Noname manuscript No.
(will be inserted by the editor)

An Empirical Study on API Usages from Code Search Engine and Local Library

Hao Zhong · Xiaoyin Wang

Received: date / Accepted: date

Abstract To help programmers find proper API methods and learn API usages, researchers have proposed various code search engines. Given an API of interest, a code search engine can retrieve its code samples from online software repositories. Through such tools, Internet code becomes a major resource for learning API usages. Besides Internet code, local library code also contains API usages, and researchers have found that library code contains many API usages that are more concise than those in client code. As samples from a code search engine are typically client code, it is interesting to explore the API usages inside library code, but the samples inside library code contain internal method invocations. If an empirical study does not remove them from API usages, it can significantly overestimate the API usages from library code. Due to this challenge, no prior study has ever analyzed API usages inside libraries, and many research questions are still open. For example, how many API usages are there inside libraries? The answers are useful to motivate future research on APIs and code search engines.

The internal usages in library code will introduce compilation errors when they are directly called from the client side. To support the exploration of the above questions, in this paper, we propose CodeEx that extracts Internet code samples from a popular code search engine and local code samples by removing internal usages from library code. With the support of CodeEx, we conduct the first empirical study on API usages of five libraries, and summarize our results into six findings as the answers to five research questions. Our results are useful for researchers to motivate their future research. For example, our results show that although code samples from library code are

H. Zhong
Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
E-mail: zhonghao@sjtu.edu.cn

X. Wang
Department of Computer Science, University of Texas at San Antonio, USA
E-mail: Xiaoyin.Wang@utsa.edu

only half of those from the code search engine, they cover 4.0 times more API classes, 4.7 times more API methods, and 3.0 times more call sequences. Meanwhile, in a controlled experiment, we compare their effectiveness in assisting programming. We find that more API usages do not lead to more complete tasks, and it highlights the importance of code recommendation approaches.

1 Introduction

Application programming interface (API) libraries become increasingly popular in both open-source and commercial projects because they can reduce implementation effort. Despite their popularity, programmers often complain that it is difficult to learn API usages [55,49]. Since there are many APIs, even experienced programmers often encounter unfamiliar ones. To learn the usages of those unfamiliar APIs, programmers typically use code search engines [42, 38] to retrieve code samples, and some engines (*e.g.*, SearchCode [8]) are built upon Internet-scale repositories.

Despite the above tools, programmers still complain that it is difficult to obtain their required code samples, especially for new and less popular APIs. For example, the prior studies [54,32] show that many programmers are reluctant to update their code, since it is difficult to retrieve valid code samples for new and uncommon APIs. As some code search engines already include millions of lines of code, adding more source files may not be useful to retrieve more API usages. Most samples from code search engines are client code, but the library code declaring and defining the APIs also calls its own APIs in various ways. As library developers are often more familiar with their APIs than client developers, library code can call APIs more effectively than client code. Based on this insight, Kawrykow and Robillard [29] propose an approach to replace API usages in client code with more concise API usages in library code. For example, JBOSS has the following code:

```
Thread th=Thread.currentThread();
ClassLoader loader=th.getContextClassLoader();
Class tC=null;
try {
  tC=Classes.getPrimitiveTypeForName(type);
  if( tC == null )
    tC=loader.loadClass(attrType);
} catch(ClassNotFoundException ignore) { }
PropertyEditor editor=null;
if( tC != null )
  editor=PropertyEditorManager.findEditor(tC);
```

By identifying the library code that implements similar functionalities, Kawrykow and Robillard [29] replace the above code with the following code:

```
PropertyEditor editor = null;
try {
    editor = PropertyEditors.getEditor(type);
} catch (ClassNotFoundException ignore) { }
```

