$$

In the above equation, $RMatched(A)$ is the set of rules whose pre-condition is a superset of *currentLib* and whose post-condition contains A. If the set $RMatched(A)$ is empty, the recommendation score of A is 0. $RecScore^{RULE}$ ranges from 0 to 1. The libraries with the highest recommendation scores are the most suitable libraries based on the mined association rules.

[4] http://www.philippe-fournier-viger.com/spmf/index.php.

Example 6. Suppose a project has a set of libraries $P = \{a, b, c\}$ and we have the following rules: $R1 = e \rightarrow d$ with $conf(R1) = 1.0$, $R2 = a, b \rightarrow f, g$ with $conf(R2) = 0.9$, and $R3 = a \rightarrow f$ with $conf(R3) = 0.8$. $R2$ and $R3$ match P because their pre-conditions are a subset of P. We then want to compute $RecScore^{RULE}$ for the library f. Both $R2$ and $R3$ contain f in their post-conditions. Thus, $R2$ and $R3$ are matching rules. $RecScore^{RULE}(f)$ will then be the maximum of 0, 0.9 ($conf(R2)$), and 0.8 ($conf(R3)$), which is 0.9.

The pseudocode for the matching process is shown in Figure 3. At lines 8-9, for each association rule in *libRules*, we check whether *curLibraries* is a superset of the pre-condition of the rule. If it is, we iterate over the items in the rule's post-condition and add them to the list of recommended libraries (*recLibraries*) (lines 10-18). We iteratively update the recommendation scores of the libraries in *recLibraries*. Whenever a matching rule of higher confidence containing a recommended library is encountered, we update the recommendation score of the library (lines 14-16).

```
 1: Input:
 2:   $curLibraries$ = libraries that the target project uses
 3:   $libRules$ = association rules
 4: Output:
 5:   Recommended libraries w. recommendation scores
 6: Method:
 7: Let $recLibraries$ = {}
 8: for all $Rule \in libRules$ do
 9:   if $curLibraries \subseteq Rule.PreCondition$ then
10:     for all $A \in Rule.PostCondition$ do
11:       if $A \notin curLibraries$ then
12:         add $A$ and $A.Conf$ to $recLibraries$
13:       else
14:         if $recLibraries[A] < A.Conf$ then
15:           $recLibraries[A] = A.Conf$
16:         end if
17:       end if
18:     end for
19:   end if
20: end for
21: return $recLibraries$
```

Fig. 3. Rule Matching Procedure

C. $LibRec^{COLLAB}$ Component

Our $LibRec^{COLLAB}$ component recommends libraries based on those that are used by similar projects, following a nearest-neighbor-based collaborative filtering approach. We measure the similarity of two projects based on their set of commonly used libraries. $LibRec^{COLLAB}$ consists of the *FeatureVectorExtractor* sub-component in the training phase and the *NearestNeighborProcessor* sub-component in the deployment phase.

1) FeatureVectorExtractor: This component converts the list of libraries used by each project in *TrainingProjects* to a feature vector. Let aL be the set of all libraries arranged in alphabetical order of their names. Each library can then be assigned a unique index in aL and referred to as $aL[i]$. The feature vector of project A, denoted as $V(A)$, is defined as follows:

$$\begin{aligned} V(A) &= (ind(aL[0], A), \ldots, ind(aL[|aL|], A)) \\ ind(L, A) &= 1,\ If\ A\ uses\ library\ L \\ &\quad 0,\ Otherwise \end{aligned}$$

2) NearestNeighborProcessor: Given a new project to receive library recommendations, *NearestNeighborProcessor* converts the list of libraries that the project uses into a feature vector in the same manner as was done by the *FeatureVectorExtractor* component. It then calculates the distance between this feature vector and the feature vectors of projects in *TrainingProjects*. We use cosine similarity as the metric to compute this distance [19]. The cosine similarity of a new project New and an existing project $Existing$ in *TrainingProjects* is:

$$Cosine(New, Existing) = \frac{V(New) \cdot V(Existing)}{|V(New)||V(Existing)|}$$

In the above equation, $\cdot$ denotes dot product, and $|V(i)|$ denotes the size of a vector, which is defined as the square root of the sum of the squares of its constituent elements.

