

A Multi-Metric Ranking Approach for Library Migration Recommendations

Hao He^{*†}, Yulin Xu^{*}, Yixiao Ma^{*}, Yifei Xu^{*}, Guangtai Liang[†], and Minghui Zhou^{*}

School of Electronic Engineering and Computer Science, Peking University, Beijing, China^{*}

Software Analysis Lab, Huawei Technologies Co., Ltd., China[†]

Email: {heh, kylinxyl, mayixiao, xuyifei, zhmh}@pku.edu.cn^{*}, liangguangtai@huawei.com[†]

Abstract—The wide adoption of third-party libraries in software projects is beneficial but also risky. Third-party libraries may have security vulnerabilities, may be abandoned by their maintainers, or may no longer align with current project requirements. Under such circumstances, developers need to migrate a library to *another library* with similar functionalities, but the migration decisions are often opinion-based and sub-optimal with limited information at hand. Therefore, several filtering-based approaches have been proposed to mine library migrations from existing software data, but they suffer from either low precision or low recall with different thresholds, which limits their usefulness in supporting migration decisions.

In this paper, we present a novel approach that utilizes multiple metrics to rank and therefore recommend library migrations. Given a library to migrate, our approach first generates candidate target libraries from a large corpus of software repositories, and then ranks them by combining the following four metrics to capture different dimensions of evidence from development history: Rule Support, Message Support, Distance Support, and API Support. We evaluate the performance of our approach with 773 migration rules (190 source libraries) that we borrow from previous work and recover from 21,358 Java GitHub projects. The experiments show that our metrics are effective to help identify real migration targets from other libraries, and our approach significantly outperforms existing works, with MRR of 0.8566, top-1 precision of 0.7947, top-10 NDCG of 0.7665, and top-20 recall of 0.8939. To demonstrate the generality of our approach, we manually verify the recommendation results of 480 most popular libraries, and confirm 661 migration rules from 231 of the 480 libraries with comparable performance. The source code and data are provided at: <https://github.com/Asdf0717/MigrationHelper>.

I. INTRODUCTION

Modern software systems rely heavily on third-party libraries for rich and ready-to-use features, reduction of development costs and increase in productivity [1], [2]. In recent years, the rise of open source software and the emergence of package hosting platforms, such as GitHub [3], Maven Central [4] and NPM [5], has led to an exponential growth of open source libraries. For example, the number of newly published JARs in Maven Central is 86,191 in 2010, 364,268 in 2015, and over 1.2 million in 2019 [4]. Nowadays, open source libraries can satisfy a diverse spectrum of development requirements, and are widely adopted in both commercial and open source software projects [6].

However, third-party libraries are known to cause problems in software maintenance, some of which require the project to migrate to an alternative library. First, third-party libraries may

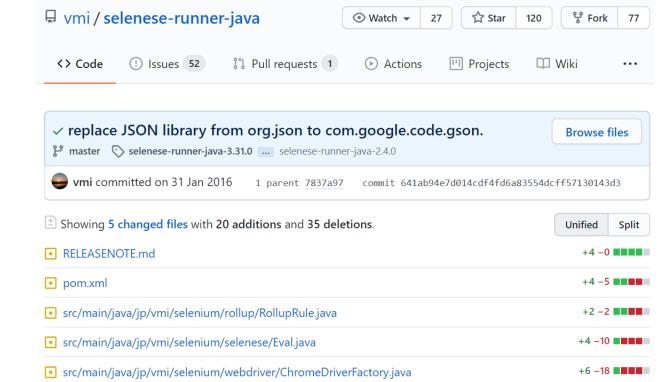


Fig. 1. An example migration between JSON libraries

have sustainability failures [7], [8]. They may be abandoned by their maintainers due to lack of time and interest, difficulty in maintenance, or being superseded by competitors [7]. Second, third-party libraries may have security vulnerabilities [9], [10], and in practice fixes are not always available at certain time or context. Finally, third-party libraries may fail to satisfy new requirements during project evolution because of features, performance, license, etc. A Java Web application at its infancy stage may be using `org.json:json` [11] for its simplicity and ease of use, but when the application matures and scales, it often has to migrate to `gson` [12] or `jackson` [13] for their richer features and higher performance (see Figure 1 for an example). In any of the above cases, a software project needs to replace the library with another similar library, which are called *library migration* in the literature [14]–[21].

However, it is often not easy to find good substitute libraries and choose between a number of candidate libraries [14], [22], [23]. Community curated lists such as `awesome-java` [24] and `AlternativeTo` [25] are often non-informative and contain low quality libraries, while blog posts and articles are often opinion based and outdated [22]. The easily accessible metrics such as popularity and release frequency have limited usefulness and they vary with domain [23]. In practice, industry projects often rely on domain experts for making migration decisions [26], while open source projects only migrate libraries when core developers reach a consensus in discussions [17]. In either case, the migration is not guaranteed to be appropriate, cost-effective or beneficial to the project.

To address this situation, researchers have proposed several approaches to mine library migrations from existing software data [14], [16], [19], based on the intuition that the presence of previous migrations provide valuable hints to developers on the worthiness of doing the same migration. Their approaches first mine candidate migration rules from a large number of projects, and then filter the candidates based on frequency [14] or related code changes [19]. However, their approaches suffer from either low recall [14], [19], or low precision [14], [16] for two reasons. First, a global filtering threshold is not equally effective for all migrations, thus hard to reach a performance balance. Second, valuable information sources are not considered in their filtering metrics. The usefulness of existing approaches is also limited, because a low precision will result in more human inspection effort, while a low recall may prevent developers from making the optimal decision because some migration opportunities may be missed.

Built upon the previous works [14], [16], [19], we propose a new approach for automatic recommendation of library migration targets from existing software development histories. Instead of mining and filtering, we formulate this problem as a *mining* and *ranking* problem, because we observe that relative ranking positions are not only more important but also more robust to metric and parameter changes. Given a library to be replaced, our approach first *mines* library candidates from dependency change sequences built from a large software corpus. After that, the candidates are *ranked* based on a combination of four carefully designed metrics: Rule Support, Message Support, Distance Support and API Support. The metrics are designed to capture different sources of evidence from data and pinpoint likely migration targets based on the evidences. Finally, the top target libraries, their metrics, and the relevant migration instances are returned for human inspection. Our approach can be effectively implemented as a web service, which can be used by project maintainers to seek migration suggestions or support their migration decisions.

To implement and evaluate our approach, we collect version control data of 21,358 Java GitHub repositories from the World of Code database [27], where we successfully extract 147,220 dependency change sequences from their dependency configuration files. To support metric computation, we also collect Maven artifacts from Maven Central [4] using Libraries.io [28], and extract API information for each collected Maven artifact. To evaluate the performance of our approach, we recover and extend migration rules in [16] from the collected 21,358 repositories and get a ground truth of 773 migration rules (190 source libraries). We use the 190 source libraries as query to our approach, which returns 243,152 candidate rules along with their metrics. The candidate rules include all ground truth migration rules mentioned above. Then, we estimate and compare survival functions for each metric, and compute Mean Reciprocal Rank (MRR) [29], top- k precision, top- k recall and top- k Normalized Discounted Cumulative Gain (NDCG) [30] for our approach along with existing approaches and a number of other baselines. Finally, we experiment our approach on additional 480 popular

libraries both to ensure our approach is generalizable and to confirm more real world migrations. Our experimental evaluations show that the metrics are effective to help identify real migration targets from other libraries, the method that combines all four metrics can reach MRR of 0.8566, top-1 precision of 0.7947, top-10 NDCG of 0.7665 and top-20 recall of 0.8939 in the ground truth dataset, and we confirm 661 new migration rules with comparable performance. Our initial deployment in an industry setting obtains positive feedback.

