The Sampling Threat when Mining Generalizable Inter-Library Usage Patterns

Yunior Pacheco Correa[a], Coen De Roover[a], Johannes Härtel[b]

[a] *Vrije Universiteit Brussel, Brussels, Belgium*
[b] *Vrije Universiteit Amsterdam, Amsterdam, Netherlands*

Abstract

Tool support in software engineering often relies on relationships, regularities, patterns, or rules mined from other users' code. Examples include approaches to bug prediction, code recommendation, and code autocompletion. Mining is typically performed on samples of code rather than the entirety of available software projects. While sampling is crucial for scaling data analysis, it can affect the generalization of the mined patterns.

This paper focuses on sampling software projects filtered for specific libraries and frameworks, and on mining patterns that connect different libraries. We call these inter-library patterns. We observe that limiting the sample to a specific library may hinder the generalization of inter-library patterns, posing a threat to their use or interpretation. Using a simulation and a real case study, we show this threat for different sampling methods. Our simulation shows that only when sampling for the disjunction of both libraries involved in the implication of a pattern, the implication generalizes well. Additionally, we show that real empirical data sampled using the GitHub search API does not behave as expected from our simulation. This identifies a potential threat relevant for many studies that use the GitHub search API for studying inter-library patterns.

Keywords: Sampling, Usage Patterns, Inter-Library, Dataset, Data Mining

Email addresses: ypacheco@vub.be (Yunior Pacheco Correa), coen.de.roover@vub.be (Coen De Roover), j.a.hartel@vu.nl (Johannes Härtel)

URL: https://orcid.org/0000-0002-7849-7841 (Yunior Pacheco Correa), https://orcid.org/0000-0002-1710-1268 (Coen De Roover), https://orcid.org/0000-0002-7461-2320 (Johannes Härtel)

1. Introduction

Tool support for software engineering often depends on regularities, patterns, or rules mined from code. If an analysis of all code does not scale, it can also be conducted on a small sample of it. Since a sample can represent an entire population, the mining process can also be performed on samples of code (e.g., top-n most starred projects on GitHub) rather than the entire population of software projects (e.g., all projects on GitHub). The specifics of how the sample is collected, however, might influence the generalization.

Sampling is crucial in empirical research [1], including Empirical Software Engineering (ESE) [2] and Mining Software Repositories (MSR) [3, 4, 5]. In MSR and ESE, researchers often sample software projects from sources like GitHub [6] and aim to generalize their findings to unseen software projects [7].

1.1. Motivation

For studies that mine library or framework (API) patterns, sampling is equally important. Researchers extract API usage patterns from code by sampling examples from existing API applications. The most basic sampling method involves searching and filtering for client applications that use a specific API, such as a subset found via GitHub's search, querying for the specific library or framework. For instance, Nuryyev et al. [8] selected 533 repositories from GitHub that use the MicroProfile framework.

Since sampling is often unavoidable, recent MSR and ESE research has started to investigate the consequences of different sampling forms [9, 5]. In this context, related work tries to optimize the accuracy of mined patterns while potentially threatening the generalizability. When sampling client software projects, we must ensure two key properties of the mined patterns:

- **Accuracy:** The patterns should be precise and accurately reflect the client projects in the sample.
- **Generalizability:** The patterns should generalize well to the entire population of client projects, potentially using the same libraries or frameworks or not.

Nuryyev et al., for instance, mined the sample of 533 repositories for annotation usage rules and validated them by human experts. We can find a rule of the following form:

$$type(\text{javax.json.JsonString}) \\ \rightarrow annotation(\text{org.eclipse.microprofile.jwt.Claim})$$

This rule can be interpreted as a logical or probabilistic relationship, indicating that when a method returns a JsonString, it should carry a Claim annotation. This statement crosses different libraries, making it an inter-library pattern. Since JavaX's JsonString may also appear in other contexts, unrelated to MicroProfile's Claim annotation, the rule is not logically true. However, it may hold probabilistically with a certain confidence. This concept of confidence originates from association rule mining and is described in Sec. 2. Such patterns are relevant, indicating that JsonString *facilitates the usage* of a Claim annotation. Another example might be Pandas DataFrame facilitating the usage of the Matplotlib library.

Nuryyev et al. discovered this pattern with unexpectedly high confidence. Human experts, however, identified it as incorrect and attributed the issue to a limitation in their rule mining approach. We show that this problem actually arises from the sampling method they used, rather than from the rule mining algorithm itself.

1.2. Problem Statement

Reflecting deeper on the origin of this problem, the authors assume JavaX to be an integral part of the MicroProfile framework. This implies that whenever we sample for MicroProfile, we also sample for JavaX. This is not true and renders it a rule between different libraries. We call it an *inter-library pattern* instead of an *intra-library pattern*. From a sampling perspective, we noticed that this distinction can be crucial. We show that the sensitive context of multiple libraries introduces the risk of incorrect rules that do not generalize.

This problem leads us to ask: Is the confidence of a rule computed on the sample the same as for the entire population? In short, can we generalize? Is there a difference between intra-library and inter-library patterns? More concretely, we define our **research question** as follows:
How do sampling methods influence the generalizability of mined inter-library usage patterns from client software projects?

1.3. Research Method and Results

To answer the research question, this paper follows a *research method* that combines an *empirical study* and a *simulation study* that examines inter-library patterns mined on data using different practices: i) *random samples*, ii) *single library samples* that collect client projects that use a particular library, and iii) *co-used library samples* that collect client projects by analyzing different combinations of usage of two libraries.

In the simulation study, we analyze the *generalizability* in terms of *confidence* for both intra-library and inter-library patterns. In the empirical study, we focus on inter-library patterns. In general, we examine the patterns obtained by the *same rule mining algorithm* run on data collected using *different sampling methods* and for *different degrees of library popularity*. We present evidence that *confidence* of mined inter-library usage patterns differs depending on the sampling method. Thereby, patterns may or may not generalize. We provide a replication package online[1].

1.4. Contributions and Actionable Insights

This paper makes the following contribution: An empirical and simulation study examining the impact of different sampling methods on the mined inter-library patterns, considering the degree of popularity of the libraries involved.

The resulting insights of our study can directly be used by future studies that extract patterns from samples. Either they **improve their sampling method in a way that the mined patterns generalize better**, or they **improve their threats to validity section by discussing the limitations that we identified**.

1.5. Roadmap

Sec. 2 gives the background on representing source code, recovering patterns, computing confidence, and the examined sampling methods. Sec. 3 presents the research method that we use to answer the research question. Sec. 4 presents the simulation study. Sec. 5 presents the empirical study. Sec. 6 discusses limitations and threats. Sec. 7 covers the related work. Sec. 8 discusses the implications of our findings. Sec. 9 concludes the paper.

[1] `https://doi.org/10.5281/zenodo.14841462`

2. Background on Pattern Mining and Sampling

This section starts with a discussion of established background needed for the paper. We introduce the relevant concepts to make the paper as self-contained as possible. We will provide the relevant pointers for further reading. In particular, we will discuss rule mining methods and establish *single library* sampling. We finally present our idea of new *co-used libraries* sampling methods. The related work section will discuss which method combinations of sampling and rule mining already appear in the literature.

2.1. Pattern Mining Method

We start our discourse with a straightforward but common method to mine usage patterns based on a logic representation of the abstract syntax tree (AST). We will use this same mining method in all our experiments, being a constant in our research design. Less straightforward alternatives are meaningful to study, but we consider them out of scope for now.

2.1.1. Representational Mapping

Approaches to pattern mining often rely on a *representational mapping*. The representational mapping converts the abstract syntax tree (AST) of the client projects, those that are contained in a sample, into a representation that can be mined.

Logic Facts. We use a fact-based representation of the AST that is common to approaches doing logic-based reasoning [10, 11, 12]. The example fact type(javax.json.JsonString) from such a representation expresses the usage of a type that is part of a JSON library. We can use such facts to create an abstract representation of the clients that use a library or framework. We choose this representation because it is a very transparent and formal way of reasoning about regularities in source code.

Structure. The usage of libraries is often reflected in the program structure. In the following experiments, we convert methods and fields we find in the code into sets of logic facts. These sets of logic facts are referred to as *transactions* in Data Mining literature [13]. Throughout the remainder of the paper, we refer to them as **observations**, as it is a terminology that better suits the context of usage pattern mining. In each analyzed client project, we collect various of these observations. This is comparable to studies like [8].

2.1.2. Confidence and Support

We use the sets of logic facts to recover new patterns or verify whether specific patterns fit the data or not. A pattern, usable in a tool for recommendation and autocompletion, can be written $A \rightarrow B$. We call it a *rule* due to the specific shape. Logic facts in A will need to be present in the code, so that B can be recommended. We call the left side the *antecedent* and the right side the *consequent*.