We rank the projects in *TrainingProjects* based on their cosine similarity scores. The higher the cosine similarity score is, the more similar a training project is to the new project. Therefore, we pick the top-n projects with the highest cosine similarity scores as the nearest neighbors for the new project. In the implementation, we sort the projects based on their cosine similarity scores followed by their names. If there are projects with rank greater than n that have the same cosine similarity score as the n-th project, we group the projects having this score, and randomly select projects from this group, to result in the final n nearest neighbors.

Our next step is to compute a recommendation score for each library. We collect all of the libraries that are used by projects in the n nearest neighbors and compute the score for each library. Given a library A, we compute the *collaborative-filtering-based recommendation score* for A as follows:

$$RecScore^{COLLAB}(A) = \frac{NNCount_{Lib}(A)}{n}$$

In the equation above, $NNCount_{Lib}(A)$ is the number of nearest neighbor projects that use library A and n is the number of nearest neighbors. $RecScore^{COLLAB}$ scores range from 0 to 1. The library with the highest $RecScore^{COLLAB}$ score is considered to be the most the most suitable library.

Example 7. Consider a project that has 3 nearest neighbors as follows: $P1 = \{ant, jsp\text{-}api, junit\}$,

$P2 = \{hsqldb, junit, commons\text{-}lang\}$, and $P3 = \{log4j, jetty, gson\}$. $junit$ is used in 2 out of 3 nearest neighbor projects. So, $NNCount_{Lib}(junit)$ is 2 and $RecScore^{COLLAB}(junit)$ is 0.67.

D. Aggregator Component

This component combines $RecScore^{RULE}$ and $RecScore^{COLLAB}$ to get an overall recommendation score, denoted as $RecScore$. The overall recommendation score of a library A is defined as follows:

$$RecScore(A) = \alpha \times RecScore^{RULE}(A) + \beta \times RecScore^{COLLAB}(A)$$

In the equation above, α and β represent weights for the two recommendation strategies. When $\alpha + \beta = 1$, $RecScore$ ranges from 0 to 1. By default, we set α and β to 0.5. The top-k libraries with the highest $RecScore$s are recommended to developers – again ties are randomly broken.

Example 8. Suppose that we have a library called z. Let $RecScore^{RULE}(z) = 0.8$ and a $RecScore^{COLLAB}(z) = 1.0$. $RecScore(z)$ is 0.9.

V. Experiments & Analysis

In this section, we describe our dataset, followed by our evaluation measures and procedure. We then present our research questions and the results of our experiments. Finally, we discuss some threats to validity.

A. Dataset

To construct the dataset, we first randomly collected a few thousand Java projects from GitHub.[5] We then filtered the collected projects based on the following criteria:

1) **The project contains more than 10,000 lines of code.** This is intended to filter out "toy projects". We counted the line of codes by using SLOCCount,[6] which excludes comments and whitespace.
2) **The project is not a fork of another project in Github.** A fork is essentially a clone of another project at a specific point of time. We do not want to consider both the original and the forked project, or multiple projects that are forked from the same source. The original and forked projects are likely to share the same libraries.
3) **The project is a Maven project.** Maven is a build automation tool. One of its features is the ability to declare library dependencies for a project. Libraries have a unique identifier that can help us to correctly identify the use of the library across multiple projects. We identify Maven projects by checking for the existence of pom.xml files in the project repository. From these xml files, we extracted the *GroupId* and *ArtifactId* of the libraries on which the project depends. The combination of these two identifiers forms a unique identifier for a library stored in the Maven repository.[7]
4) **The project uses at least ten libraries.** We focus on projects that rely on third-party libraries. Our recommendation system has more value for library-intensive projects.