The main contributions of this paper are the following.

- 1) We formulate the migration recommendation problem as a *mining* and *ranking* problem, and propose a new approach for it based on mining dependency change sequences and ranking on four metrics, which significantly outperforms existing works.
- 2) We implement and systematically evaluate our approach for Java and Maven. We provide a ground truth dataset of 1,384 migration rules and 3,340 related commits discovered from 1,651 open source Java projects. The dataset can be used to facilitate further research in library migration.

II. BACKGROUND

In this section, we first provide a brief introduction to the context of our approach. After that, we define common terminologies and a dependency model used throughout the remainder of this paper. Then, we provide our definition to the library migration recommendation problem. Finally, we discuss the existing approaches provided by Teyton et al. [14], [16] and Alrubaye et al. [19].

A. Library Migration

Migration is a common phenomenon in software maintenance and evolution, and may refer to different development activities that stem from various motivations. Common cases of migrations in software development include: migrating from an legacy platform to a modern platform [31], [32], one programming language to another programming language [33]–[39], one library version to another library version [40], [41], one API to another API [15], [18], [42]–[47], or one library to another library [14], [16], [17], [19], [48], [49]. In this paper, we use the term **library migration** to refer to the process of replacing one library with another library of similar functionalities, as in [14]–[21].

Two steps are typically needed for a library migration. The first step is to decide which library to migrate to and whether a migration is worthy. The second step is to conduct the real migration by modifying API calls in source code, changing configuration files, etc. For example, Kabinna et al. [17] studied logging library migrations in Apache Software Foundation (ASF) projects. They discovered that ASF projects conduct logging library migrations for flexibility, new features and better performance, but some projects fail to migrate due to absence of submitted patches or lack of maintainers' consensus. Given current situation, the main objective of our approach is to provide *explainable* and *evidence based* support

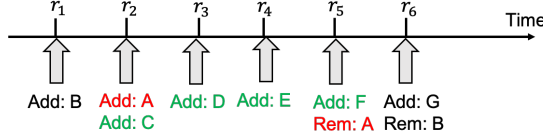


Fig. 2. An example dependency change sequence

to the first step of migration, such that developers can make the most appropriate migration decision for their project based on the recommendation results. As introduced earlier, the existing approaches to achieve this goal suffer from low precision or low recall, and limited usefulness in practice.

B. Terminologies and Dependency Model

Let \mathcal{P} be the set of projects and \mathcal{L} be the set of libraries to be analyzed. A project $p \in \mathcal{P}$ has a set of revisions $Rev(p) = \{r_1, r_2, \dots, r_n\}$, where each revision has zero, one or more parent revisions. We define $r_i < r_j$ if r_i happens before r_j in p , and vice versa. For each revision $r_i \in Rev(p)$, it depends on a set of libraries $L_i \subset \mathcal{L}$, which we call **dependencies** of r_i . By comparing with its parent revision(s) (e.g. r_{i-1}), we can extract the dependency changes of r_i as¹

$$L_i^+ = L_i - L_{i-1} \quad (1)$$

$$L_i^- = L_{i-1} - L_i \quad (2)$$

where L_i^+ is the set of added libraries and L_i^- is the set of removed libraries. By sorting all revisions in topological order and aggregating dependency changes, we can build a **dependency change sequence** (or **dep seq**) D_p for project p .

$$D_p = L_1^+, L_1^-, L_2^+, L_2^-, \dots, L_n^+, L_n^- \quad (3)$$

For library $a \in \mathcal{L}$ and library $b \in \mathcal{L}$, we consider (a, b) as a **migration rule** if and only if $\exists p \in \mathcal{P}, r_i, r_j \in Rev(p)$, p has conducted a migration from a to b between r_i and r_j , where $b \in L_i^+$ and $a \in L_j^-$. Note that a migration may either happen in one revision ($r_i = r_j$), or spans over multiple revisions ($r_i < r_j$). We refer to a as **source library** and b as **target library** in the subsequent paper.

Figure 2 shows an artificial example of a dependency change sequence where a project performs a migration from library A to F in revision r_5 . In this case, $L_5^+ = \{F\}$ and $L_5^- = \{A\}$, and (A, F) is a migration rule by our definition.

C. Problem Definition

Let R be the set of all migration rules. Given projects \mathcal{P} , libraries \mathcal{L} , dep seqs D_p for $p \in \mathcal{P}$, and a set of library queries (i.e. source libraries) Q , the objective of *library migration recommendation* is to identify migration rules with both high coverage (for objectivity) and accurate results (for minimizing human inspection effort). For library $a \in Q$, an approach should seek to find **migration targets** $T = \{b | (a, b) \in R\}$. We view this problem as a two-step mining and ranking problem. In the mining step, an approach should generate a **candidate**

rule set R_c from all dep seqs. In the ranking step, for all $(a, b) \in R_c$, it should compute a confidence value $conf(a, b)$ and sort R_c by this value. It should ensure that the target libraries in top ranked candidate rules are more likely to be real migration targets.

D. Existing Approaches

While several existing approaches address migration recommendation problem, they adopt the idea of mining and filtering. Teyton et al. [14] define candidate rules as the Cartesian product of added and removed libraries in the same revision. More formally, for project p ,

$$R_{ci}^p = \{(a, b) | (a, b) \in L_i^- \times L_i^+, r_i \in Rev(p)\} \quad (4)$$

$$R_c = \bigcup_{p \in \mathcal{P}} \bigcup_{r_i \in Rev(p)} R_{ci}^p \quad (5)$$

And they define confidence value as the number of revisions a rule occurs divides the maximum number of all rules with same source or same target as follows.

$$conf_T(a, b) = \frac{|\{r_i | (a, b) \in R_{ci}^p\}|}{\max(|\{(a, x) \in R_c\}|, |\{(x, b) \in R_c\}|)} \quad (6)$$

It captures most frequent rules and is partially reused in our first metric RS (Section III-B1). Their approach requires manual specification of a threshold t , and they consider a candidate rule (a, b) as a migration rule if and only if $conf_T(a, b) \geq t$. In their evaluation, by setting $t = 0.06$, they confirm 80 migration rules with a precision of 0.679 in 38,588 repositories.

In their subsequent work [16], they also consider migrations that spans over multiple revisions, using the following definition of candidate rules

$$R_c^p = \{(a, b) | (a, b) \in L_j^- \times L_i^+, r_i \leq r_j\} \quad (7)$$

$$R_c = \bigcup_{p \in \mathcal{P}} R_c^p \quad (8)$$

By this definition, (A, C) , (A, D) , (A, E) and (A, F) in Figure 2 will all be considered as candidate rules given A as source library. In the hope of covering as many migration rules as possible, they manually verify 17,113 candidate rules generated from 15,168 repositories, in which they successfully confirm 329 migration rules.

Alrubaye et al. [19] propose a tool called MigrationMiner, which uses the same candidate rule definition as in [14], but filters candidate rules by their relative frequency and whether API replacements occur in code changes. Their filtering strategy can be effectively described in our framework as Rule Support $RS(a, b) = 1$ and API Count $AC(a, b) > 0$ (See Section III for their definitions). The tool is tested on 16 repositories where 6 migration rules can be confirmed with 100% precision. Table I provides a summary and comparison of existing approaches along with ours.

III. OUR APPROACH

In this section, we introduce our approach for recommending library migration targets at a high level. As mentioned

¹We will discuss how we deal with multi-parent revisions in Section IV.