Although the latter code is more concise, its usage is less known to client programmers, so their code illustrates a different and less effective way to call APIs. In total, Kawrykow and Robillard [29] found more than 400 such cases. Although their results show that library code can be used to improve existing API usage samples, to the best of our knowledge, no prior research effort studies the feasibility and potential benefits of directly extracting API usages from library code. While there are already many empirical studies on API usages (see Section 5 for details), none of these studies analyzes API usages in library code. A widely believed difficulty in reusing library code as API usage samples lies in that *library code can invoke APIs together with internal code references (e.g., private code elements) which are not accessible outside the library*. If researchers do not remove internal usages in their empirical studies, they can significantly overestimate the API usages from library code and their answers to research questions can be superficial. It is challenging to remove internal usages. For example, the names of internal code elements seldom have any hints. Due to the challenge, many research questions along with this research line are still open. For example, compared with Internet code, does library code provide extra API usages, and to what degree? These questions are important, since their answers can shed light on new research directions and tools.

In our study, we implement a tool that retrieves Internet code from a code search engine and removes internal usages from library code. With its support, we conduct an empirical study to explore the following research questions:

- **RQ1. What proportion of APIs can have their usage samples extracted from a code search engine called SearchCode?**
  Motivation. The result is useful to understand the situations when programmers use code search engines to retrieve code samples.
  Protocol. SearchCode is a code search engine, and its repository includes billions of lines of code collected from millions of projects. Based on the Web APIs of SearchCode, we implement a tool called CodeEx to analyze the API coverage of SearchCode.
  Result. We find that even with such a large repository, except for one case (lucene, 21.3%), our retrieved *relevant* and *up-to-date* samples averagely cover only 9.3% of API classes (Finding 1). Our results explain why programmers complain that code samples are insufficient.
- **RQ2. What proportion of APIs can have their usage samples extracted from local library code?**
  Motivation. Programmers need code samples that illustrate more API usages, especially for those new and less popular libraries. As SearchCode already has millions of projects, it is less useful to add more projects. Instead, this research question explores the API coverage of libraries.
  Protocol. To eliminate the bias of internal usages, we implement CodeEx that removes internal usages from libraries. With its support, we analyze to what proportion of APIs has their usage samples from the library code.

Result. For 90% API classes with at least one usage sample available, library code contains more code samples than Internet code. Library code covers 4.0 times more API classes, 4.7 times more API methods, and 3.0 times more call sequences than Internet code (Finding 2).

- **RQ3. What are the characteristics of samples?**
  Motivation. While researchers [60] mine patterns from Internet code, this research question explores whether similar techniques work on library code.
  Protocol. In this research question, we compare the API usages from Internet code with those from library code.
  Result. The code samples from library code are less repetitive than those from Internet code (Finding 3).
- **RQ4. To what degree are API usages overlapped?**
  Motivation. The answers are useful to understand whether one source can replace the other.
  Protocol. We explore the overlapped API usages.
  Result. We find that the API usages from the two sources have 90% overlapped API classes and 80% overlapped methods, but the overlapped samples reduce to about 20%, when we consider their call sequences (Finding 4). The results indicate that the API usages from the two sources can complement each other, and it can be interesting to further analyze their different call sequences.
- **RQ5. To what degree can programmers obtain benefits from code samples that illustrate more API usages?**
  Motivation. The results are useful to understand the usefulness of API usages in assisting programming.
  Protocol. We prepared 20 programming tasks, and in a controlled setting, we compared their completed programming tasks in the two treatments of using samples from either source.
  Result. We found that for cases where usage samples can be extracted from both sources, samples extracted from library code have similar effectiveness compared with those extracted from Internet code in assisting developers. However, usage samples from library code cover more cases and help developers in more tasks. The result highlights the importance of code recommendation techniques, since it is ineffective for programmers to manually identify useful samples when samples are many.