After filtering projects that contain fewer than 10,000 lines of code, are forked from other projects, and are not Maven projects, we are left with 1008 projects. The distribution of the number of libraries used in these projects is shown in Figure 4. The minimum, maximum, and average number of libraries used in these projects are 0, 627, and 28.1 projects, respectively. To focus on projects that rely heavily on third-party libraries, we randomly selected 500 projects that use at least ten libraries as our experimental dataset. Projects included in this dataset include popular projects such as Tapestry5 (146.4 kLOC), Sonar (132.3 kLOC), JBoss (499.0 kLOC).[8]

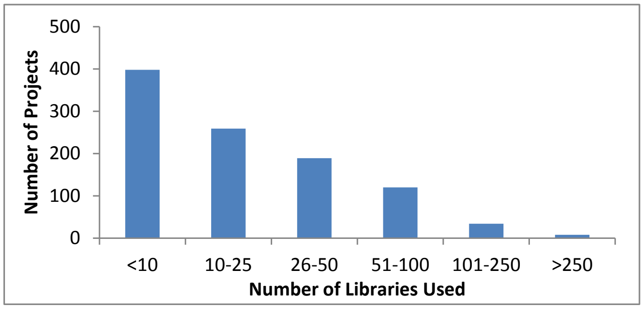

Fig. 4. Distribution of Library Usage in 1008 Projects

B. Evaluation Measures and Procedure

We evaluate our experiments using a well known evaluation metric, recall rate@k [22], [23], [27], [30], [36].[9] Consider m target projects that should receive library recommendations. For each project p_i, let the ground truth be the set of libraries GT_i. The recall rate@k of a library recommendation system that recommends a set of top-k libraries R_i for each of the projects p_i, is the proportion of recommendation R_i, in the set of all recommendations R (for all projects), that includes at least one library in the ground truth (i.e., $R_i \bigcap GT_i \neq \emptyset$). We use a small value for k as developers are unlikely to look through a long recommendation list.

We perform ten-fold cross validation to measure the accuracy of our approach. We randomly distribute the dataset into ten equal-sized parts. Each fold consist of nine parts of the dataset as the training data and the remaining part as the testing data. For each project in the test data, we drop half of

[5] http://github.com

[6] http://www.dwheeler.com/sloccount

[7] repo1.maven.org, http://search.maven.org

[8] The list of selected projects is available at: https://sites.google.com/site/autolibrec/projects

[9] Note that precision rate@k is not defined and is not used in past studies [22], [23], [27], [30], [36].

their libraries and use these as the ground truth. The remaining half are used as inputs to our recommendation approach. This methodology mimics the scenario where a developer knows some of the needed libraries but needs help to find other relevant libraries.

LibRec takes a number parameters: $minsup$, $minconf$, n (i.e., number of neighbors), α (i.e., weight of the $LibRec^{RULE}$), and β (i.e., weight of the $LibRec^{COLLAB}$). We set $minsup = 0.1$ and $minconf = 0.8$. We chose $minsup = 0.1$ because we do not want to miss specific rules that only exist in a small portion of the dataset. We chose $minconf = 0.8$ because we want the rule to have a high likelihood of being followed in a software project. We set the parameter n to 20, and the other parameters to their default values i.e., $\alpha = \beta = 0.5$.

C. Research Questions

We consider the following three research questions:

R1 How accurate is our proposed approach in recommending libraries to client applications?

R2 What are the effects of the various components and parameters of our approach on the overall accuracy?

R3 What is the impact of the various experimental settings on the overall accuracy?

D. RQ1: Accuracy of the Proposed Approach

We have performed a ten-fold cross validation on the 500 projects. The experiment shows that *LibRec* achieves a recall rate@5 of 0.852 out of 1.