TABLE I
COMPARISON WITH MOST RELATED WORK [14], [16], [19]. SEE SECTION V FOR MORE PERFORMANCE COMPARISONS.

| Approach | Candidate Rules for A in Figure 2 | Migration Rule Selection | Performance |
|----------------------|-------------------------------------|------------------------------------|---|
| Teyton et al. [14] | (A, F) | $conf_T(a, b) \geq 0.06$ | 0.679 precision, 80 migration rules confirmed in 38,588 repositories |
| Teyton et al. [16] | $(A, C), (A, D), (A, E), (A, F)$ | All candidate rules | 0.019 precision, 329 migration rules confirmed in 15,168 repositories |
| Alrubaye et al. [19] | (A, F) | $RS(a, b) = 1 \wedge AC(a, b) > 0$ | 1.000 precision, 6 migration rules confirmed in 16 repositories |
| Our Approach | $(A, C), (A, D), (A, E), (A, F)$ | Rank by $conf(a, b)$ (Equation 16) | 0.795 top-1 precision, 1,384 migration rules confirmed in 21,358 repositories |

before, we tackle this problem using a mining step and a ranking step. We first describe how candidate rules are mined for each dep seq. Then we detail how we design the four ranking metrics. Finally, we describe the confidence value, ranking strategy, and the pseudo code for our final algorithm.

A. Mining Candidate Rules

Given a library a , the first step of our recommendation algorithm is to find a set of candidate rules $\{(a, x)\}$, in which x may be a feasible migration target. We follow a similar process like in [16] for mining candidate rules R_c (Equation 7 and 8), due to its high coverage. Since it is known to generate many false positives, we will specifically consider candidate rules mined from the same revision (R_{ci}^p in Equation 4) during metric computation. We also collect all the relevant revision pairs for each candidate rule (a, b) , defined as

$$Rev(a, b) = \{(r_i, r_j) | a \in L_j^- \wedge b \in L_i^+ \wedge r_i \leq r_j\} \quad (9)$$

B. Four Metrics for Ranking

1) *Rule Support*: For each candidate rule (a, b) , we define Rule Count $RC(a, b)$ as the number of times a is removed and b is added in the same revision:

$$RC(a, b) = |\{r_i | (a, b) \in R_{ci}^p, \forall p \in \mathcal{P}, r_i \in Rev(p)\}| \quad (10)$$

We define Rule Support $RS(a, b)$ as Rule Count divides the maximum value of Rule Count for all candidate rules with a as source library:

$$RS(a, b) = \frac{RC(a, b)}{\max_{(a, x) \in R_c} RC(a, x)} \quad (11)$$

This metric is basically a reuse of $conf_T(a, b)$ in Equation 6, based on the intuition that the most frequent same-revision dependency changes are more likely to be library migrations. We omit the second denominator because it can only be computed with a full-scale mining ($Q = \mathcal{L}$), which is costly given the current number of libraries.

2) *Message Support*: Besides the “implicit” frequency-based hint characterized by Rule Support, we observe that the “explicit knowledge” provided by developer written messages accompanying revisions (e.g. commit messages or release notes) is extremely valuable. Therefore, we design the following heuristic to determine whether a revision pair (r_i, r_j) seems to be doing a migration from a to b :

- 1) First, we split the names of a and b into possibly informative parts, because developers often use shortened names to mention libraries in these messages.
- 2) For $r_i = r_j$, we check whether its message is stating a migration from a to b .
- 3) For $r_i \neq r_j$, we check whether the message of r_i is stating the introduction of library b while mentioning a , and whether the message of r_j is stating the removal of a as a cleanup.

The checks are implemented via keyword matching of library name parts along with different hint verbs for migrations, additions, removals, and cleanups. For example, we consider words like “migrate”, “replace”, “switch” as hinting a migration. We also iteratively refined the keyword matching strategies using ground truth commits discovered in Section V.

Let $h(r_i, r_j) \mapsto \{0, 1\}$ be the heuristic function described above, where $h(r_i, r_j) = 1$ when the messages are considered as indicating a migration by our heuristic. We define Message Count $MC(a, b)$ as the number of revision pairs (r_i, r_j) whose messages are indicating a migration from a to b , where b is added in r_i and a is removed in r_j :

$$MC(a, b) = |\{(r_i, r_j) | (r_i, r_j) \in Rev(a, b) \wedge h(r_i, r_j)\}| \quad (12)$$

And we further define Message Support $MS(a, b)$ as

$$MS(a, b) = \log_2(MC(a, b) + 1) \quad (13)$$

3) *Distance Support*: The previous two metrics do not take into consideration multi-revision migrations without any meaningful messages, but such cases may still be common because not all developers write high quality messages. Based on the observation that most real target libraries are introduced near the removal of source library, we design the Distance Support metric to penalize the candidates that often occur far away. Let $dis(r_i, r_j)$ be the number of revisions between revision r_i and r_j (0 if $r_i = r_j$). We define Distance Support $DS(a, b)$ as the average of the inverse square distances of all revision pairs $(r_i, r_j) \in Rev(a, b)$:

$$DS(a, b) = \frac{1}{|Rev(a, b)|} \sum_{(r_i, r_j) \in Rev(a, b)} \frac{1}{(dis(r_i, r_j) + 1)^2} \quad (14)$$

4) *API Support*: Another strong information source of a library migration is the real code modifications and API replacements performed between the two libraries. To capture this, we define API Count $AC(a, b)$ as the number of code

hunks² in $Rev(a, b)$ where the APIs (i.e reference to public methods and fields) of b are added and the APIs of a are removed. Then, we define API Support as

$$AS(a, b) = \max(0.1, \frac{AC(a, b)}{\max_{(a, x) \in R_c} AC(a, x)}) \quad (15)$$

The reason for setting a minimum threshold is that we cannot detect any code change for some migration rules, either because the code changes are performed before or after, or because the migration does not require any code changes. We find that the choice of threshold values will not significantly impact the final performance in our ranking framework.

C. Recommending Migration Targets

Our recommendation algorithm combines the four metrics introduced above to generate a final confidence value for all candidate rules, using a simple multiplication as follows:

$$conf(a, b) = RS(a, b) \cdot MS(a, b) \cdot DS(a, b) \cdot AS(a, b) \quad (16)$$

For each library query $l \in Q$, the corresponding candidate rules are sorted by this confidence value from the largest to smallest. If the confidence values of some candidate rules are the same, we further sort them using other metrics. Finally, the recommendation results along with the relevant revision pairs, grouped by each library query, are returned for human inspection. Algorithm 1 provides a full description of our migration recommendation algorithm. After initialization of the required data structure (line 1-5), we generate the candidate rules and accumulate necessary data for metric computation in one iteration of all projects and their dep seqs (line 6-14). After that, the metrics and confidence values are computed through another iteration of all candidate rules (line 15-24).

Several important optimizations can be applied to Algorithm 1. First, the initialization in line 2-4 can be done lazily to avoid excessive memory consumption. Second, for a project with length n dep seq, the iteration in line 6 can be efficiently done in $O(n)$ by reversely traversing through the dep seq while maintaining a set of removed libraries and relevant revisions. Finally, we can precompute $AC(a, b)$ for all $(a, b) \in \mathcal{L} \times \mathcal{L}$ through one iteration of all project diffs, to avoid costly and often repetitive on-demand computation of $AC(a, b)$.

IV. IMPLEMENTATION

In this section, we go through important implementation details for our approach, including design considerations, implementation environment, and the data collection process. Figure 3 provides an overview of our implementation.

A. Design Considerations

We choose to implement our approach for Java projects because of Java's popularity and industrial importance, and also because previous works [14], [16], [19] are implemented for Java. We choose to mine dep seqs only for Maven [50] managed projects because it will make dependency extraction

²A hunk is a group of added and removed lines extracted by the diff algorithm of a version control system.

Algorithm 1 Recommending Library Migration Targets

Input: Projects \mathcal{P} , libraries \mathcal{L} and library queries Q .