2 Methodology

Section 2.1 introduces our subjects. We select a popular code search engine called SearchCode [3] to retrieve Internet code samples. Library code contains internal usages (*e.g.*, private method calls). If we count them as API usages, we will overestimate API usages from library code. To support our study, we implement a support tool called CodeEx that extracts Internet code from SearchCode (Section 2.2) and removes internal calls from library code (Section 2.3). Section 2.4 introduces our analysis protocol.

Table 1: The libraries.

Name	Version	Class	Method	URL
accumulo	1.9.2	2,839	26,678	drzhonghao.github.io/accumulodoc/
cassandra	3.11.2	2,273	17,325	drzhonghao.github.io/cassandradoc/
karaf	4.2.3	998	4,755	drzhonghao.github.io/karafdoc/
lucene	7.4.0	2,414	12,189	drzhonghao.github.io/lucenedoc/
poi	4.0.1	2,432	21,045	drzhonghao.github.io/poidoc/

2.1 Subject

Table 1 shows the subjects: `accumulo` [1] allows storing and managing large data across clusters; `cassandra` [2] is a database for data centers; `karaf` [5] is an application runtime for the enterprise; `lucene` [6] is an indexing and search library; and `poi` [7] is a library for manipulating Microsoft documents. We selected different projects to show the impacts of factors such as size and popularity. All the libraries are shipped with their API documents. For example, although `cassandra` [2] does not provide online API documents, its offline API documents are compressed in its released files [9]. APIs evolve rapidly across versions, but their projects do not present online documents for all versions. For example, `accumulo` provides the API documents of `1.9` [10], but does not present the API documents of `1.9.2`. `ClientContext` is an API class of `accumulo 1.9.2`, but does not belong to `1.9`. To avoid ambiguity, we upload the API documents of all the libraries. Column "URL" shows their urls. Columns "Class" and "Method" list the numbers of API classes and methods as defined in their documents.

2.2 Retrieving Internet Code

As SearchCode [8] is a popular code search engine, we used SearchCode to retrieve code samples for all the APIs of our subjects. The repository of SearchCode includes billions of lines of code collected from millions of projects. As the libraries in Table 1 have thousands of APIs, it is infeasible to manually query their code samples. As described in Section 3.1.1, to resolve the problem, we implement CodeEx upon the interfaces of SearchCode [3], and it extracts samples for all the API classes of each library. SearchCode locates relevant code samples by matching text contents. If a code sample calls a `t` type, to resolve its reference, the sample must have an `import` statement: `import t.name;`, where `t.name` is the full name of `t`. As full names of types appear in code samples, for each API class, our tool searches its full name for its code samples. CodeEx removes the following samples:

1. Irrelevant code samples. A retrieved sample can be irrelevant when it has unused `import` statements. Most compilers give warning for them, but do not produce compilation errors. If the full name of `t` is only matched in an unused `import` statement, SearchCode will return the code sample, but the sample does not call any methods of the class. Our tool removes a code sample of a type, if this type appears in unused `import` statements.

2. Obsolete code samples. When we calculate how many APIs are covered by code samples, we must determine the complete set of API elements. In this study, we use the API elements declared by the API documentation of the latest library as the complete set. Although obsolete code samples are useful to learn old libraries, we have to remove them, since they call many API elements that do not appear in the complete set of our study. From each code sample of a library, our tool extracts all its called API classes and methods. If a called class or method is not declared by the version of interest, our tool removes the code sample. It should be noted that many popular libraries release new versions rapidly. For example, since April 2019, `lucene` has released a newer version (`8.2.0`). From such newer versions, it is more difficult to retrieve up-to-date samples for developers than the versions in Table 1.

A retrieved code sample is partial code because its dependent source files are not retrieved. Even for a tool, removing the above two types of samples is nontrivial. To handle this issue, we build our tool on PPA [20], a tool for partial code analysis. As partial analysis is difficult, PPA may fail to resolve the full names of some code elements. Our tool takes a conservative strategy to handle unresolved code elements. If PPA fails to resolve the full name of a class, our tool will not consider its enclosing code sample as irrelevant. From each code sample, CodeEx extracts its called API classes, methods, and fields, and matches them against the API elements declared by the API documentation of the latest libraries to locate obsolete code samples. If the API usage of a code sample does not violate the backward compatibility, it will not be removed.