Rules are a form of pattern or regularity. Whereas patterns can only be associated with a support when mined from data, rules are typically associated with some sort of confidence or likelihood because they have the shape of an implication (i.e., *antecedent* implies *consequent*)

There are mining algorithms, like association rule mining, that can recover patterns in the form of rules from data [14, 15]. However, one can also compute confidence and support of existing rules on a dataset. In the experiments, we will use basic association rule mining, but in several cases, it will be sufficient to only compute the confidence and support of rules on a sample. We will repeat the definition of confidence and support next (but also refer to [14, 15]).

Support. The support of a logic fact is defined by counting its occurrences in the representation of the client projects. The support of a rule $A \rightarrow B$ is defined as the support of $\{A, B\}$. The support reflects the popularity of a certain usage.

Confidence. The confidence of a rule $A \rightarrow B$ is defined as the support of $\{A, B\}$ divided by the support of $\{A\}$. Such a definition is close to the probability of facing B when A is present.

Giving the example of having $A = \{$type(JsonString), return$\}$, $B = \{$annotation(Claim)$\}$ and the confidence of rule $A \rightarrow B$ is 50%, then it means that 50% of the observations that contain "return" and "type(JsonString)" also contain "annotation(Claim)".

A recommender system using this rule can operate as follows: If a method returns "JsonString", it can suggest including the annotation "Claim" with a 50% confidence. Whether the recommender system makes this suggestion depends on a threshold. Our primary interest is in providing the correct confidence to the recommender system.

2.2. Examined Sampling Methods

The methods we discuss here are all referred to as 'sampling' methods because empirical research uses them to collect data. However, we acknowledge that they can also be called data collection or filtering methods. For consistency, throughout the paper, we use the term 'sampling', even for more black-box notions of accessing data via the GitHub search API.

We now discuss established ways of sampling, which are the central subject of our paper. A general introduction to sampling can be found in [1].

2.2.1. Random Sample

Findings of empirical studies target a *population*. For us, the population consists of all software projects on GitHub, denoted as $\mathbb{P}$. Due to scalability limitations, studies often rely on *random samples*, which are random subsets of the original population. Random samples have desirable properties; they can be small, yet insights may still generalize from the sample to the entire population. Such insights often need to be associated with estimates of uncertainty. Software engineering studies frequently use random sampling from dumps, as seen in [6, 4]. In the remainder of this paper, we will refer to a random sample as $\mathbb{R}$.

2.2.2. Single Library Samples

Random sampling faces practical limitations when studies aim to examine specific and rare aspects of a population. This is due to the lack of sufficient data for studying the target aspect. When studying rare libraries and frameworks, we encounter this issue, as discussed in [16] regarding the popularity of libraries.

Therefore, several studies filter samples from GitHub for the usage of a particular (target) library (e.g., [8, 17]). Assuming our study's population corresponds to all projects on GitHub, we consider that a project *uses* a library if the source code includes a reference to the library. Specifically, we search for client projects with Java imports to the target library using the standard filter of the GitHub search API. The exact method of access is detailed in the online material. We call this a *single library sample*. We will later refer to single library samples as $\mathbb{H}$ and $\mathbb{L}$, depending on the popularity of the library.

2.2.3. Restrictive Single Library Samples

A more restrictive sampling strategy considers only observations directly related to the target library within the selected projects. Since observations can be more fine-grained than projects, this strategy may exclude any class, method, or field in a project that is not directly related to the target library. Filtering out observations unrelated to the target library may initially seem to improve scalability.

This approach is used when mining API usage patterns from client code. For example, the authors of [8] only use observations with a type from the MicroProfile framework as input to the mining algorithm. Samples generated by this method are identified as $\mathbb{H}'$ and $\mathbb{L}'$.

2.2.4. Co-used library samples

Some patterns might span across multiple libraries, which can also be relevant in the context of a target library. We consider co-used library samples that include such relevant libraries. Our study specifically examines pairs of libraries and investigates the results when mining inter-library patterns involving them. If necessary, we differentiate between the two libraries in a pair by their popularity. In this context, we refer to popularity as the degree to which a library is used in the population of projects on GitHub. The next section will clarify this notion. We use the convention that the samples of the more popular library (*high*) are identified as $\mathbb{H}$ and of the less popular library (*low*) as $\mathbb{L}$. We now define sampling methods that target both, *high* and *low* libraries. Accordingly, we have new sampling methods:

- $\mathbb{H} \vee \mathbb{L}$: This sample includes projects that use either the more popular (*high*) library **or** the less popular (*low*) one.
- $\mathbb{H} \wedge \mathbb{L}$: This sample includes projects that use **both** the more and less popular library.

Co-used library samples are expected to be more representative of possible interrelations between the involved libraries. Therefore, they are subject to our study.

3. Research Method

We first outline the design of our research method to address the impact of sampling methods on mined inter-library patterns. We then explain our

adoption of a hybrid research method, combining empirical and simulation studies. Limitations and threats to our study are discussed in Sec. 6. Figure 1 depicts a sketch of the design of this study. Specific aspects in green and blue correspond to the simulation and empirical study, respectively.

3.1. Scenarios on Asymmetric Library Popularity

Our design aims to examine the impact of the sampling method on the resulting patterns and the possible influence of the popularity of the libraries involved in the patterns. To analyze the effect of library popularity on the generalizability of inter-library patterns, we define two scenarios for the degree of popularity for *high* and *low*.

- *Imbalanced scenario*: While *high* is very popular, *low* is rare. We use $p_{high} = 37\%$ and $p_{low} = 0.12\%$ to represent this large difference in popularity.
- *Balanced scenario*: Both, *high* and *low* have more balanced popularity. We use $p_{high} = 14\%$ and $p_{low} = 2\%$.

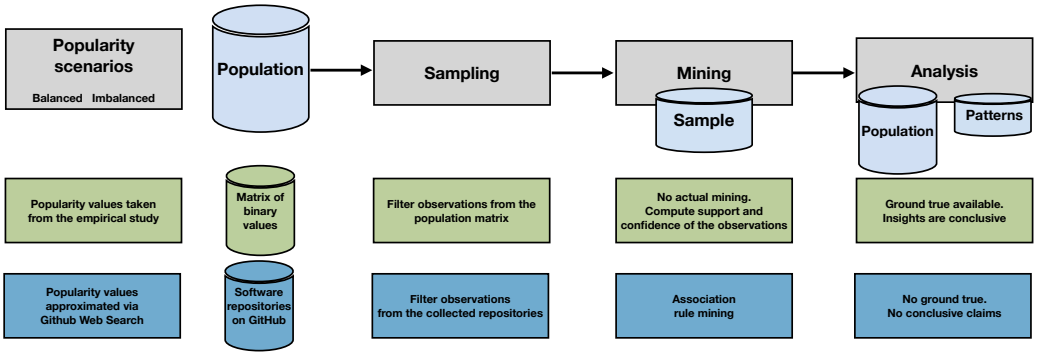


Figure 1: **A sketch of our research methodology: Simulation and empirical studies are shown in green and blue, respectively.**

The values of p are set based on two scenarios of co-usage presented in our empirical study. In particular, the empirical study examined pairs JavaX and MicroProfile as the *imbalanced scenario* and JUnit and Mockito as the *balanced scenario*. The JavaX and MicroProfile have also been studied in [8].

The second pair of libraries are two popular libraries that are often used together in unit testing. We believe that, as applications use both frequently, this pair is representative of a more balanced co-usage relationship.

We use the GitHub web search tool to estimate the popularity of these libraries. First, the entire population of projects is obtained by querying for Java files that contain an import statement with the package "java.", resulting in $71.8M$ code files. Likewise, for each library, we query for Java files that contain an import statement with the base package name of the library. An example of the query for the JavaX library is "import javax" language:Java, resulting in $26.9M$ code files and a value of $p = 26.9M/71.8M \approx 37\%$.

In the simulation, we use the same values for p. This enhances the realism of the simulation and provides a better reference point for discussion in the empirical study.

3.2. Hybrid Study

We adopt a research method that combines an empirical study with a simulation study. For the use of simulations, we refer to [18] and [19, 20]. Also see our recent work on simulations between SE and active learning [21].

To ensure we isolate the effect of the sampling method, we keep all other parts of the study constant. We do not evaluate entirely different combinations of methods, such as alternative rule mining algorithms. This approach aligns with a standard controlled experiment design, where most variables are held constant while some are varied by us. A standard reference is [22]. The constants in our research design are:

- We target the same population: synthetic data in the simulation and Java projects on GitHub for the empirical study.
- We use the same rule mining method throughout the study.

The size and properties of the resulting samples vary across different sampling methods. Consequently, the properties of the mined patterns also vary. We align and compare the patterns computed by the different sampling methods to answer our research question. Table 1 summarizes the different types of patterns examined in our hybrid setup.