E. RQ2: Effectiveness of Various Components and Parameter Settings

To answer this research question, we investigate the effectiveness of the two major components of *LibRec* and the sensitivity of *LibRec* to the various parameter settings.

Effectiveness of the Individual Components. We investigate the how well our individual components $LibRec^{RULE}$ and $LibRec^{COLLAB}$ work separately. The result is shown in Table II. Our $LibRec^{COLLAB}$ component performs better than our $LibRec^{RULE}$ component. Still, both of their recall rates are lower than that of *LibRec*. Thus, combining the two components is beneficial, as it improves accuracy.

TABLE II
EFFECTIVENESS OF INDIVIDUAL COMPONENTS

Component	Recall Rate@5
$LibRec^{RULE}$	0.702
$LibRec^{COLLAB}$	0.800

Effect of the Varying Number of Neighbors. We next investigate the sensitivity of our approach to the number of nearest neighbors taken into account. In practice, developers might not know the best number, and thus it is best if our approach is robust on different numbers of nearest neighbors, within a particular range. We vary the number of nearest neighbors from 5 to 25 and show the accuracy of *LibRec* in Table III. We find that the accuracy of our approach is relatively stable across different numbers of nearest neighbors (differences are less than 0.012). This shows the robustness of our approach.

TABLE III
EFFECT OF VARYING THE NUMBER OF NEAREST NEIGHBORS

Number of Nearest Neighbors	Recall Rate@5
5	0.840
10	0.848
15	0.850
20	0.852
25	0.848

Effect of Varying *minsup* and *minconf*. We next investigate the effect of varying *minsup*. As shown in Table IV, there is a small drop in accuracy when *minsup* is increased from 0.1 to 0.3 (about 0.05 reduction in recall rate@5). Increasing *minsup* eliminates some high confidence rules that apply to only a few projects. This reduces the effectiveness of our approach on these projects. The recall rate is relatively stable when we increase *minsup* from 0.3 to 0.4 and 0.5.

TABLE IV
EFFECT OF VARYING *minsup*

minsup	Recall Rate@5
0.1	0.852
0.2	0.816
0.3	0.796
0.4	0.800
0.5	0.800

We also investigate the effect of varying *minconf*. We notice that on very high *minconf* settings, there is a small drop in accuracy (about 0.07 drop in recall rate@5, when *minconf* is increased from 0.9 to 1.0). Raising the required confidence too high can cause good rules that might not apply in a few cases to be omitted. Note that *LibRec* uses the confidence of a matching rule to compute the rule-based recommendation score. Thus, we notice that reducing *minconf* from 0.9 to 0.8 results in little change in accuracy as *LibRec* takes a matching rule with the *highest* confidence.

TABLE V
EFFECT OF VARYING *minconf*

minconf	Recall Rate@5
0.8	0.852
0.85	0.852
0.9	0.848
0.95	0.824
1.0	0.778

F. RQ3: Effectiveness of Various Experimental Settings

To answer this research question, we investigate the sensitivity of *LibRec* to two experimental settings.

Effect of Varying Training Set Size. We vary the training set size by varying the value of k in k-fold cross validation. As k increases, the training set size also increases. The result for this experiment is shown in Table VI. We notice that the average recall rate@5 does not vary much on the training set size (differences are at most 0.014).

TABLE VI
EFFECT OF VARYING TRAINING SET SIZE

k **Fold**	**Recall Rate@5**
2	0.840
4	0.848
6	0.840
8	0.854
10	0.852

Effect of Changing Recall Rate@k. Finally, we investigate the effect of changing the value of k for recall rate@k. Intuitively, a larger k results in a higher recall rate. As shown in Table VII, the recall rate@1 is 0.616, and recall rate@3 is 0.804 which means, for most cases, correct recommendations appear early in the recommendation list.

TABLE VII
EFFECT OF VARYING RECALL RATE@K

Recommendation Size (i.e., k)	**Recall Rate @ k**
10	0.894
7	0.866
5	0.852
3	0.804
1	0.616

G. Threats to Validity

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.