2.3 Removing Internal Call

Given an API class of a library as a query, CodeEx extracts its relevant code snippets from the library. The JDT [4] extends Eclipse to a Java IDE, and implements various features (*e.g.*, searching for local files). CodeEx extends the local search of JDT. Given an API class of a library, CodeEx searches the library for the source files that contain the references of the class. As libraries provide complete code, CodeEx is able to use JDT to resolve the full names of code elements, which is more accurate than a partial analysis tool like PPA. Like other code search engines, the local search of JDT can retrieve irrelevant code snippets. CodeEx takes the same criteria as described in Section 2.2 to remove irrelevant code snippets. As code snippets are extracted from libraries, they are all up-to-date.

When programmers write their code, they typically choose package names that are different from their libraries. As their package names are different, client code cannot call some private and protected code elements that are visible to only library code. To reveal internal usages, after a source file of library code is retrieved, CodeEx first removes its package name so that it cannot access internal usages anymore (*e.g.*, protected methods from classes in the same package). After the package name is removed, a class may not access even the public classes in this package. To resolve the problem, CodeEx ex-

plicitly imports all classes of the package. From the viewpoint of client-code programmers, internal usages are not accessible. If the client code calls internal usages, it is not compilable or executable. As a result, removing internal usages does not lose any useful API usages. From the viewpoints of programmers, CodeEx does not decrease the quality of the code samples from libraries. From the viewpoint of our study, the impact is mixed. On one hand, removing internal calls can drastically change samples, so it can introduce negative impacts to library code. On the other hand, a tool typically recommends partial source files. As such source files produce various compilation errors, it is difficult for programmers to manually identify internal usages. After internal usages are removed, programmers do not need to identify them.

For each source file, CodeEx extends JDT to compile it. The compilation reports code elements with errors and their error types. CodeEx extracts these error types, and implements removing rules (referred to as *removers* as follows) to resolve our found error types. We next introduce the removers.

ROI1. Removing statements with internal usages. If a statement calls an internal code element of a library, CodeEx removes the statement.

ROI2. Removing a method or a constructor with internal type parameters. If a method or a constructor has a parameter whose type is internal, the method/constructor is removed.

ROI3. Removing a field or a variable whose types are internal. If the type of a field or a variable is internal, CodeEx removes the declaration of the field/variable.

ROI4. Removing super types and super interfaces that are internal. If a class extends an internal type, CodeEx removes the extension. If a class implements an internal interface, CodeEx removes it from its implemented interfaces.

ROI5. Removing incompatible anonymous types or enumerations. If a class declares an anonymous class or an enumeration, after we remove its package name, the anonymous class or the enumeration can introduce compilation errors. For example, Figure 1 shows a source file of cassandra. Line 7 compares whether the current operating system is Linux. In this line, the Java compiler resolves that the type of `osType` is `OSType` in Line 2, but resolves that `LINUX` is declared by `org.apache.cassandra.utils.NativeLibrary`, which is library code. As a result, the Java compiler reports that the operand types are incompatible. CodeEx removes the `if` check statement in Line 7 to resolve the problem. Anonymous classes and enumerations can lead to other types of compilation errors, such as in the following code snippet.

```
1 AutoSavingCache<KeyCacheKey, RowIndexEntry> keyCache = new
      AutoSavingCache<>(kc, CacheType.KEY_CACHE, new
      KeyCacheSerializer());
```