Output: For $a \in Q$, R_c sorted by $conf(a, b)$.

```

1: Initialize:  $Rev(a, b) \leftarrow \emptyset$ 
2: for  $(a, b) \in \mathcal{L} \times \mathcal{L}$  do
3:   Initialize:  $RC(a, b) \leftarrow MC(a, b) \leftarrow DS(a, b) \leftarrow 0$ 
4:   Initialize:  $AS(a, b) \leftarrow 0.1$ 
5: end for
6: for  $p \in \mathcal{P}, a \in Q, b \in R_c^p, (r_i, r_j) \in Rev(a, b)$  do
7:    $Rev(a, b) \leftarrow Rev(a, b) \cup (r_i, r_j)$ 
8:   if  $r_i = r_j$  then
9:      $RC(a, b) \leftarrow RC(a, b) + 1$ 
10:  end if
11:   $MC(a, b) \leftarrow MC(a, b) + h(r_i, r_j)$ 
12:   $DS(a, b) \leftarrow DS(a, b) + 1/(dis(r_i, r_j) + 1)^2$ 
13:   $AC(a, b) \leftarrow AC(a, b) + \text{getAPICount}(r_i, r_j)$ 
14: end for
15: for  $(a, b) \in R_c = \bigcup_{p \in \mathcal{P}} R_c^p$  do
16:   $RS(a, b) \leftarrow RC(a, b) / \max_{(a, x) \in R_c} RC(a, x)$ 
17:   $MS(a, b) \leftarrow \log_2(MC(a, b) + 1)$ 
18:   $DS(a, b) \leftarrow DS(a, b) / |Rev(a, b)|$ 
19:   $AS'(a, b) \leftarrow AC(a, b) / \max_{(a, x) \in R_c} AC(a, x)$ 
20:  if  $AS'(a, b) > AS(a, b)$  then
21:     $AS(a, b) \leftarrow AS'(a, b)$ 
22:  end if
23:   $conf(a, b) = RS(a, b) \cdot MS(a, b) \cdot DS(a, b) \cdot AS(a, b)$ 
24: end for
25: return For  $a \in Q$ ,  $R_c$  sorted by  $conf(a, b)$ 

```

trivial by parsing the `pom.xml` files. Each `pom.xml` file has a dependency section where developers can declare used libraries by stating their library group IDs, artifact IDs and version numbers. During compilation, most declared libraries are downloaded from Maven Central [4], which provides a central hosting service for most Java open source libraries indexed as “artifacts.” We consider artifacts with the same group ID and artifact ID as one library, and the version string as marking the different versions of a library. We do not consider version number during recommendation. Although multiple (group ID, artifact ID) may refer to the same library, we do not handle such aliases, because we observe that a renaming of group ID and artifact ID often accompanies important library changes, such as major version updates, intentional dependency shadowing, etc. Therefore, we consider the migrations between such aliases as eligible migration rules. However, we do not consider target libraries with the same group ID as source library, because we observe that they are often siblings, sub-components or auxiliary libraries that are very unlikely to be a real migration target.

B. Implementation Environment

We implement our approach on a Red Hat Linux server with 2 Intel Xeon E5-2630 v2 CPUs, 400GB RAM and 20TB storage. It has access to World of Code [27], a database for storing open source version control data including almost all

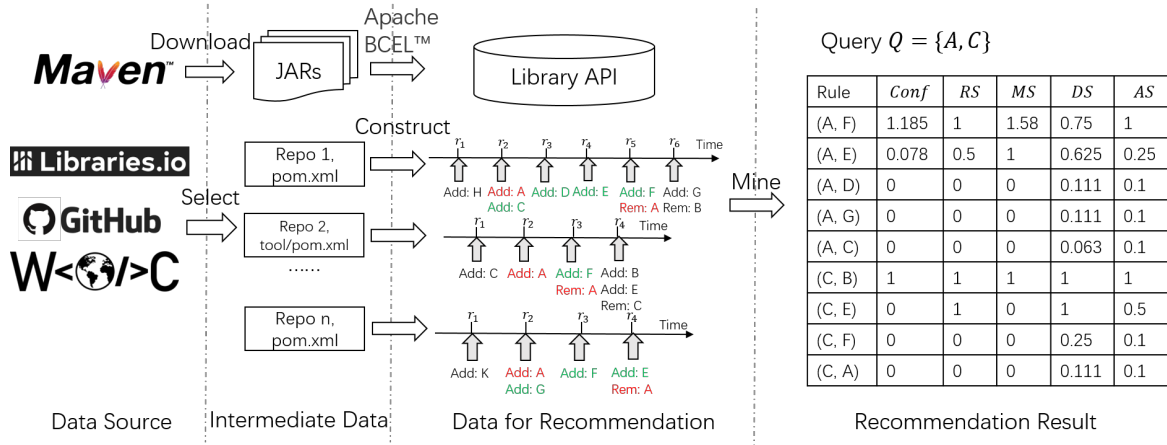


Fig. 3. Overview of the approach implementation, with some example data and results.

repositories from GitHub. We use World of Code because it offers much higher performance when constructing dep seqs, compared with directly cloning from GitHub and analyzing with `git`. The libraries and APIs are collected using Java, and the project dep seqs are constructed using Python. They are all stored in a local MongoDB [51] instance. The final recommendation service is implemented in Java using the Spring framework [52].

C. Data Collection

1) *Library Metadata and API Retrieval*: We use the Libraries.io dataset [28] (last updated in January 2020) to get a list of Maven artifacts and their metadata. After updating the latest version information from Maven Central in October 2020, we get a total of 184,817 distinct libraries for mining candidate rules ($|\mathcal{L}| = 184,817$), and 4,045,748 distinct library versions for API extraction.

Then, we download the corresponding JAR file from Maven Central for each version, if it has one. The total size of downloaded JAR files is ~ 3 TB, but they can be safely deleted after the APIs are extracted. For each JAR file, we use Apache Commons BCEL™ library [53] to extract all public classes along with their fields and methods and store each class as a document in MongoDB. Each document describes a unique and compact class signature object describing its fields, methods, inheritance relationships, etc. They are indexed by SHA1 computed from all its properties. We also maintain mappings between library versions and classes. Finally, we get 25,272,024 distinct classes in total. The whole download and extraction process takes about 3 days to finish with 16 parallel threads on our server.

2) *Dep Seq Construction*: We use the GitHub repository list provided by the Libraries.io dataset, in which we select non-fork repositories with at least 10 stars and one `pom.xml` file, resulting in 21,358 repositories. The repository commits and blobs are then retrieved and analyzed from World of Code version R (last updated April 2020). A repository may have multiple `pom.xml` files in different paths for different sub-

projects or sub-modules, so we consider each file as tracking the dependency of one project, and choose to construct one dep seq for each of the `pom.xml` files. For each `pom.xml` file, we extract all commits where this file is modified, separately parse its old version and new version to see whether any library is added or removed. For merge commits, we only compare with one of its old version due to design and performance constraints. For parallel branches, we sort changes by time and merge them into one dep seq. We then clean duplicate changes introduced by merge commits or parallel branches. After filtering out dep seqs with only one revision, we finally get 147,220 different dep seqs ($|\mathcal{P}| = 147,220$).

3) *API Count Precomputation*: As mentioned in Section III-C, it is worthwhile to precompute an API Count table for each $(a, b) \in \mathcal{L} \times \mathcal{L}$, because on-demand computation during recommendation is not only time-consuming but also inefficient in that many Java file diffs will be analyzed multiple times for different candidate rules. Therefore, we iterate over all Java file diffs for each repository, while maintaining a set of current candidate rules, to see whether some of them should increment their API Count values. The whole computation can be done in a highly parallel manner both for each repository and for code analysis of different Java file pairs. In the end, we get 4,934,677 library pairs with non-zero AC values. The whole computation takes about two weeks to finish with 16 threads, but we believe it can be further optimized and only needs to be run once, because the table can be incrementally updated when new repository data come in. Table II provides an overview of the final recommendation database we use.