3.2.1. Simulation Study

Simulations are essential in our context, as not all insights can be derived from empirical data alone. Simulations provide transparent, controllable, and repeatable environments to study our problem. We can control the simulated population and make it fully transparent for analysis. This allows us to identify which sampling methods produce correct confidence values

Table 1: Type of patterns examined in our research design

Scenario	Simulation Type of pattern examined	Empirical
Balanced $p_{high} = 14\%$ and $p_{low} = 2\%$	Intra-library / Inter-library	Inter-library
Imbalanced $p_{high} = 37\%$ and $p_{low} = 0.12\%$	Inter-library	Inter-library

that reflect the entire population. We examine both intra-library and inter-library patterns, demonstrating that generalizability issues arise only with inter-library relations. These simulation results serve as a reference point for our empirical study. Simulation insights, however, are conditional on the assumptions made in our simulations.

3.2.2. Empirical Study

The empirical study then turns to real data using GitHub. Here, we observe a contradiction between real sampling and simulated sampling. The simulation suggests that confidence values should be similar, while the empirical data shows differences between alternatives that we considered both good approximations of the correct confidence.

Unlike the simulation, the empirical setting makes definitive statements about the entire population of Java projects difficult. However, the difference between the simulation and empirical results indicates a potential threat to the reliability of sampling from GitHub if our simulation assumptions are correct.

3.3. Insights

The insights answering our research question are listed below and are repeated in the remainder of the paper when we show them. They have been tagged, denoting whether they come from the empirical or from the simulation study. We only use **correct** and **incorrect** in the context of the simulation because there we can definitely state a relation to the entire population. In the empirical study, we only describe **differences**.

- **Insight 1 (simulation)**: For implications of shape $A \rightarrow B$, sampling for the library in the antecedent (A) will result in **correct** confidences; sampling for the library in the consequent (B) will result in **wrong** confidences.

- **Insight 1 (empirical)**: For implications of shape $A \rightarrow B$, and using the GitHub Search API, sampling for the library in the antecedent (A) **can** result in **different** confidence values than sampling for the library in the consequent (B). This conforms to the simulation insight.
- **Insight 2 (simulation)**: For implications of shape $A \rightarrow B$ and vice versa ($B \rightarrow A$), sampling for the disjunction $\mathcal{A} \vee \mathcal{B}$ will result in **correct** confidence values for both types of rules.
- **Insight 2 (empirical)**: For implications of shape $A \rightarrow B$ and vice versa ($B \rightarrow A$), and using the GitHub Search API, sampling for the corresponding library in the antecedent will result in **different** confidence values than sampling for the disjunction $\mathcal{A} \vee \mathcal{B}$. This does not conform to the simulation insight.

4. Simulation Study

The first part of our evaluation of the sampling methods involves a simulation study. A simulation is almost unavoidable due to a conceptual problem: It is nearly impossible to obtain data on the entire population of GitHub projects. All insights that describe the generalization of patterns from any sample to the entire population are therefore demonstrated through simulation.

The simulation is divided into two parts. The first part focuses on intra-library patterns, where we mine for patterns within a single library. The core insight is that intra-library patterns are accurate and generalizable regardless of the sampling method used. The second part addresses inter-library patterns that cross library boundaries. This highlights issues with generalizability. We use the simulation to demonstrate that the sampling method used to collect the data is crucial for ensuring that inter-library patterns mined from the sample can generalize to the entire population. We first describe the common setup and then specialize patterns in each part.

4.1. Common Assumptions

The following are the assumptions and closely related simplifications that we are willing to make about library usage patterns in GitHub projects. The evidence that comes from the simulation is conditional on these assumptions. We consider them plausible, but we cannot prove this.

4.1.1. Co-usage

The simulation captures our scenario where two libraries are used together in client projects. We refer to these libraries as *high* and *low*, based on their popularity as mentioned in Sec. 2.2.4. We simplify by not structuring usages (into client projects, classes, or fields in the Java files). Instead, we simulate observations flat, only with the presence of an *import statement* or a *method call*. An observed usage is represented in terms of four logic facts $\mathcal{C} = \{\mathsf{importHigh()}, \mathsf{callHigh()}, \mathsf{importLow()}, \mathsf{callLow()}\}$. These logic facts are binary.

We can then simulate an observation with a vector of four binary values. Using $\mathcal{C}$ as the index of the vectors, the observation vector $v = (1, 1, 1, 0)$ indicates that both *high* and *low* are imported, and the method from *high* is called.

4.1.2. Population

The following are the population details:

- We use a flat population represented as a matrix, where each row corresponds to a vector of binary values.
- The population size is a parameter n. In the simulations presented, we use a population size n of $1,000,000$ observed usages.
- Logic facts in the matrix are populated randomly with probabilities p_{high} and p_{low}, which correspond to the index positions in $\mathcal{C}$. These parameters reflect the popularity of *high* and *low* libraries.
- We assume a set of logic rules built on top of $\mathcal{C}$ in the form of implications $A \rightarrow B$.

4.1.3. Rules

We simulate different types of rules enforced in the population matrix.

Logic. We simulate rules in the form $A \rightarrow B$ that are always correct. Whenever we see A, we also see B. This is enforced in the simulation by setting the value of the facts in the consequent of the rule to 1 whenever the facts in the antecedent are 1.

For example, if we want to enforce the rule $\mathsf{callHigh()} \rightarrow \mathsf{callLow()}$, we set the value of callLow() to 1 whenever callHigh() is 1 for all vectors in the population matrix. In this case, the observation represented by vector $v = (1, 1, 1, 0)$ would be transformed to $v' = (1, 1, 1, 1)$.

Facilitate. We are certain that logic rules hold in all samples. This makes the opposite direction interesting because there we have a **confidence** (the confidence if seeing B also seeing A). We refer to this as **facilitate-rules** $\mathcal{F}$ and they are problematic for sampling.

Import. In each simulation, apart from the logic rule above, we enforce two other logic rules that reflect the real world restriction that if we want to use a library, we must import it: callHigh() $\rightarrow$ importHigh() and callLow() $\rightarrow$ importLow().

4.1.4. Sampling Methods

We examine the sampling methods introduced in Sec. 2.2 in the context of the simulation. We produce the samples by filtering the observations from the population for a specific set of facts, depending on the sampling method. To not influence our insights, we use the same sample size for all methods: 1% of n. Details of sampling are the following:

- The method to produce a random sample ($\mathbb{R}$) does not filter for any facts and randomly selects the observations.
- The method to produce the sample targeting the *high* library ($\mathbb{H}$) filters for the fact importHigh(). This aligns with our notion that an application *uses* a library if it includes 'some' reference, and thereby an import to the library. The same applies for the sample $\mathbb{L}$ targeting the *low* library, which filters for the fact importLow().
- To simulate a more restrictive filter, we use the call facts for filtering. Thus, the method to produce the sample $\mathbb{H}'$ filters for the fact callHigh() and not by the import. The same applies for $\mathbb{L}'$.
- The method for sample $\mathbb{H} \vee \mathbb{L}$ filters observations containing at least one of the facts importHigh() or importLow(). For sample $\mathbb{H} \wedge \mathbb{L}$, the filter requires that both importHigh() and importLow() must be present in the observation.

As an example, the vector $v = (1, 1, 1, 0)$ could have been filtered to be included in any of the samples explained above, except for the method to produce sample $\mathbb{L}'$. The value for the fact callLow() is 0 and therefore the observation would not be included in the sample $\mathbb{L}'$.

4.1.5. General Simulation Setup

In summary, the simulation generates a matrix with n rows (number of observations) and 4 columns (size of $\mathcal{C}$) using a binomial distribution with probabilities p_{high} and p_{low}, which correspond to the columns. As mentioned previously, these parameters are adjusted to reflect the popularity of the libraries analyzed in the empirical study.

The process of population generation, sampling, and computing the confidence for the facilitate-rules for each sample is repeated 100 times. The results are then aggregated or shown as distributions. This repetition mitigates the risk of deriving insights from an anomalous distribution of the population data. Algorithm 1 outlines the general simulation process. Further implementation details can be found online.

Algorithm 1 Simulation

Require: n, p_{high}, p_{low}, $runs$ **return** $Confs$

$methods \Leftarrow (\mathbb{P},\mathbb{R},\mathbb{H},\mathbb{L},\mathbb{H}',\mathbb{L}',\mathbb{H} \vee \mathbb{L},\mathbb{H} \wedge \mathbb{L})$
$Confs \Leftarrow \emptyset$
for $iter = 1, 2, \ldots, runs$ **do**
    $population \Leftarrow initialize(n, p_{high}, p_{low})$
    $population \Leftarrow enforceLogicRule(population)$
    $population \Leftarrow enforceImports(population)$
    for each $m \in methods$ **do**
        $sample \Leftarrow filter(population, m)$
        $conf \Leftarrow confidence(sample)$ *//Compute the confidence of the facilitate-rule.*
        $Confs \Leftarrow add(iter, m, conf)$
    end for
end for
return $Confs$

4.2. Intra-library Usage Patterns

We start with the intra-library patterns. **Can we generalize the confidence of rules that involve a single library to the entire population under a given sampling method?**

We use the same simulation setup as for the inter-library patterns, but with some minor changes.