In the above statement, `CacheType.KEY_CACHE` is an enumeration. As the corresponding parameter type of the constructor is resolved as an internal code element, the Java compiler fails to infer the type arguments for `AutoSavingCache`. CodeEx removes such statements to handle the problem.

```
1 public final class NativeLibrary{
2   public enum OSType{LINUX, MAC, WINDOWS, ...;}
3   private static final OSType osType;
4   private static final int MCL_CURRENT;...
5   static{...
6    osType = getOsType();
7    if(osType == LINUX) {
8        MCL_CURRENT = 0x2000;...}
9   }...
```

Fig. 1: Incompatible enumeration

ROI6. Removing incompatible casts and type checks. After we change the package name of a class, its class hierarchy is changed, and `cast` statements and type checks can introduce compilation errors, such as follows:

```
1 abstract class AbstractQueryPager ...{
2  public boolean isExhausted(){
3    return exhausted || remaining == 0 || ((this instanceof
       SinglePartitionPager) && remainingInPartition == 0);}}
```

In cassandra, `AbstractQueryPager` is the super type of `SinglePartitionPager`, and Line 3 does not have compilation errors. After we remove the package name of `AbstractQueryPager`, it is no longer the super type of `SinglePartitionPager`. As a result, Line 3 produces a compilation error. CodeEx removes the corresponding `cast` statements and type checks to handle the problem.

CodeEx defines the following removers to handle consequential errors:

ROM1. Removing `catch` clauses. An internal method call can throw an exception (`e`). When calling the method, programmers can enclose the method call with a `try` statement, and handles `e` in a `catch` clause (`cat`). After the internal call is removed, `cat` introduces a compilation error, since the remaining statements inside the `try` statement do not throw `e` anymore. To handle the problem, CodeEx removes `cat` from the `try` statement. If `cat` is the only catch clause and `e` is the only handled exception, CodeEx takes its inner statements out of the `try` statement, and then removes the `try` statement.

ROM2. Removing unknown variables. If the type of a variable is internal, ROI3 removes the declaration statement of this variable. After that, in the following statements, the variable becomes unknown, and introduces compilation errors. At each time, CodeEx removes a statement that uses the variable, until no more such compilation errors are found.

ROM3. Removing annotations. Java annotations provide metadata for code. For example, the annotation, `@Override`, denotes that a method of a class (`c`) overrides a method that is declared by the superclass of `c`. If `c` is an internal type, ROI4 removes it, and after the removal, the `@Override` annotations will lead to compilation errors, since the superclass is removed. When such compilation errors are found, CodeEx removes the corresponding annotations to resolve the problem.

ROM4. Removing `final` modifiers. After some statements are removed, `final` values can become uninitialized. For example, in Figure 1, after ROI5 removes the `if` statement in Lines 8, the `final MCL_CURRENT` field is not initialized,

so the removal leads to a compilation error, which complains that the two fields are not initialized. To handle the problem, CodeEx removes the `final` modifiers of the two fields.

ROM5. Removing constructors that call undefined super constructors. If a constructor calls a super constructor and the call is removed, the constructor will produce a compilation error. For example, a piece of code is as follows:

```
1 private class StaticLeaf extends Leaf{
2  public StaticLeaf(Iterator<Token> tokens, Leaf leaf){
3    this(tokens, leaf.smallestToken(), leaf.largestToken(), leaf.
         tokenCount(), leaf.isLastLeaf());
4 }...}
```

In the above code, Line 3 calls an internal method, `isLastLeaf()`. The removal of this line introduces another compilation error, since `Leaf` does not define a constructor without parameters. To handle the problem, CodeEx removes the whole constructor in Line 2.

ROM6. Generating `return` statements. If a `return` statement contains internal usages, CodeEx will remove the statement. After the `return` statement is removed, the remaining code will produce a compilation error, since the method requires a `return` statement. To synthesize the statement, CodeEx resolves the return type of the method. If it is a number value (*e.g.*, integer), CodeEx adds `return 0`; if it is a Boolean value, it adds `return true`; and for other cases, it adds `return null` to the end of the method.