V. EVALUATION

In this section, we first introduce two research questions for evaluation. Then, we describe how we build two ground truth datasets for different evaluation purposes. At last we show the evaluation methods and results for each research question.

A. Research Questions

Our main goal is to evaluate the effectiveness of Algorithm 1 in recommending library migration targets. Since the algorithm

TABLE II
STATISTICS OF THE RECOMMENDATION DATABASE

| Data Type | Count or Size | Time to Construct |
|----------------------------|---------------|-------------------|
| GitHub repositories | 21,358 | Several minutes |
| Commits with diffs | 29,439,998 | About 1 day |
| Parsed pom.xmls | 10,009,952 | About 1 day |
| Dep seqs | 147,220 | Several Hours |
| Libraries | 185,817 | Several Minutes |
| Library versions | 4,045,748 | Several Hours |
| Java classes | 25,272,024 | About 3 days |
| Non-zero API counts | 4,934,677 | About 2 weeks |
| Database size (compressed) | ~100GB | About 3 weeks |

mainly relies on a confidence value computed from four metrics (Equation 16), we first verify the effectiveness of each proposed metric, forming the first research question:

- RQ1: How effective is RS , MS , DS , and AS in identifying real migration targets?

Then, to show that our approach has good overall performance, outperforms existing approaches, and generalize well in different datasets, we ask the second research question:

- RQ2: What is the performance of our approach in the ground truth datasets, compared with baselines and existing approaches?

B. Ground Truth

We use two ground truth datasets for different evaluation purposes. The first is borrowed from previous work [16] and recovered in the 21,358 repositories we collected. This set of ground truth has reasonably good coverage for the source libraries in it, so we can compute accurate performance metrics and compare different approaches. The second is manually verified from the recommendation results of our algorithm, with 480 most popular libraries as queries, which we use to verify the generalizability of our approach and confirm more real word migrations. In the subsequent section, we refer to the first ground truth set as GT2014 and the second as GT2020.

1) *GT2014*: We choose to recover a new ground truth of migration rules based on the ground truth in [16], as the current situation may have significantly changed since its publication. The ground truth in [16] consists of 329 “abbreviated” library name pairs which may correspond to multiple group IDs and artifact IDs. They solve the aforementioned library aliasing issue (Section IV-A) through manual abbreviation, but we consider it too costly to repeat this on the 180k libraries we collected. As the Maven artifact pairs before abbreviation is not provided in [16], we choose to map the rules back to group IDs and artifact IDs. To cover as much migration rules as possible, we consider all pairs in Cartesian product as possible rules for multiple mappings. We also conduct reverse and transitive expansion of the rules until saturation (i.e. $(a, b) \in R \Rightarrow (b, a) \in R$, $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$), based on the intuition that libraries within one functionality category can be replaced with any other library in the same category. We get 3,878 “possible” migration rules after extension.

Recall in Section II-B that we define a migration rule as a rule that we can confirm at least one migration in one project,

so the rules collected before may still contain many false positives by this definition. Therefore, we collect all relevant commits for these rules, filter them by commit messages, and manually label the rules using the following criteria:

- 1) If library a and library b provide obviously different functionalities, we label (a, b) as false.
- 2) For other rules, we manually check the “most possible” commits by applying the message matching algorithm in Section III-B2. If there exists some messages stating a migration from a to b , we label (a, b) as true.
- 3) Otherwise, we check the commit diffs for easily understood small commits, and only label (a, b) as true when we can find at least one understandable commit performing a migration from a to b .

The labelling process is done by three of the authors with at least one year of Java development experiences. To mitigate threats brought by manual labelling, we adopt a conservative labelling strategy and ensure all rules with true labels are double checked by the author who has 3 years of Java experiences and 1 year of industry experiences. We also manually check the top-20 recommendation results and add any new migration rules using the same labelling process. Finally, we get a ground truth of 773 confirmed migration rules with 190 different source libraries, distributed in 2,214 commit pairs from 1,228 repositories. This set of migration rules are mainly used for performance evaluations and comparison with other approaches, and denoted as R_t in the remainder of this section.

In RQ1 and RQ2, we use the aforementioned 190 source libraries in R_t as the query to our approach ($|Q| = 190$), which returns 243,152 candidate rules ($|R_c| = 243,152$) along with the metrics, commits and confidence values. Note that the number of candidate rules is significantly larger than the size of ground truth rules ($|R_t| = 773$), which makes an effective ranking absolutely necessary. The recommendation only takes several minutes to finish with precomputed API Count data.

2) *GT2020*: We select the most popular 500 libraries based on the number of repositories they have been added. After filtering out existing source libraries in R_t , we get 480 libraries. We use them as query to our approach and collect their recommendation output, resulting in 383,218 candidate rules. To ensure that the migration rules we confirm are valuable for a broad audience and keep a reasonable amount of human inspection effort, we only inspect candidate rules that:

- 1) occur in top-20 recommendation result,
- 2) have been added in more than 10 repositories,
- 3) have non-zero confidence values.

After the filtering, we need to label 4,418 candidate rules in 12,565 pom.xml file changes, 2,353 commit pairs and 1,313 repositories. We repeat the labelling process as described before, and we successfully confirm 661 migration rules in 1,233 commits and 785 repositories, for 231 (48.125%) of the 480 queries. In RQ2, we will also use GT2020 to verify and compare the performance of our approach with existing approaches for the 231 libraries, despite its incompleteness.

TABLE III
PERFORMANCE COMPARISON OF DIFFERENT APPROACHES IN GT2014

| Approach | MRR | Precision@1 | NDCG@10 | Recall@20 |
|-----------------|--------|-------------|---------|-----------|
| Teyton et al. | 0.7133 | 0.6368 | 0.5972 | 0.7257 |
| Teyton et al.’ | 0.7335 | 0.6757 | 0.6952 | 0.6391 |
| Teyton et al.” | 0.8858 | 0.8737 | 0.4407 | 0.1759 |
| Alrubaye et al. | 0.8461 | 0.8462 | 0.5440 | 0.0142 |
| RS Only | 0.7208 | 0.6474 | 0.6025 | 0.7270 |
| MS Only | 0.7619 | 0.6737 | 0.6548 | 0.7736 |
| RS · MS | 0.8275 | 0.7579 | 0.7481 | 0.8616 |
| RS · MS · DS | 0.8401 | 0.7737 | 0.7630 | 0.8680 |
| RS · MS · AS | 0.8379 | 0.7737 | 0.7490 | 0.8745 |
| Our Approach | 0.8566 | 0.7947 | 0.7665 | 0.8939 |

By merging GT2014 and GT2020, we get 1,384 migration rules from 14,334 pom.xml changes, 3,340 commit pairs and 1,651 repositories (7.73% of the 21,358 repositories). If we consider top 20% largest repositories in terms of commit numbers, 1,092 out of 4,271 repositories (25.57%) have undergone at least one library migration. Both percentages are higher than the reported percentage in [16] (5.57% and 9.95%), indicating that library migrations are more prevalent in Java open source projects since 2014. The dataset is available on our project website (see Abstract).

C. RQ1: How effective is RS, MS, DS, and AS in identifying real migration targets from other libraries?

To answer RQ1, we plot survival functions for the confidence value of the approach in [14], the four proposed metrics, and the final confidence value of our approach, respectively. For metric M , its survival function is defined as $y = P(M \geq x)$, where $P()$ is the probability function. We estimate the probabilities using GT2014, for both ground truth rules R_t and other candidate rules $R_c - R_t$ in Figure 4. We can observe in Figure 4 that ground truth rules generally have significantly higher probability of survival for all metrics, indicating the effectiveness of each metric. We can also observe that any filtering threshold x will result in either significant ground truth losses or undesirable survival of many other candidate rules due to their much larger quantity. Therefore, despite the effectiveness of each metric, we conclude that any filtering based method is inherently ineffective and we should leverage relative ranking positions instead of absolute metric values for selecting migration rules from the large number of candidates.