Threats to external validity refers to the generalizability of our findings. Our dataset consists only of open source Java projects. Moreover, we pick only Java projects that use Maven. In practice, only a subset of Java developers use Maven to develop their applications. Even so, Maven is a popular tool in the Java developer community. We expect big projects that use many libraries to use Maven or similar tools to help manage their build process. We have already tried to minimize this threat by investigating 500 random projects. In the future, we plan to reduce this threat further by adding more projects.

Threats to construct validity refers to the suitability of our evaluation measure. Currently, we use recall rate@k to measure the effectiveness of our approach. This is a well known measure that is used in many past studies, e.g., [22], [30], [36].

VI. RELATED WORK

Mandelin et al. propose the problem of jungloid mining [18]. Given a query that describes the input and output types, jungloid mining returns code fragments that satisfy the query. Thummalapenta and Xie propose the tool ParseWeb that recommends code examples from the web [33]. Similar to jungloid mining, their tool accepts as input the source object type and the destination object type. It generates a sequence of methods that convert the source object type to the destination object type. Chan et al. [5] and Thung et al. [34] recommend API methods from queries expressed in natural language.

Robbes and Lanza analyze recorded program history to improve code auto-completion [25]. Hindle et al. propose a code auto-completion feature by investigating the "naturalness" of software [8]. A number of tools also support real-time code clone detection [9], [12]. These tools can also potentially be used to recommend code to developers.

Gall et al. detect logical couplings among software modules that are changed in a similar way [7]. Zimmermann et al. use association rule mining to infer that if changes are made to a set of program elements, then another set of program elements need to also be changed [38]. McCarey et al. recommend methods to a developer in a group of developers by investigating the history of methods that the developers have used before [20].

The above-mentioned studies recommend various code elements (method calls, blocks of code, etc) using various heuristics leveraging various information sources (source code, commit logs, etc.). Our study differs in the following respects:

1) We propose a novel hybrid approach that combines compact association rule mining and nearest neighbor based collaborative filtering to recommend libraries. Mandelin et al., Thummalapenta and Xie, and Robbes and Lanza use static analysis [18], [25], [33]. Hindle et al. use a statistical language model [8]. Kawaguchi et al. and Lee et al. use clone detection methods [9], [12]. Gall et al. use an algorithm that performs subsequence matchings [7]. Chan et al. use a graph analysis technique [5]. Thung et al. use a text analysis technique [34]. Zimmermann et al. also use association rule mining [38] however they do not integrate it with collaborative filtering. McCarey et al. also use collaborative filtering, however they use it in a different setting and they do not integrate it with association rule mining [20].
2) We leverage a different information source namely the lists of libraries used by other projects.
3) We consider a higher level of granularity. We recommend new libraries to client applications. Various libraries exist and developers are often unaware of their existence. Successful library recommendations can allow developers to reuse more code than successful code recommendations.

VII. CONCLUSION AND FUTURE WORK

Third party libraries can help to reduce software system development time. Developers can reuse the libraries to code some parts of the system rather than implementing them by themselves. Using well tested libraries also makes the system more reliable. Many third party libraries are publicly available. However, the large number of libraries makes it hard for developers to pick the relevant libraries that can improve their productivity.

We propose an automated technique to recommend relevant libraries to developers. Our approach combines association rule mining techniques and collaborative filtering to perform the recommendation. Based on the libraries used by other projects, we recommend a number of likely relevant libraries to developers of a target project. We have evaluated our approach on 500 open source Java projects hosted on GitHub. Our approach achieves a promising results with recall rate@5 and recall rate@10 of 0.852 and 0.894 respectively.