4.2.1. Specialized Rules

To stress single library usage, we change $\mathcal{C}$ a little to $\mathcal{C}_1 = \{$importHigh(), callHigh1(), callHigh2(), importLow(), callLow()$\}$. Note that we introduce two new facts callHigh1() and callHigh2() to represent a possible usage pattern inside the single library *high*. We ignore calls to *low* for brevity but we keep imports to it. Sampling for another library (*low*) might still be relevant for the generalization.

To simulate the intra-library pattern we use the logic facts callHigh1() and callHigh2(). The idea is to represent the typical usage pattern where a call to a method from a library depends on a previous method call to the same library.

We enforce the logic rule callHigh2() $\rightarrow$ callHigh1() in the same way as described before. Similarly, we are interested in the facilitate-rule which is the opposite direction. We want to know the confidence that if we call method callHigh1(), we also need to call method callHigh2() referring to this as $high1 \rightarrow high2$. For parameters p_{high} and p_{low}, we use the values from the balanced scenario ($p_{high} = 14\%$ and $p_{low} = 2\%$).

4.2.2. Simulation Results

We present the confidence of the facilitate-rule in the opposite direction of the encoded logic rule. This is $high1 \rightarrow high2$. Figure 2 shows the difference in the confidence values computed in the different samples compared to the confidence in the entire population ($\mathbb{P}$). We expect that the confidence values computed in the other samples approximate the confidence values in the entire population. **The population $\mathbb{P}$ is the ground truth. Ideally, comparable confidence values should be computed from the other samples.**

The figure shows that, on average, the confidence values computed from all other samples correctly approximate the confidence computed from the entire population. There is no systematic error by any method, even for samples like $\mathbb{L}$ or $\mathbb{L}'$. We only observe some uncertainty. The impact on generalization is negligible.

These results support the idea that when mining for intra-library patterns, the sampling method is not relevant for the generalization of the patterns. Using any of the sampling methods presented here, **a recommendation system can suggest a pattern with the same certainty as if that pattern were mined on the entire population**. This behavior is what we would like to have in inter-library patterns, but as shown in the

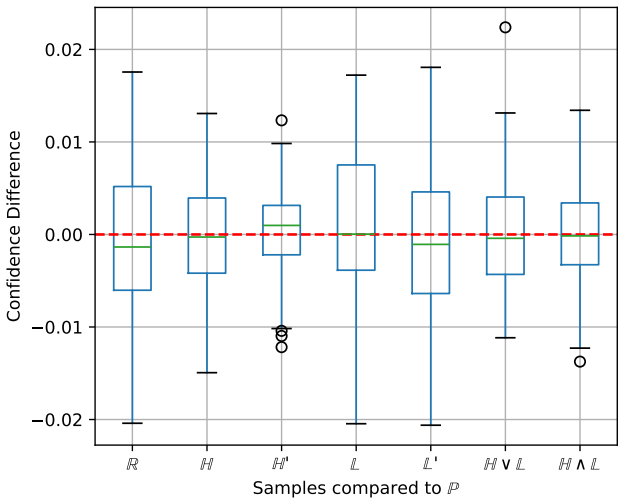


Figure 2: Difference in confidence between samples and $\mathbb{P}$ for rule $high1 \rightarrow high2$. The red line denotes the baseline $\mathbb{P}$.

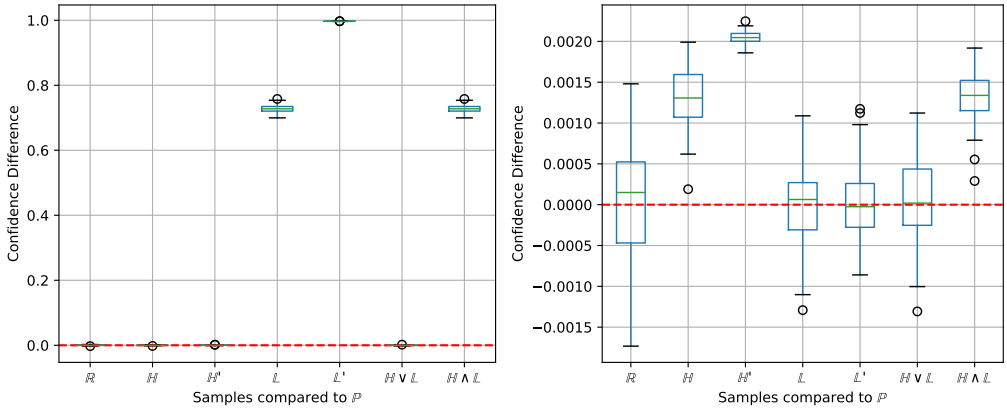
next section, this is not the case.

4.3. Inter-library Usage Patterns: Simulation Results

In this part, we show how we derive the two simulations insights. We present the results for the two popularity scenarios introduced in Sec. 3.1, with facilitate-rules in both directions, referred to as $high \rightarrow low$ and $low \rightarrow high$. This results in four plots shown in Figure 3.

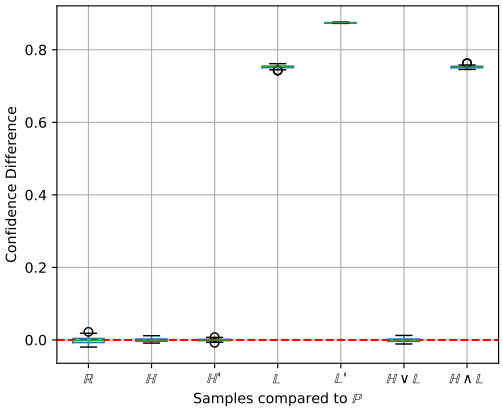
Random Sample. The random sample $\mathbb{R}$ suffices to correctly identify confidence values. The confidence values are almost identical to the entire population. However, for rule $low \rightarrow high$ they are less accurate. This is typical for random samples and can be compensated by statements on uncertainty (e.g., confidence intervals or p-values). If we compare the difference in confidence between $\mathbb{R}$ and the entire population in Figure 3b and Figure 3d, we observe that in the balanced scenario, the variation in accuracy is lower than in the imbalanced scenario. This may indicate that $\mathbb{R}$ is more accurate for popular libraries.

Regardless of uncertainty, in both scenarios, one can approximate the confidence values of the entire population correctly. This enables generalizing from the random sample to the entire population. However, random samples become impractical in our empirical study, not always providing sufficient observations for both libraries.

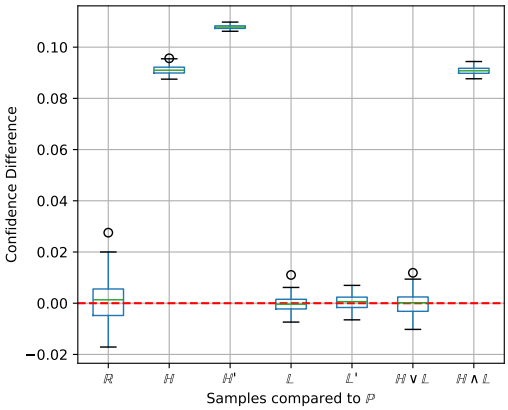


(a) rule *high* → *low* imbalanced scenario.

(b) rule *low* → *high* imbalanced scenario.



(c) rule *high* → *low* balanced scenario.



(d) rule *low* → *high* balanced scenario.

Figure 3: Difference in confidence between samples and $\mathbb{P}$ in the two scenarios. A red dotted line denotes the baseline $\mathbb{P}$.

Single Library Samples. For confidence values computed from *single library* samples, we get different results for the facilitates rules.

In general, for a rule $A \rightarrow B$, the confidence values computed on *single library* samples targeting the library of the antecedent (A) correctly approximate the confidence computed on the entire population. However, the confidence values computed on *single library* samples targeting the library of the consequent (B) overestimate the real confidence values (if the rule is not logic).

This is depicted in Figure 3. For rule $low \rightarrow high$: samples $\mathbb{L}$ and $\mathbb{L}'$ closely approximate $\mathbb{P}$, while samples $\mathbb{H}$ and $\mathbb{H}'$ largely overestimate the confidence. Analogously, for rule $high \rightarrow low$: samples $\mathbb{H}$ and $\mathbb{H}'$ closely approximate $\mathbb{P}$, and samples $\mathbb{L}$ and $\mathbb{L}'$ do not. This behavior is consistent in both scenarios, independently of the degree of popularity of the libraries. However, in the balanced scenario, the overestimation is less dramatic than in the imbalanced scenario.

Summarizing these results independent of popularity; **Insight 1 (simulation)**: For implications of shape $A \rightarrow B$, sampling for the library in the antecedent (A) will result in **correct** confidences; sampling for the library in the consequent (B) will result in **wrong** confidences.