ROM7. Generating value initializers. A value can be initialized by internal method calls. After the internal calls are removed, the value is not initialized, and can cause compilation errors, if the latter statements access the variable. To handle the problem, CodeEx generates an assignment statement, before the statement that produces the compilation error. In the assignment statement, the variable is assigned to zero, if it is a number value; `true`, if it is a boolean value; and `null`, if it is an object.

ROM8. Removing duplicated assignments to `final` variables. If more than one statement accesses an uninitialized variable, ROM6 can generate more than one assignment statement. In such cases, if the variable is `final`, the duplicated assignment leads to a compilation error. To handle the problem, CodeEx removes the assignment statement that reports the error, *i.e.*, the latter one.

ROM9. Removing dead code. Removing and generating statements can lead to dead code. For example, ROM5 assigns `null` or zero values to uninitialized variables. Such values can be used in latter loop statements, which lead to infinitive loops. When it happens, the Java compiler reports the statements after infinitive loops as unreachable code or dead code. CodeEx removes dead code to resolve the problem.

CodeEx applies the above removers until all compilation errors are removed. As libraries are complete code, internal usages cause compilation errors, but as most code samples from the Internet cannot compile, we cannot

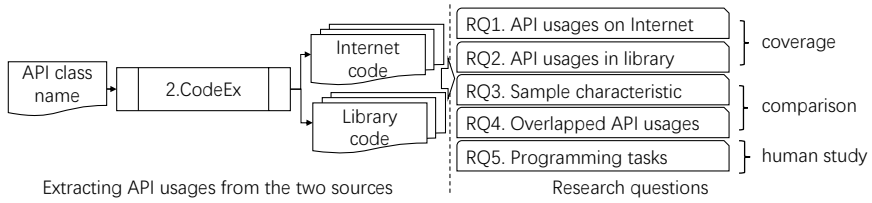


Fig. 2: The overview of our study

use compilation errors to guide the removal of internal usages. As a result, in our study, we remove the internal usages only from library code.

2.4 Analysis Overview

Figure 2 shows the overview of our study. Given each class name from the API documentation, CodeEx extracts API samples from SearchCode and local library code. Based on the code samples from the two sources, in total, we analyze five research questions. RQ1 and RQ2 analyze how many APIs appear in the code samples from the two sources. RQ3 analyzes the characteristics of their samples. RQ4 analyzes the overlapped samples. The two RQs concern the characteristic of samples from the two sources. In RQ5, we prepared 20 programming tasks, and invited four students to complete them. We put the four students into two teams, and split the tasks into two phases. In each phase, one team is asked to switch its role in our controlled experiment. In one role, students searched SearchCode for code samples, and in the other role, students used the code samples of CodeEx. We prepared test cases for all the tasks, and counted the number of tasks that were accomplished under different settings to measure their quality.

2.5 Discussion

Although API tutorials are useful to learn API usages, they cover only a small portion of APIs, since it is expensive to write API tutorials. As a comparison, the code samples from a code search engine or libraries can cover much more APIs. Unlike the code samples from API tutorials, the code samples from a code search engine or libraries are not written to illustrate API usages. As a result, in most cases, the API usages from the two sources often scatter in multiple methods. For example, if two API methods shall be called in pairs and an API method is called by a client method, the other API method can be called by other client methods or even by other client classes. As another example, the code samples from libraries can lose some API calls, since calling their internal alternative methods is more effective for library developers. As a result, programmers often retrieve only partial API usages from both sources. Still, partial usages are useful for programmers. For example, given an input type and an output type, Jungloid [39] can construct the missing API calls