In the future, we plan to include more projects to further validate our results. We also plan to develop a better approach that can increase the recall rate. To achieve this, we plan to analyze cases where our approach and individual components are ineffective and make appropriate modifications to our approach. One approach could be to consider not only libraries but also abstractions of libraries, e.g., the domain that they address. Another approach could be to consider nonfunctional properties of the projects, such as the time at which they were developed. Yet another approach would be to take into account the textual descriptions of libraries, by employing advanced NLP techniques, e.g., [4], [35], or to analyze the usage specifications of these libraries inferred by specification mining techniques, e.g., [6], [10], [15], [16], [17], [28]. We would also like to extend our approach to be able to recommend libraries to projects that only use a small number of libraries or do not use any libraries at all.

In terms of our experimental method, we plan to experiment with various experimental settings, e.g., considering different numbers of projects being dropped, considering different number of libraries used, etc. We also plan to integrate our proposed approach in an IDE (e.g., Eclipse) and perform a user study.

Finally, we also want to integrate our approach with techniques that recommend specific methods to use in a library, e.g., [18], [25], [33], [34] and to recommend specific library versions.

ACKNOWLEDGEMENT

This project is partly supported by MERLION MS12C0014 grant.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of Int. Conf. on Very Large Data Bases*, 1994.

[2] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal, "Mining minimal non-redundant association rules using frequent closed itemsets," in *International Conference on Computational Logic*, 2000.

[3] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *ICSM*, 2012, pp. 483–492.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, 2003.

[5] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *SIGSOFT FSE*, 2012.

[6] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Trans. Software Eng.*, vol. 38, no. 2, 2012.

[7] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history." in *ICSM*, 1998.

[8] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *ICSE*, 2012, pp. 837–847.

[9] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "SHINOBI: A tool for automatic code clone detection in the IDE," in *WCRE*, 2009, pp. 313–314.

[10] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Mining message sequence graphs," in *ICSE*, 2011.

[11] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *SAC*, 2011, pp. 1317–1324.

[12] M.-W. Lee, J.-W. Roh, S. won Hwang, and S. Kim, "Instant code clone search," in *SIGSOFT FSE*, 2010, pp. 167–176.

[13] J. Li, H. Li, L. Wong, J. Pei, and G. Dong, "Minimum description length principle: Generators are preferable to closed patterns," in *AAAI*, 2006, pp. 409–414.

[14] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[15] D. Lo and S. Maoz, "Mining hierarchical scenario-based specifications," in *ASE*, 2009.

[16] ——, "Scenario-based and value-based specification mining: better together," in *ASE*, 2010.

[17] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," in *WCRE*, 2009, pp. 62–71.

[18] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *PLDI*, 2005, pp. 48–61.

[19] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[20] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: A recommender agent for agile reuse," *Artif. Intell. Rev.*, vol. 24, 2005.

[21] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *ICSE*, 2012, pp. 364–374.

[22] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *ASE*, 2012, pp. 70–79.

[23] J. N. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell, "Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering," in *RE*, 2004.

[24] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *ICDT*, C. Beeri and P. Buneman, Eds., 1999.

[25] R. Robbes and M. Lanza, "Improving code completion with program history," *Autom. Softw. Eng.*, vol. 17, no. 2, pp. 181–212, 2010.

[26] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[27] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007, pp. 499–510.

[28] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Trans. Software Eng.*, 2008.

[29] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," *Adv. Artificial Intellegence*, vol. 2009, 2009.

[30] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011, pp. 253–262.

[31] L. Szathmary, A. Napoli, and S. O. Kuznetsov, "Zart: A multifunctional itemset mining algorithm," in *CLA*, 2007.

[32] L. Terveen and W. Hill, "Beyond recommender systems: Helping people help each other," in *HCI in the New Millennium*, 2001.

[33] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *ASE*, 2007, pp. 204–213.

[34] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *ASE*, 2013.

[35] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting paraphrases of technical terms from noisy parallel software corpora," in *ACL/IJCNLP*, 2009.

[36] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008, pp. 461–470.

[37] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *ECOOP*, 2009, pp. 318–343.

[38] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes." in *ICSE*, 2004.