Co-used library samples. Sample $\mathbb{H} \vee \mathbb{L}$ approximates the confidence values computed on $\mathbb{P}$ correctly.

For the two types of rules in Figure 3, we observe that on sample $\mathbb{H} \vee \mathbb{L}$ the confidence values don't differ too much from the confidence values computed on $\mathbb{P}$.

For co-used libraries samples, we cannot identify a consistent difference in the accuracy between imbalanced scenario and balanced scenario. For rule $high \rightarrow low$, the confidence values appear to be more accurate in an imbalanced scenario (see Figure 3a). For rule $low \rightarrow high$, the difference in confidence between sample $\mathbb{H} \vee \mathbb{L}$ and $\mathbb{P}$ is lower in the balanced scenario (see Figure 3d). This does not influence the generalizability of the confidence computed on these samples.

Summarizing these results independent of popularity; **Insight 2 (simulation)**: For implications of shape $A \rightarrow B$ and vice versa ($B \rightarrow A$), sampling for the disjunction $\mathcal{A} \vee \mathcal{B}$ will result in **correct** confidence values for both types of rules.

In the case of the sample $\mathbb{H} \wedge \mathbb{L}$, we did not observe confidence values that correctly generalize to the population.

4.4. Sample Size

As mentioned earlier, we used a sample size of 1% of the population. A valid question is whether this sample size is large enough, or how does sample size influence the results. The intuitive answer is based on the well-established idea that more is better and that the larger the sample, the more accurate the results. In our case, this is not correct, since the method used to collect the sample is the primary factor regardless of the sample size.

We conducted a small experiment to analyze the impact of the sample size on the results obtained for the inter-library patterns. For this experiment, we use the case where the facilitate-rule is $high \rightarrow low$ in the balanced scenario (Figure 3c). To examine sample size, we change it to 1%, 10%, 20% and 100% of the population. Figure 4 shows the results of these four simulations in terms of the average and the standard deviation of the confidence difference between the samples and $\mathbb{P}$. The standard deviations are scaled by the same constant factor for better visualization.

The results indicate that the sample size does not have a significant impact on the results. **The standard deviation decreases as the sample size increases**. Larger samples provide a better representation of the population, as random noise diminishes. However, **the average difference in confidence remains consistent regardless of sample size**. The mean of the differences we show depends on the filtering strategy, used by the sampling method, and not on the size of the sample.

5. Empirical Study

We now turn to real data. The objective is to understand, in practice, the generalizability of patterns mined by the same mining approach, using input from different sampling methods. We expect to observe results similar to those of the simulation.

5.1. Baseline Related Work

For this study, we keep close to the procedure presented in [8] while putting an emphasis on the sampling methods. We point out that authors run into a filtering that might be problematic using the following direct citation:

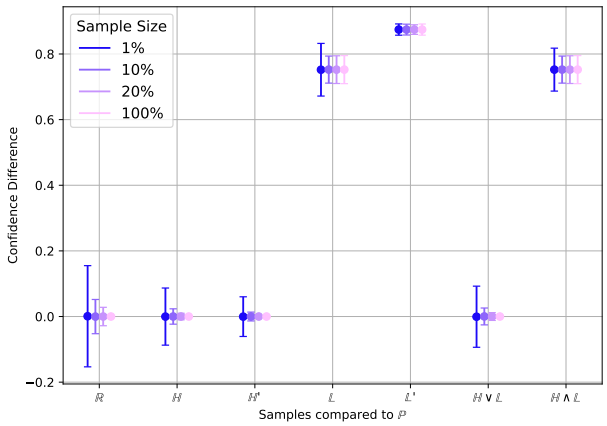


Figure 4: Difference in confidence between samples and $\mathbb{P}$ for different sample sizes, for rule *high* $\rightarrow$ *low* in the balanced scenario.

'*Remove frequent itemsets with no target API usage: Developers tend to use different libraries and frameworks in their code to accomplish different tasks. Our technique focuses on mining API usages of MicroProfile. Therefore, we are not interested in API usages from other libraries or frameworks. Thus, we remove all frequent itemsets that do not have at least one element of MicroProfile API.*' (direct citation of [8], page 5).

To extend our analysis, we not only studied the same combination of libraries as Nuryyev et al., but also included a second scenario with a more balanced popularity of libraries.

- Imbalanced scenario (JavaX and MicroProfile): We study the same target library as Nuryyev et al. and report on similar results from association rule mining as the original work. We include alternatives of sampling by considering the secondary JavaX library in co-used library samples. In this scenario, we face a significant difference in their respective popularity.

- Balanced scenario (JUnit and Mockito): We also study a second case with a more balanced popularity of libraries, focusing on the combination of JUnit and Mockito, which are frequently used together.

Notation: To describe the results obtained in these empirical scenarios, we maintain consistency with the notation used in the simulation. Library

high corresponds to JavaX in the imbalanced scenario and to JUnit in the balanced scenario, and the *low* library corresponds to MicroProfile in the imbalanced scenario and to Mockito in the balanced scenario.

Table 2: Size of the different samples

Sample	Imbalanced Scenario		Balanced Scenario	
	Repos.	**Observ.**	**Repos.**	**Observ.**
$\mathbb{P}$	?	?	?	?
$\mathbb{R}$	754	1,254,633	754	1,254,633
$\mathbb{H}$	2,693	19,852,532	2127	40,649,726
$\mathbb{H}'$	2,335	3,260,983	2016	6,942,311
$\mathbb{L}$	1,317	10,566,780	1055	59,537,772
$\mathbb{L}'$	1,315	142,396	701	440,376
$\mathbb{H} \vee \mathbb{L}$	1,316	11,090,623	951	56,051,500
$\mathbb{H} \wedge \mathbb{L}$	1,123	11,142,344	869	56,343,950

5.2. Characteristics of the Samples

Every sample is a set of client projects. We used the GitHub search API to search for projects that include import sections with a reference to the *high* library, the *low* library, or a combination of both. The characteristics of the samples are summarized in Table 2.

5.2.1. Random Sample

We use the dataset provided in [4] to create a random sample of client projects. The only restriction applied to the sample is a filter on Java as the programming language. We sample for 1% of the Java projects, but only 754 of them include valid observations. The random sample has $1,254,633$ observations in total.

5.2.2. Single Library Samples

For samples $\mathbb{H}$ and $\mathbb{L}$, we use the GitHub Search API to search for Java files containing import statements with the base package name of *high* and *low* libraries, respectively. After we collect the observations in the client projects for $\mathbb{H}$ and $\mathbb{L}$, the samples $\mathbb{H}'$ and $\mathbb{L}'$ are derived by filtering the observations for the presence of the respective library.

For example, in the case of the imbalanced scenario, from the $10,566,780$ observations in $\mathbb{L}$, we only keep those logic facts with a type of MicroProfile,

packaged under org.eclipse.microprofile. This results in $142,396$ observations in $\mathbb{L}'$ (approximately 1% of $\mathbb{L}$). A consequence is that even more data on other libraries is removed from this sample.

5.2.3. Co-used library samples

Similarly to $\mathbb{H}$ and $\mathbb{L}$, we use the GitHub Search API to construct the samples $\mathbb{H} \vee \mathbb{L}$ and $\mathbb{H} \wedge \mathbb{L}$. This search combines the presence of *high* and *low* libraries in a file with the respective logical operators.

For example, in the case of the balanced scenario, we use the query "import org.junit" OR "import org.mockito" language:Java for sample $\mathbb{H} \vee \mathbb{L}$ and the query "import org.junit" AND "import org.mockito" language:Java for sample $\mathbb{H} \wedge \mathbb{L}$.

5.3. Mining Algorithm

This section describes the procedure we designed to mine and compare rules across different samples. First, we construct the different samples as described above. Then, we mine a set of rules from the samples.

The technical details of mining align with the original method in [8]. We use this same mining method, which is association rule mining, with a threshold of 0.001 for support and 0.01 for confidence. We set these thresholds low (less restrictive) to recover as many overlapping rules as possible across the different samples. The thresholds do not influence the computed confidence values, so they are not a threat to the validity of the results.