between the two types. As another example, given a set of input types and an output type, XSnippet [59] can recommend code samples whose input and output types match the given ones. Although the code samples from Jungloid and XSnippet cannot illustrate full API usages, they are useful in specific programming contexts. Indeed, programmers often need partial but critical API usages. For example, many approaches are proposed to mine API patterns between two API method calls [64,43,14]. Programmers can learn such short patterns, even if code samples provide partial API usages. As larger API patterns (*e.g.*, sequential patterns [75] and graphs [37,58]) are desirable, Gabel and Su [22] have proposed an approach to merge shorter patterns into larger specifications. In addition, researchers [53,33,40] propose to mine complex specifications in the formats of temporal logic. Complex specifications define API usages precisely, but static analysis is inaccurate. As a result, complex specifications are typically mined from traces that are collected through dynamic analysis. After their internal usages are removed, source files from libraries are compilable, and it becomes feasible to mine complex specifications from library code. Meanwhile, most source files from code search engines are not compilable. Given a source file with compilation errors, researchers [73,26] have proposed approaches to repair its contexts (*e.g.*, library dependencies). After their compilation errors are all removed, complex specifications can be mined from Internet code.

3 Empirical Result

This section introduces detailed protocols and empirical results. More details are listed on our project website: `https://github.com/drzhonghao/apistudy`

3.1 RQ1. API Usages on Internet

3.1.1 Protocol

In this research question, given an API class of a library as a query, we use CodeEx to extract its samples from Internet code. As the repository of SearchCode contains millions of projects, for some popular APIs, it retrieves many pages of samples. As it is not affordable to download and analyze all the samples, we limit the analysis scope of our tool to the top 20 pages. Although the pages beyond the top 20 pages can illustrate some non-popular API usages, programmers are unlikely to read them, since they typically read only several top pages. The prior studies analyze even fewer samples. For example, when Sim *et al.* [56] analyze the quality of samples that are retrieved by code search engines, they analyze only the top ten samples.

Table 2: The retrieved Internet code samples.

Name	Star	Sample	%S	Real	%R
accumulo	884	74	2.6%	35	1.2%
cassandra	6.8k	466	20.5%	163	7.2%
karaf	549	332	33.3%	55	5.5%
lucene	395	1,322	54.8%	514	21.3%
poi	1.2k	471	19.4%	252	10.4%
total		2,665	24.3%	1,019	9.3%

Table 3: The extracted library code samples.

Name	Clean	Fix	Error	%	Number
accumulo	714	179	21	89.5%	893
cassandra	860	245	46	84.2%	1,105
karaf	316	195	66	74.7%	511
lucene	1,238	407	87	82.4%	1,645
poi	1,198	377	72	84.0%	1,575
total	4,326	1,403	292	82.8%	5,729

3.1.2 Result

Table 2 shows our results. Column "Star" lists the number of stars that are received from GitHub. Column "Sample" lists the number of API classes that have retrieved sample (maybe irrelevant or obsolete). Column "%S" is calculated as $\frac{Sample}{Class}$. We find that `lucene` is the mostly covered library. The next is `karaf` (33.3%), but its high coverage may be due to its small size. For `cassandra` and `poi`, the retrieved code samples cover about 20% of their classes, although they are popular. The retrieved code samples cover only 2.5% of classes from `accumulo`. Column "Real" lists the number of classes with real samples (*i.e.*, after removing irrelevant and obsolete samples). We find that even for the best library (`lucene`), the retrieved samples cover only 21.3% of classes, and for `accumulo`, the percent is reduced to 1.2%. The observations lead to our finding:

Finding 1. To sum up, even searching samples in a huge repository like SearchCode, from Internet code, we can collect relevant and up-to-date samples for only 9.3% of API classes on average.

As the source files of our libraries are much fewer than the repository of SearchCode, in total, fewer samples were extracted from library code, but for less popular libraries such as accumulo and karaf, even more samples were extracted from library code than from Internet code.

3.2 RQ2. API Usages in Library

3.2.1 Protocol

We used CodeEx to extract code samples from the library code. To compare the samples from Internet code and library code, we implement another tool. For each sample, this tool collects its called unique API classes, methods, and

Table 4: API usages covered by Internet / library code.