D. RQ2: What is the performance of our approach compared with baselines and existing approaches?

For performance evaluation, we use the following quality metrics common for evaluating ranking problems: Mean Reciprocal Rank (MRR) [29], top- k precision, top- k recall, and top- k Normalized Discounted Cumulative Gain (NDCG) [30]. These metrics have also been used to evaluate other recommendation problems in software engineering [54]–[57].

Mean Reciprocal Rank (MRR) is defined as the mean of the multiplicative inverse of the rank of the first ground truth

rule for all queries:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\min_{(q,x) \in R_t} \text{rank}(q,x)} \quad (17)$$

It ranges between $[0, 1]$, and a higher value means that the user can see the first ground truth rule more quickly in general for each query.

Let R_k be the top- k ranked candidate rules for all queries, we define top- k precision and top- k recall as:

$$\text{Precision}@k = |R_k \cap R_t| / |R_k| \quad (18)$$

$$\text{Recall}@k = |R_k \cap R_t| / |R_t| \quad (19)$$

Normalized Discounted Cumulative Gain (NDCG) measures to what extent the ranking result deviates from the ideal result. For each library $l \in Q$ and its top- k returned rules R_{lk} , let $r(i) = 1$ if the rank i is in ground truth and 0 otherwise, we can define its Discounted Cumulative Gain (DCG) and Ideal Discounted Cumulative Gain (IDCG) as:

$$DCG_l@k = \sum_{i=1}^k \frac{r(i)}{\log_2(i+1)} \quad (20)$$

$$IDCG_l@k = \sum_{i=1}^{|R_{lk} \cap R_t|} \frac{1}{\log_2(i+1)} \quad (21)$$

And NDCG@ k is defined as:

$$NDCG@k = \frac{1}{|Q|} \sum_{l \in Q} \frac{DCG_l@k}{IDCG_l@k} \quad (22)$$

which ranges in $[0, 1]$ where 1 means a perfect match with ideal top- k ranking result and 0 means the worst.

We first compare performance of our approach with the following alternatives using GT2014:

- **Teyton et al.** [14]: Use conf_T (Equation 6) for ranking.
- **Teyton et al.’** [14]: Keep rules $\text{conf}_T \geq 0.002$ and rank.
- **Teyton et al.”** [14]: Keep rules $\text{conf}_T \geq 0.015$ and rank.
- **Alrubaye et al.** [19]: Filter out $RS < 1$ or $AC = 0$.
- **RS Only**: Use only RS for ranking.
- **MS Only**: Use only MS for ranking.
- **RS · MS**: Use $RS \cdot MS$ for ranking.
- **RS · MS · DS**: Use $RS \cdot MS \cdot DS$ for ranking.
- **RS · MS · AS**: Use $RS \cdot MS \cdot AS$ for ranking.

Some metric combinations are omitted because we find DS and AS only work well in combination with RS and MS . We only show MRR, top-1 precision, top-10 NDCG and top-20 recall in our paper, due to space constraints.

The performance comparison results are shown in Table III. Our approach can outperform all alternatives in term of top-20 recall and top-10 NDCG. What’s more, our approach achieves comparable top-1 precision and MRR with the most aggressive filtering based approaches while keeping a high recall. Although Teyton et al.” and Alrubaye et al. have higher precision, their recall is significantly lower than other approaches. The comparison with other metric combinations also shows that all four metrics are necessary to achieve the best performance. Therefore, we conclude that our approach achieves the best

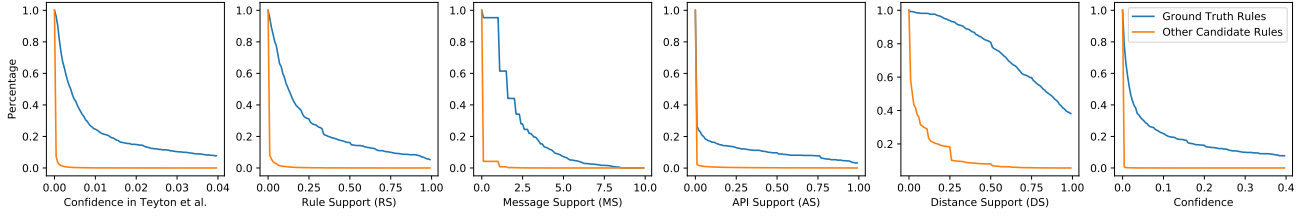


Fig. 4. The survival functions of ground truth rules and other rules. Note that the 0.1 minimum threshold is not applied for *AS* here.

TABLE IV
PERFORMANCE OF OUR APPROACH IN GT2020

| Approach | MRR | Precision@1 | NDCG@10 | Recall@20 |
|----------------|--------|-------------|---------|-----------|
| Teyton et al.’ | 0.6985 | 0.6123 | 0.6429 | 0.8052 |
| Our Approach | 0.7902 | 0.6826 | 0.7733 | 1.0000 |

overall performance and is the most suitable one for migration recommendation, compared with the evaluated alternatives.

To demonstrate the generality of our approach, we further compute the quality metrics using GT2020 for the 231 queries. Since GT2020 is incomplete, we also show performance of Teyton et al.’ for comparison. Note that precision and MRR will be lower than their real values, and top-20 recall is 1 because we only check until top-20 results. However, we can still observe that our approach outperforms Teyton et al.’ with a similar margin as in Table III. For example, the MRR of our approach is 13.1% higher than Teyton et al.’ in Table IV and 16.8% higher in Table III. Therefore, we conclude that our approach generalizes well in an unknown dataset.

VI. RECOMMENDATION EXAMPLE

Table V presents one recommendation example from our results: `c3p0:c3p0`. It is a database connection pooling library which has been renamed to `com.mchange:c3p0` since version 0.9.2 in 2007. Many other competitors for database pooling also emerged from 2007, under different licenses or for specific scenarios. We choose `c3p0:c3p0` as an example not only because of the diversity of migration targets returned, but also because it well demonstrates common failures and peculiarities of our approach. We can see from Table V that the top-1 recommended target is `com.mchange:c3p0`, with much higher confidence than the rest. The second result is Hikari CP, a database pooling library since 2013, which boasts its light-weight and much higher performance compared with other libraries. The 5th and 10th result is Alibaba Druid, a connection pool designed specifically for monitoring purposes. The 7th result is `c3p0` for integrated use with Hibernate. The 15th result is Apache Commons DBCP, a pooling library under the ASF license. The rest of the results are omitted in the Table because they are all false positives. Given the prevalence of migrations from `c3p0:c3p0`, a developer should also consider abandoning it for his/her project, and choose one of the above libraries instead after evaluating the migration costs, benefits, licenses and other factors through investigation of the migration commits and other available information.

VII. LIMITATIONS

A. Failure Case Analysis

Cold Start. The most obvious failure case in our approach is that, it can only recommend migration targets that have been frequently migrated in the project corpus. However, this limitation can be mitigated by using a large project corpus, because the more worthy a migration is, the more common it will be in such a corpus. As shown in [17], 33 of 223 ASF projects have undergone at least one logging migration, mostly from ad-hoc logging libraries (e.g. `log4j`) to log abstraction libraries (e.g. `slf4j`) and log unification libraries (e.g. `logback`), and our approach performs well in these popular migrations. Developers can also conclude that a library is not a worthy migration target if it is absent in our result, even if it provides similar functionalities.