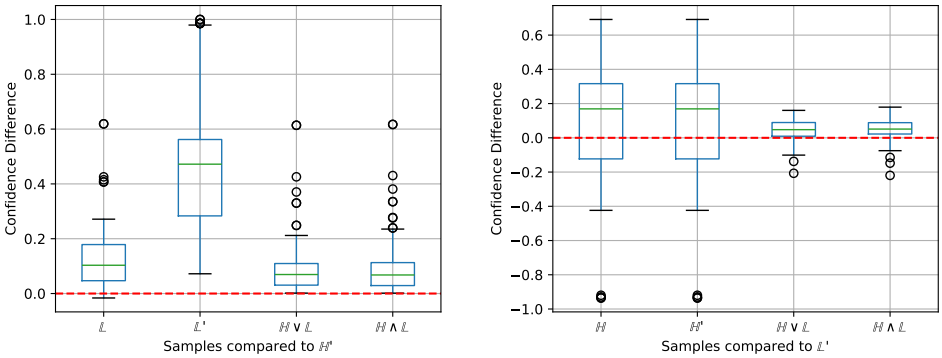
We mine observations that are logic facts for methods and fields. For each method and field, we collect information describing the usage of libraries. Usage information is represented as a set of the following logic constraints:

- The difference between method and field (method vs. field)
- Method's return type (e.g., type(javax.json.JsonString))
- Field's type (e.g., type(javax.json.JsonString))
- Parameter types (e.g., parameterType(javax.json.JsonString))
- Annotation types (e.g., annotation(org.eclipse.microprofile.jwt.Claim))
- Class annotations (containing a method or field) (e.g., classAnnotation(javax.ws.rs.Path))
- Inheritance hierarchy of a class (containing a method or field) (e.g., implements(javax.ws.rs.Path))
- Calls to methods in the body of a method.

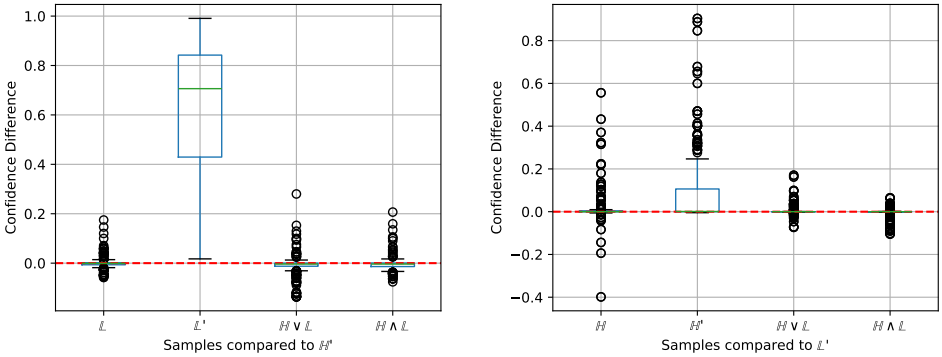
5.4. Rule Selection, Comparisons and Baseline

A key difference between the empirical and simulation studies is that the empirical study lacks confidence values computed on the population and thereby a formal baseline. Thus, this part of the study needs to compare confidence values for the same rule mined from different samples with something else.

Guided by the simulation results, we compare the confidence measures to the best approximations of the population, thereby establishing a formal baseline. Specifically, we use sample $\mathbb{L}'$ for $low \rightarrow high$ and sample $\mathbb{H}'$ for $high \rightarrow low$ (refer to the simulation results in Figure 3). We exclude random samples from this comparison because they yield almost no overlapping rules, rendering the comparison meaningless.



(a) rule $high \rightarrow low$ imbalanced scenario. (b) rule $low \rightarrow high$ imbalanced scenario.



(c) rule $high \rightarrow low$ balanced scenario. (d) rule $low \rightarrow high$ balanced scenario.

Figure 5: Difference in confidence between samples and the baseline for the two scenarios.

5.5. Results

We now compare the rules mined on the samples.

5.5.1. Random Samples

Random sampling is commonly known to work, but it is impractical in our case. For many rules that we see in other samples, the random sample shows numerical errors or 0 confidence computed, rendering the comparison meaningless. We exclude random samples from the plots.

5.5.2. Single Library Samples

In Figure 5, we show the differences between the confidence values by a certain sampling method to our respective approximations of the population. The four charts separately show $high \rightarrow low$ and $low \rightarrow high$ for the imbalanced and balanced scenarios.

- Similar to the simulation, the confidence values computed on *single library* samples targeting the library in the consequent (the 'wrong' side according to the simulation) can overestimate confidence.
- For the more restrictive samples $\mathbb{H}'$ and $\mathbb{L}'$, this overestimation is more pronounced.
- In the balanced scenario, where popularity is more even, we noted that overestimation is only an issue when taking samples $\mathbb{H}'$ and $\mathbb{L}'$ (lower part of Figure 5).

Summarizing these results independent of what we assume approximates the population best; **Insight 1 (empirical)**: For implications of shape $A \rightarrow B$, and using the GitHub Search API, sampling for the library in the antecedent (A) **can** result in **different** confidence values than sampling for the library in the consequent (B). This conforms to the simulation insight.

5.5.3. Co-used library samples

We now discuss co-used library samples. We limit our discussion to the imbalanced scenario since the influence of sampling in the balanced scenario on confidence is negligible. This relates to popularity (see Sec. 5.6.1).

From the simulation results, we expected that sample $\mathbb{H} \vee \mathbb{L}$ would be the one closest to the confidence for both rule types. Here we find a central inconsistency detected by our study. Sample $\mathbb{H} \vee \mathbb{L}$ does not work as expected. We confirm by observing a difference to the baseline for both rule types in the imbalanced scenario.

For sample $\mathbb{H} \wedge \mathbb{L}$, which we do not assume to be correct, the same problems occur; confidence is systematically higher.

5.6. Interpretation GitHub Search API

We conclude that the GitHub Search API is even more of a black box than we originally expected. We already know of several restrictions that must be considered, such as pagination, rate limitations, and bisection strategies. However, the problems with disjunction and conjunction are novel to us, demonstrating a significant lack of transparency in how the API functions. The API does not seem to properly reflect the proportions of the libraries in the population when forming them and this is a significant threat for generalizability as we show in the simulation.

Summarizing these results; **Insight 2 (empirical)**: For implications of shape $A \rightarrow B$ and vice versa $(B \rightarrow A)$, and using the GitHub Search API, sampling for the corresponding library in the antecedent will result in **different** confidence values than sampling for the disjunction $\mathcal{A} \vee \mathcal{B}$. This does not conform to the simulation insight.

5.6.1. Interpretation Popularity

The empirical studies, as well as the simulations, show a decrease in the overestimation in the more balanced popularity scenario when compared to imbalanced. The threat that we detected appears to be limited to highly asymmetric popularity.

6. Threats and Limitations

The following are threats and limiting factors to our work.

- **Generalizability Across Library Pairs**: Evidence that stems from our empirical case study is limited to the combinations of libraries examined. We presented two representative library combinations: JavaX

vs. MicroProfile, and JUnit vs. Mockito. We assume that other library combinations should not deviate significantly from these scenarios, but we cannot guarantee that. We also do not yet know how more complex library structures involving more than just two libraries are related to our findings.

On the other hand, we tried to mitigate threats of overfitting our experiments to a particular library by showing our findings for a very basic rule-mining algorithm. We do not learn parameters that could relate to generalization in the sense of overfitting.

- **Generalizability Across Mining Algorithms**: We do not make general statements about other mining algorithms.

- **Potential Bias from Uncovered Filtering Criteria**: Additional filtering criteria, such as excluding forked repositories, archived projects, and commented code, are relevant and often used in empirical software engineering to reduce noise and improve data quality. For example, forks can be found searching for the same dependencies as used to find the original repository. This can cause libraries to be overrepresented that are used in popular client projects that are forked a lot.

  We assume that influence is marginal in our case. Out of the 2,693 repositories that use JavaX in our empirical study (imbalanced scenario), none were forks, and only 6 were archived.

- **GitHub as a Black Box**: For our empirical study, we used the GitHub search API to construct the samples. While the GitHub search API provides access to a vast number of projects, the particular selection procedure applied by GitHub internally is not transparent to us. This could have a strong effect on the way we 'sample' projects from GitHub using the search API. One might not even call this sampling in the strict sense. Even structuring of code files into repository might have an impact.

  We resolve this with transparency: We deploy the code used for this study to allow for replication, but we cannot mitigate that the behavior of the search API potentially changes. The black box nature of GitHub Search remains a problem that is also part of the following discussion section.

- **Simulation Assumptions**: The simulation study is limited by its assumptions. Simulation results do not hold if the assumptions do not hold. We tried to make the assumptions as realistic as possible. However, most importantly, our assumptions are explicitly given in code, executable, and can be subject to discussion and revision.

7. Related Work

There are many automated methods to extract properties about library usage from a wide range of inputs. For a comprehensive survey, see [23]. Due to the variety, we apply the following restrictions to scope our discussion.

- Our interest is in works that analyze regularities in software projects that use (are clients of) a library. We do not focus on mining artifacts specific to the library [24, 25, 26]. Such work is not relevant for a systematic examination of methods to sample client projects.
- We focus on the static methods because dynamic ones work on data collected from a running program. Such data is fundamentally different from usage-related data and difficult to obtain. Future work should examine the impact of sampling methods on dynamic techniques, too.
- We restrict our scope to works that infer usage patterns that represent relationships between two elements X and Y. We exclude work for retrieving code examples [27] or extracting usage data apart from API usage [28, 29].