Data Sparsity. Other failure cases are generally caused by the sparsity of many real migrations in our data. If only one or several relevant commits can be identified for a query, other added libraries in these commits tend to occur as false positives together with the true migration target (e.g. `hibernate-core` in Table V). *MS* and *AS* can handle such cases, but they sometimes fail either because developers do not write informative commit messages, or because code changes are not needed for the migration or are completed before/after the `pom.xml` changes.

Miscellaneous. Our approach will fail if a library has no substitute at all, and it cannot issue any warnings in such cases. Our method of matching commit messages sometimes falsely identify a migration, and can be improved by more sophisticated NLP techniques. Our approach also frequently returns Bills of Materials, libraries that warp other libraries, or “full solution” frameworks like Spring in our results. Even if they are not considered as migration targets during the labelling process, they may still be useful for developers.

B. Threat to Validity

1) *Construct Validity:* We cannot guarantee that the metrics we design and the parameters we choose are optimal. To avoid over-fitting on a single dataset, we choose the simplest possible form for each metric, and evaluate our approach on an additional dataset. For evaluation, it is generally impossible to compute accurate recall for all possible migration targets in the wild [14], [16]. Even if we limit the evaluation on a set of ground truth libraries, we cannot guarantee completeness of the identified migration targets. To mitigate this threat, we

TABLE V
RECOMMENDATION RESULTS OF c3p0:c3p0. MIGRATION COMMITS ARE OMITTED HERE DUE TO SPACE CONSTRAINTS

| Rank | Is Correct | Library | Confidence | <i>RS</i> | <i>MS</i> | <i>DS</i> | <i>AS</i> |
|------|------------|---|------------|-----------|-----------|-----------|-----------|
| 1 | True | com.mchange:c3p0 | 0.4083 | 0.9880 | 4.1700 | 0.9910 | 1.0000 |
| 2 | True | com.zaxxer:HikariCP | 0.0124 | 0.0238 | 1.0000 | 0.5222 | 0.1000 |
| 3 | False | org.jboss.jbossts.jta:narayana-jta | 0.0071 | 0.0357 | 2.0000 | 1.0000 | 0.1000 |
| 4 | False | org.springframework.boot:spring-boot-starter-test | 0.0050 | 0.0595 | 1.0000 | 0.8518 | 0.1000 |
| 5 | True | com.alibaba:druid | 0.0039 | 0.0476 | 1.0000 | 0.8222 | 0.1000 |
| 6 | False | org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec | 0.0038 | 0.0238 | 1.5849 | 1.0000 | 0.1000 |
| 7 | True | org.hibernate:hibernate-c3p0 | 0.0037 | 0.0357 | 1.5849 | 0.5430 | 0.1000 |
| 8 | False | org.hibernate:hibernate-core | 0.0030 | 0.0476 | 1.5849 | 0.3870 | 0.1000 |
| 9 | False | org.springframework.boot:spring-boot-starter-web | 0.0029 | 0.0357 | 1.0000 | 0.7539 | 0.1000 |
| 10 | True | com.alibaba:druid-spring-boot-starter | 0.0026 | 0.0238 | 1.0000 | 1.0000 | 0.1000 |
| 11 | False | org.modshape:modshape-web-explorer | 0.0024 | 0.0238 | 1.0000 | 1.0000 | 0.1000 |
| 12 | False | org.springframework.boot:spring-boot-devtools | 0.0024 | 0.0238 | 1.0000 | 1.0000 | 0.1000 |
| 13 | False | org.springframework.boot:spring-boot-starter-thymeleaf | 0.0024 | 0.0238 | 1.0000 | 1.0000 | 0.1000 |
| 14 | False | org.springframework.boot:spring-boot-starter-aop | 0.0024 | 0.0238 | 1.0000 | 1.0000 | 0.1000 |
| 15 | True | commons-dbcp:commons-dbcp | 0.0012 | 0.0238 | 1.0000 | 0.3199 | 0.1667 |

aggressively extend the largest set of migration rules from existing work [16] and validate all extended rules, to present a “best effort” estimation of recall for GT2014 in RQ2.

2) *External Validity*: Our approach may not generalize to a different dataset (e.g. industry projects) and to other programming languages and library ecosystems. We mitigate this threat by collecting a large number of open source Java projects and libraries, and testing on many library queries. Also, our approach in Section III does not make any language specific assumptions and can be easily re-implemented for other programming languages. Others can also refer to Section IV as an example of how to make implementation choices for a specific language or library ecosystem.

VIII. RELATED WORK

In this section, we go through related works not discussed before, and we summarize the relationship, differences or improvements of our work compared with theirs.

Bartolomei et al. [48], [49] studied the feasibility, peculiarities, and design patterns of using API wrappers to migrate between two XML libraries [48] and two Java GUI libraries [49]. However, it is generally costly to implement a wrapper and nearly impossible to implement wrappers for all possible migrations.

To recommend libraries to developers, Thung et al. [54] propose to recommend libraries based on the libraries a project is already using, through association rule mining and collaborative filtering. More approaches are proposed for this problem later, such as multi-objective optimization [57], hierarchical clustering [58], advanced collaborative filtering [59], matrix factorization [60], etc. However, their main objective is to recommend missed reuse opportunities based on the libraries a project is using and the properties of the project, not to recommend migration opportunities of a library already in use. Chen et al. [22] mine semantically similar libraries by training word embedding on Stack Overflow tags. Given a library, their method can return a list of possibly similar libraries, but it provides no evidence on the feasibility and prevalence of migrations between this library and the returned libraries.

Mora et al. [23] study the role of common metrics in developer selection of libraries. Alrubaye et al. [20] analyze several code quality metrics before and after library migration. They all use existing metrics for their study and the metric cannot be used to recommend library migration. By contrast, we specifically design new metrics to mine and recommend migrations from existing software data.

A number of studies have proposed methods for mining API mappings of two similar libraries [15], [18], [21], [44], [47], [61] or directly editing code to use the new API [62], [63]. Zheng et al. [61] mine alternative APIs from developer shared knowledge in forums or blog posts. Gokhale et al. [44] build mappings by harvesting the execution traces of similar APIs. Teyton et al. [15] find API mappings by analyzing their co-occurring frequencies in existing migrations. Alrubaye et al. improve their method using information retrieval techniques [18] and machine learning models [21]. Chen et al. [47] propose an unsupervised deep learning based approach that embeds both API usage patterns and API descriptions. Xu et al. [62] propose an approach to infer and apply migration edit patterns from existing projects. Collie et al. [63] propose to model and synthesize library API behavior without prior knowledge, which can be used for identifying migration mappings and applying migration changes. The output of our approach can serve as input for any of the approaches above, because they all require manual specification of library pairs. Our new migration dataset may also be helpful in facilitating further research into this area.

Other recent works also aim to aid dependency management, but from a different perspective, such as characterizing library usage [6], versioning [64]–[66] and update behavior [41], discovering considerations for library selection [23], [26], resolving version conflicts in dependency tree [67], [68], in multi-module projects [69], and so on.

IX. CONCLUSION AND FUTURE WORK

In this paper, we propose an approach for automated recommendation of library migration targets, through mining dependency change sequences of a large corpus of software

projects. We formulate this problem as a mining and ranking problem and design four advanced metrics for ranking: Rule Support, Message Support, Distance Support and API Support. To the best of our knowledge, our approach achieves best performance in this problem domain. We also verify and build a latest migration dataset for further research.

In the future, we plan to improve our approach to overcome current limitations (Section VII). More importantly, we plan to deploy our tool in more complicated industry context and collect usage feedback from industry developers, while extending the migration rule dataset in a crowd-sourced manner. Finally, we plan to systematically investigate developer considerations for library migration, using the collected migration dataset.

REFERENCES

- [1] W. C. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, 1994.
- [2] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007.
- [3] Github. [Online]. Available: <https://github.com/>
- [4] Maven central repository. [Online]. Available: <https://mvnrepository.com/repos/central>
- [5] Npm. [Online]. Available: <https://www.npmjs.com/>
- [6] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Liu, and Y. Wu, "An empirical study of usages, updates and risks of third-party libraries in java projects," *The 36th IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*, 2020.
- [7] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 186–196.
- [8] M. Valiev, B. Vasilescu, and J. D. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 644–655.
- [9] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 181–191.
- [10] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 995–1010.
- [11] Maven repository: org.json:json. [Online]. Available: <https://mvnrepository.com/artifact/org.json/json>
- [12] Maven repository: com.google.code.gson:gson. [Online]. Available: <https://mvnrepository.com/artifact/com.google.code.gson/gson>
- [13] Maven repository: com.fasterxml.jackson.core:jackson-core. [Online]. Available: <https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core>
- [14] C. Teyton, J. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012, pp. 289–298.
- [15] —, "Automatic discovery of function mappings between similar libraries," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 192–201.
- [16] C. Teyton, J. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.
- [17] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 154–164.
- [18] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "On the use of information retrieval to automate the detection of third-party java library migration at the method level," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 347–357.
- [19] —, "Migrationminer: An automated detection tool of third-party java library migration at the method level," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 414–417.
- [20] H. Alrubaye, "How does API migration impact software quality and comprehension? an empirical study," *CoRR*, vol. abs/1907.07724, 2019.
- [21] H. Alrubaye, M. W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, and J. Mcgoff, "Learning to recommend third-party library migration opportunities at the API level," *Appl. Soft Comput.*, vol. 90, p. 106140, 2020.
- [22] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions - incorporating relational and categorical knowledge into word embedding," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 338–348.
- [23] F. L. de la Mora and S. Nadi, "An empirical study of metric-based comparisons of software libraries," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2018, Oulu, Finland, October 10, 2018*, 2018, pp. 22–31.
- [24] Awesome java: A curated list of awesome frameworks, libraries and software for the java programming language. [Online]. Available: <https://github.com/akullpp/awesome-java>
- [25] Alternativeto: Crowd-sourced software recommendations. [Online]. Available: <https://alternativeto.net/>
- [26] E. Larios-Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.
- [27] Y. Ma, C. Bogart, S. Amreen, R. Zaretski, and A. Mockus, "World of code: an infrastructure for mining the universe of open source VCS data," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, 2019*, pp. 143–154.
- [28] Libraries.io open data. [Online]. Available: <https://libraries.io/data>
- [29] N. Craswell, *Mean Reciprocal Rank*. Boston, MA: Springer US, 2009, pp. 1703–1703. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_488
- [30] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Trans. Inf. Syst.*, vol. 20, no. 4, pp. 422–446, 2002.
- [31] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J. Jézéquel, "Model-driven engineering for software migration in a large industrial context," in *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 482–497.
- [32] B. Verhaeghe, A. Etien, N. Anquetil, A. Seriai, L. Deruelle, S. Ducasse, and M. Derrass, "GUI migration using MDE from GWT to angular 6: An industrial case," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 579–583.
- [33] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 651–654.
- [34] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 457–468.
- [35] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (T)," in *30th IEEE/ACM International Conference on Automated Software*

Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, 2015, pp. 585–596.

- [36] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, “Mapping API elements for code migration with vector representations,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, 2016, pp. 756–758.
- [37] A. T. Nguyen, Z. Tu, and T. N. Nguyen, “Do contexts help in phrase-based, statistical source code migration?” in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, 2016, pp. 155–165.
- [38] B. Dorninger, M. Moser, and J. Pichler, “Multi-language re-documentation to support a COBOL to java migration project,” in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 536–540.
- [39] N. D. Q. Bui, Y. Yu, and L. Jiang, “SAR: learning cross-language API mappings with little knowledge,” in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 796–806.
- [40] B. Cossette and R. J. Walker, “Seeking the ground truth: a retroactive study on the evolution and migration of software libraries,” in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE’12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 55.
- [41] R. G. Kula, D. M. Germán, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.
- [42] M. Nita and D. Notkin, “Using twinning to adapt programs to alternative apis,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 205–214.
- [43] P. Kapur, B. Cossette, and R. J. Walker, “Refactoring references for library migration,” in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, 2010*, pp. 726–738.
- [44] A. Gokhale, V. Ganapathy, and Y. Padmanaban, “Inferring likely mappings between apis,” in *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 82–91.
- [45] A. Santhiar, O. Pandita, and A. Kanade, “Mining unit tests for discovery and migration of math apis,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 4:1–4:33, 2014.
- [46] A. C. Hora and M. T. Valente, “Apiwave: Keeping track of API popularity and migration,” in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, R. Koschke, J. Krinke, and M. P. Robillard, Eds. IEEE Computer Society, 2015, pp. 321–323.
- [47] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, “Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding,” *IEEE Transactions on Software Engineering*, 2019.
- [48] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm, “Study of an API migration for two XML apis,” in *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, 2009, pp. 42–61.
- [49] T. T. Bartolomei, K. Czarnecki, and R. Lämmel, “Swing to SWT and back: Patterns for API migration by wrapping,” in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–10.
- [50] Apache maven project. [Online]. Available: <http://maven.apache.org/>
- [51] MongoDB: The database for modern applications. [Online]. Available: <https://www.mongodb.com/>
- [52] Spring home. [Online]. Available: <https://spring.io/>
- [53] Apache commons bccl™ - byte code engineering library. [Online]. Available: <https://commons.apache.org/proper/commons-bccl/>
- [54] F. Thung, D. Lo, and J. L. Lawall, “Automated library recommendation,” in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 182–191.
- [55] X. Ye, R. Bunescu, and C. Liu, “Learning to rank relevant files for bug reports using domain knowledge,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 689–699.
- [56] H. Niu, I. Keivanloo, and Y. Zou, “Api usage pattern recommendation for software development,” *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.
- [57] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. Germán, and K. Inoue, “Search-based software library recommendation using multi-objective optimization,” *Information & Software Technology*, vol. 83, pp. 55–75, 2017.
- [58] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, “Improving reusability of software libraries through usage pattern mining,” *Journal of Systems and Software*, vol. 145, pp. 164–179, 2018.
- [59] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, “Crossrec: Supporting software developers by recommending third-party libraries,” *Journal of Systems and Software*, vol. 161, p. 110460, 2020.
- [60] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, “Diversified third-party library prediction for mobile app development,” *IEEE Transactions on Software Engineering*, 2020.
- [61] W. Zheng, Q. Zhang, and M. R. Lyu, “Cross-library API recommendation using web search engines,” in *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 480–483.
- [62] S. Xu, Z. Dong, and N. Meng, “Meditor: inference and application of api migration edits,” in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 335–346.
- [63] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, and M. O’Boyle, “M3: Semantic api migrations,” *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*, 2020.
- [64] A. Decan and T. Mens, “What do package dependencies tell us about semantic versioning?” *IEEE Transactions on Software Engineering*, 2019.
- [65] J. Dietrich, D. J. Pearce, J. Stringer, A. Tahir, and K. Blincoe, “Dependency versioning in the wild,” in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, 2019*, pp. 349–359.
- [66] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, “The emergence of software diversity in maven central,” in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, 2019*, pp. 333–343.
- [67] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S. Cheung, “Do the dependency conflicts in my project matter?” in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 319–330.
- [68] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, “Watchman: Monitoring dependency conflicts for python library ecosystem,” *The 42nd International Conference on Software Engineering (ICSE 2020)*, 2020.
- [69] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, “Interactive, effort-aware library version harmonization,” *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.