Element X and Y can be part of different or the same libraries or types. Our structure follows such a division. The distinction between intra-library, and inter-library usage is borrowed from [30].

7.1. Intra-library Usage Patterns

We first list methods that mine usage patterns that span multiple types of a particular library. For example, a class in a client project may extend or implement two different types from the same library. Examples for this more general type of patterns include [31, 32, 33, 34, 17, 35, 36].

Authors of [8] derive the usages of a target library via the dependency graph of its GitHub repository, and further remove "demo" or "toy" projects. In [35], authors gather relevant code snippets from a code corpus obtained

through another baseline method. In [17], client projects are collected using the domain-specific language and infrastructure called Boa [37].

Approaches like [34, 36, 33] retrieve relevant projects by searching for the name of a library in the import statements of Java source files. Other approaches, like [31], select projects manually.

Additionally, intra-library methods also apply filters to the collected data. The approach proposed in [8] leverages a version of the mining algorithm FP-Growth to mine frequent itemsets and further removes those that do not contain at least one element of the library API. We have shown that this might be harmful for inter-library usage patterns.

Graph-based models are usually built from library-related statements. This is the case of the graph representation (GRAAM) produced in [32]. In [36] and [33], framework extension graphs must have at least one parameter that is related to a framework type. Authors in [34] mine for patterns in graphs derived from client class declarations that subtype a framework type. In [31, 38], only public methods of the API of interest are considered.

The previous work does not mention sampling explicitly, but discusses the data collection, or the data sets. However, in [31], Saied et al. evaluate the generalizability of the detected patterns. The main assumption is that: API usage patterns are generalizable if they have similar usage cohesion degree in different contexts of client programs. In a subsequent paper [38], Saied & Sahraoui intend to improve the generalizability of the mined patterns also considering the library as such. More recent work uses representation learning to improve the recommendation accuracy of statistical approaches, especially for low-frequency APIs [39, 40].

All the discussed papers, except for [8], focus on a specific API and do not analyze possible co-occurrences with APIs from other libraries. Data collection, in particular the sampling, remains a central problem to all of them.

7.2. Inter-library Usage Patterns

We did not find studies that particularly focus on mining inter-library usage patterns. However, there is a group of studies positioned under the term library co-usage.

In [41], authors visualize software component interactions between clients and libraries, and between libraries and other libraries. In the conducted pilot study, the authors selected 101 client projects of 11 libraries from Duets [42]. Sampling is not studied explicitly.

In [43], authors propose interactive tool support for exploration of API usage scoped to a single project. This includes cases where two APIs are used separately in two methods that perform different tasks and yet are in the same project. The authors only restrict the collection of data to open-source Java projects and use the Qualitas corpus [44] in their study. We can see this as a form of random sampling.

In [30], the authors conduct an empirical study on API usages, which explores, among other issues, to what degree programmers work with APIs from different libraries. Authors count API usages involving single or multiple libraries. In this case, the authors try to ensure the representativeness of the collected data and select both small and large projects from various categories such as databases, servers, platforms, and API libraries. This is a form of stratification in sampling that we did not yet examine. In [16], authors conduct a large-scale study on Java and Android, and assess if popular APIs tend to be used together. None of the previous work systematically examines the impact of sampling methods on the quality of the mined patterns.

8. Discussion

Table 3 summarize the results of our hybrid study in terms of the different types of patterns explored and the sampling methods where these patterns best generalize to the population.

Table 3: Sampling methods that approximate results to the population. ($*$ Theoretical best approximation, less accurate than in the simulation)

		Simulation		Empirical	
Type of pattern	Rule examined	Available baseline	Approximates to baseline	Available baseline	Approximates to baseline
Intra-library	$high1 \rightarrow high2$	$\mathbb{P}$	All	Not examined	
Inter-library	$high \rightarrow low$		$\mathbb{H}$ $\mathbb{H}'$ $\mathbb{H} \vee \mathbb{L}$	$\mathbb{H}'$	$\mathbb{H} \vee \mathbb{L}_*$
	$low \rightarrow high$		$\mathbb{L}$ $\mathbb{L}'$ $\mathbb{H} \vee \mathbb{L}$	$\mathbb{L}'$	

From the insights presented in Sec. 3.3, we derived two main conclusions:

- When mining from client software projects, the sampling method used influences the generalizability of the mined inter-library usage patterns; see **insight 1 and 2 (simulation) and insight 1 (empirical)**.

- Open-source forges, like GitHub, remain black boxes, and data collected from them should be treated with caution; see **insight 2 (empirical)**.

We discuss these ideas in detail in the following subsections.

8.1. Sampling when Mining for Inter-library Usage Patterns

Our study shows that inter-library usage patterns in the form of implications $A \rightarrow B$, where A and B involve different libraries, are complicated to mine from samples. In particular, this is the case when sampling implies an upfront filter by the libraries.

In a simulation study, we show that only when sampling for the disjunction of library A and B, the patterns generalize well. Sampling for library A will provide misleading results for rules of the form $B \rightarrow A$; sampling for library B will provide misleading results for rules of the form $A \rightarrow B$. This poses a conceptual limitation for eventual approaches that need to mine both directions.

Our empirical study shows that real empirical data sampled from GitHub does not behave as we would expect from our simulation. Two important sampling methods that should give the same correct result behave differently. That means that one of the two sampling methods is definitely malfunctioning. We conclude that the GitHub search API is even more of a black box than we originally expected.

This aligns with the findings of previous studies [3, 45, 46]. Although it is a rich source of data on software development, mining GitHub for research purposes must consider several pitfalls, like that some data is excluded entirely, large files and long lines lead to truncated data and search is not exhaustive[2].

In essence, we don't understand the indexing mechanisms and cannot be sure what happens behind the GitHub search API. As a consequence, researchers should investigate alternatives with more transparent and comprehensive indexing mechanisms.

8.2. Practical Implications and Recommendations

The main concern of our work is the generalizability of mined inter-library usage patterns. When mining from a *single library* sample there is a risk of

[2] `https://docs.github.com/en/search-github/github-code-search/about-github-code-search#limitations`

discovering patterns involving different libraries that do not generalize to the broader population. The implication of this is relevant for both researchers and developers.

8.2.1. Implications for Researchers

Researchers should be well aware of the possible implications for the validity of the results obtained depending on the applied sampling method. When possible, relying only on single library samples for inter-library patterns (without manual validation) should be avoided. Instead, it is advisable to use random sampling.

Empirical papers studying libraries may have to deal with libraries that are uncommon in the population. Since random sampling may still not work for unpopular or rare libraries, other sampling methods (that combine multiple libraries) should be considered.

The paper shows that sampling projects using multiple libraries (e.g., $\mathbb{H} \vee \mathbb{L}$) could improve generalizability in theory. However, in real-world data, sample $\mathbb{H} \vee \mathbb{L}$ might not behave as expected. It is recommended to clearly state the sampling strategy and its limitations in the threats to validity sections of studies if producing and interpreting inter-library usage patterns.

Another important point to consider by researchers is that having a large data set does not guarantee that results approximate the entire population. Bigger datasets do not mitigate some threats of some sampling strategies. We did show that when mining for patterns with multiple libraries, size might not matter if limiting samples to specific libraries.

8.2.2. Implications for Developers

Mining of usage patterns has been widely adopted to automate various software engineering tasks, such as bug and API misuse detection, code completion and recommendation systems, API documentation, and more [47]. The threats discussed in our paper are relevant for such tasks, especially when the mined patterns are practically used in predictions across libraries.

Overestimation in the confidence of patterns extracted from single library samples can lead to false positives during bug prediction activities and irrelevant recommendations in code completion systems or API documentation techniques. Recommender systems could confidently suggest usage patterns that do not work well outside this sample. Models trained on large samples but obtained by limited sampling methods may show good performance in

the test environments but fail in real-world use. Even cross-validation will not help if the original data is produced by a biased sampling method.

Developers using these tools and techniques that rely on mined patterns should handle inter-library suggestions with caution, especially for rare libraries. Developers should be more skeptical about patterns for very unpopular libraries. In popular libraries, this seems to be less of a problem.

8.2.3. Getting the Samples from GitHub

The last insight we can contribute is about a problem mostly out of our reach (for developers and researchers). GitHub acts as a black box in the sense that the way it stores, exposes, filters, and limits its data is not fully transparent and accessible for us. The limitations of the GitHub search API can be summarized by two main points:

1. Incomplete or unavailable data
2. Biased search

The GitHub search API do not expose all data about repositories. Some historical data may be unavailable if a repository was deleted or made private, and certain events are missing or truncated (e.g., long issue comments). Additionally, API rate limits restrict data extraction, especially for unauthenticated requests, making it difficult to collect large datasets.

On the other hand, the GitHub search API does not guarantee returning all matching results, as searches are ranked, limited to a maximum number, and filtered using undisclosed criteria. This means older or less popular repositories may be excluded, and search results may differ between identical queries. These limitations could be the reason for the discrepancies observed in our empirical study.

As a consequence, researchers and practitioners looking to study inter-library usage patterns should seek alternative datasets with transparent indexing, explore other code forges, or mirror data from other sources such as: GHTorrent [48] or GHS (GitHub Search) [4]. Combining data from diverse sources and validating findings across different sampling strategies, rather than relying solely on GitHub, is recommended for more representative data. Finally, all technical limitations, including data constraints and rate-limit impacts, should be explicitly documented as threats to validity.

9. Conclusion

For the specific case of mining inter-library usage patterns, we present an empirical study showing that mining on data sampled from GitHub does not behave as expected based on a corresponding simulation. Specifically, two sampling methods that should theoretically yield the same results instead produce different outcomes. This discrepancy suggests that at least one of the sampling methods is unreliable, assuming our simulation assumptions hold. Our findings highlight that the GitHub search API operates more as a black box than previously anticipated.

At its core, this study examines the generalizability of findings in a specific area of software engineering. Our insights have practical implications: they can inform the choice of sampling methods in future research, or be highlighted as potential threats to validity in similar studies.

Relying only on data from single-library sampling for inter-library patterns (without manual validation) can weaken the validity of results. Random sampling is better but often not practical for rare libraries. Combining data from multiple sources and considering alternative mirror datasets like GHTorrent is recommended when possible.

Researchers and practitioners must stay alert. Recognizing and documenting the limitations of data collection and the black box nature of the GitHub search API is important to ensure the validity of mining studies. Developers using mined usage patterns in tools should also be careful, especially when working with patterns involving less popular libraries.

Future work should conduct more experiments to confirm these findings in similar settings. It is important to address and resolve the issues identified in this study. Further research should develop methods that help ensure the generalizability of results when mining inter-library usage patterns. This includes providing more transparent indexing mechanisms as alternatives to GitHub.

[1] P. S. Levy and S. Lemeshow, *Sampling of populations: methods and applications*. John Wiley & Sons, 2013.

[2] S. Baltes and P. Ralph, "Sampling in software engineering research: a critical review and guidelines," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 94, 2022.

[3] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Findings from github: methods, datasets and limitations," in *MSR*. ACM, 2016, pp. 137–141.

[4] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *MSR*. IEEE, 2021, pp. 560–564.

[5] W. J. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, "The app sampling problem for app store mining," in *MSR*. IEEE Computer Society, 2015, pp. 123–133.

[6] G. Gousios, "The ghtorent dataset and tool suite," in *MSR*. IEEE Computer Society, 2013, pp. 233–236.

[7] S. Ghaisas, P. Rose, M. Daneva, K. Sikkel, and R. J. Wieringa, "Generalizing by similarity: lessons learnt from industrial case studies," in *CESI@ICSE*. IEEE Computer Society, 2013, pp. 37–42.

[8] B. Nuryyev, A. K. Jha, S. Nadi, Y. Chang, E. Jiang, and V. Sundaresan, "Mining Annotation Usage Rules: A Case Study with MicroProfile," in *ICSME*. IEEE, 2022, pp. 553–562.

[9] P. T. Nguyen, R. Rubei, J. D. Rocco, C. D. Sipio, D. D. Ruscio, and M. D. Penta, "Dealing with popularity bias in recommender systems for third-party libraries: How far are we?" in *MSR*. IEEE, 2023, pp. 12–24.

[10] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *ICSE*. ACM, 2008, pp. 391–400.

[11] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *Datalog*, ser. Lecture Notes in Computer Science, vol. 6702. Springer, 2010, pp. 245–251.

[12] C. D. Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *PPPJ*. ACM, 2011, pp. 71–80.

[13] C. C. Aggarwal, *Data Mining - The Textbook*. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-14142-8

[14] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD Conference*. ACM Press, 1993, pp. 207–216.

[15] A. Ceglar and J. F. Roddick, "Association mining," *ACM Comput. Surv.*, vol. 38, no. 2, p. 5, 2006.

[16] C. Lima and A. C. Hora, "What are the characteristics of popular apis? A large-scale study on java, android, and 165 libraries," *Softw. Qual. J.*, vol. 28, no. 2, pp. 425–458, 2020.

[17] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static api-misuse detection," in *MSR*. IEEE / ACM, 2019, pp. 265–275.

[18] T. P. Morris, I. R. White, and M. J. Crowther, "Using simulation studies to evaluate statistical methods," *Statistics in medicine*, vol. 38, no. 11, pp. 2074–2102, 2019.

[19] J. Härtel and R. Lämmel, "Operationalizing threats to MSR studies by simulation-based testing," in *MSR*. ACM, 2022, pp. 86–97.

[20] ——, "Operationalizing validity of empirical software engineering studies," *Empir. Softw. Eng.*, vol. 28, no. 6, p. 153, 2023.

[21] J. Härtel, "Improved labeling of security defects in code review by active learning with llms," in *EASE*, 2025, to appear.

[22] G. W. Imbens and D. B. Rubin, *Causal inference in statistics, social, and biomedical sciences*. Cambridge University Press, 2015.

[23] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2012.

[24] R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *SAC*. ACM, 2011, pp. 1317–1324.

[25] D. Qiu, B. Li, and H. Leung, "Understanding the API usage in java," *Inf. Softw. Technol.*, vol. 73, pp. 81–100, 2016.

[26] J. Härtel, H. Aksu, and R. Lämmel, "Classification of apis by hierarchical clustering," in *ICPC*. ACM, 2018, pp. 233–243.

[27] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 5653. Springer, 2009, pp. 318–343.

[28] M. A. Saied, A. Ouni, H. A. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *J. Syst. Softw.*, vol. 145, pp. 164–179, 2018.

[29] C. Thiede, W. Scheibel, D. Limberger, and J. Döllner, "Augmenting library development by mining usage data from downstream dependencies," in *ENASE*. SCITEPRESS, 2022, pp. 221–232.

[30] H. Zhong and H. Mei, "An empirical study on API usages," *IEEE Trans. Software Eng.*, vol. 45, no. 4, pp. 319–334, 2019.

[31] M. A. Saied, O. Benomar, H. Abdeen, and H. A. Sahraoui, "Mining multi-level API usage patterns," in *SANER*. IEEE Computer Society, 2015, pp. 23–32.

[32] A. Shokri, J. C. S. Santos, and M. Mirakhorli, "Arcode: Facilitating the use of application frameworks to implement tactics and patterns," in *ICSA*. IEEE, 2021, pp. 138–149.

[33] Y. Pacheco, J. D. Bleser, T. Molderez, D. D. Nucci, W. D. Meuter, and C. D. Roover, "Mining scala framework extensions for recommendation patterns," in *SANER*. IEEE, 2019, pp. 514–523.

[34] Y. Pacheco, A. Zerouali, and C. D. Roover, "Mining for framework instantiation pattern interplays," in *SCAM*. IEEE, 2022, pp. 121–131.

[35] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of API usage examples," in *ASE*. IEEE, 2019, pp. 590–601.

[36] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Recommending framework extension examples," in *ICSME*. IEEE Computer Society, 2017, pp. 456–466.

[37] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *ICSE*. IEEE Computer Society, 2013, pp. 422–431.

[38] M. A. Saied and H. A. Sahraoui, "A cooperative approach for combining client-based and library-based API usage pattern mining," in *ICPC*. IEEE Computer Society, 2016, pp. 1–10.

[39] C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for API usage recommendation," in *SANER*. IEEE, 2021, pp. 36–47.

[40] Y. Chen, C. Gao, X. Ren, Y. Peng, X. Xia, and M. R. Lyu, "API usage recommendation via multi-view heterogeneous graph representation learning," *IEEE Trans. Software Eng.*, vol. 49, no. 5, pp. 3289–3304, 2023.

[41] S. Venkatanarayanan, J. Dietrich, C. Anslow, and P. Lam, "Vizapi: Visualizing interactions between java libraries and clients," in *VISSOFT*. IEEE, 2022, pp. 172–176.

[42] T. Durieux, C. Soto-Valero, and B. Baudry, "Duets: A dataset of reproducible pairs of java library-clients," in *MSR*. IEEE, 2021, pp. 545–549.

[43] C. D. Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of API usage," in *ICPC*. IEEE Computer Society, 2013, pp. 152–161.

[44] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *APSEC*. IEEE Computer Society, 2010, pp. 336–345.

[45] G. Gousios and D. Spinellis, "Mining software engineering data from github," in *ICSE (Companion Volume)*. IEEE Computer Society, 2017, pp. 501–502.

[46] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian, "An in-depth study of the promises and perils of mining github," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2035–2071, 2016.

[47] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[48] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *MSR*. IEEE Computer Society, 2012, pp. 12–21.