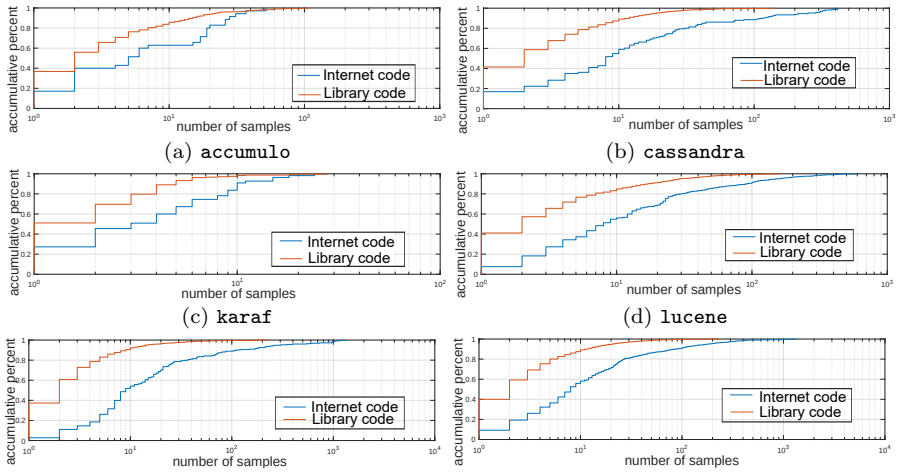
Name	LCC	%	CCM	LCM	CCS	LCS
accumulo	595	21.0%	83	1,994	67	1,875
cassandra	1,156	50.9%	550	4,298	805	3,817
karaf	235	23.5%	107	557	73	442
lucene	871	36.1%	1,292	2,259	2,184	3,104
poi	1,352	55.6%	1,030	5,392	1,458	4,618
total	4,209	38.4%	3,062	14,500	4,587	13,856

LCC: the number of API classes that have samples from library code;
% is calculated as $\frac{LCC}{Class}$, where "Class" is defined in Table 2
CCM and LCM: the numbers of unique API methods that have at least a sample from Internet code and library code, respectively;
CCS and LCS: the numbers of unique call sequences that are extracted from the samples of Internet code and library code, respectively.

call sequences. For each library, the tool sums up the results of all its samples. This tool is built on the Java compiler called JDT. If in a sample, a method (`m`) calls API methods, our tool extracts a call sequence from `m`. As the samples from SearchCode are partial and are not compilable, it is infeasible to introduce advanced static analysis or dynamic analysis to extract their accurate call sequences. CodeEx uses a lightweight static analysis to extract call sequences, and ignores branch conditions. For example, if an API call appears in the `if` branch and another API call appears in the `else` branch, CodeEx extracts them in the same call sequence. Our extraction is not accurate, but it provides a way to measure API usages of samples.

3.2.2 Result

Table 3 shows the results from library code. Column "Clean" lists the number of source files without internal calls. Column "Fix" shows the number of source files whose internal calls are fully removed. Column "Error" lists the number of source files whose internal calls are not successfully removed. In this study, we analyze only the clean and fixed source files. In total, we extract fewer samples from library code than from SearchCode. Table 4 shows the number of unique covered API elements. In total, the samples from library code cover 4,209 API classes. As a comparison, Table 2, the samples from Internet code cover 1,019 API classes. Column "%" shows that in total, the samples from library code cover 38.4% of API classes. As a comparison, Table 2 shows that in total, the samples from Internet code cover only 9.3% of API classes. Here, for both sources, we count only relevant and up-to-date samples. Table 3 shows that the samples from library code are fewer than those from Internet code, as far as the popular libraries such as `cassandra`, `lucene`, and `poi` are considered. Even for the three libraries, Table 4 shows that the samples from library code cover significantly more API classes (`cassandra` from 7.2% to 50.9%; `lucene` from 21.3% to 36.1%; and `poi` from 10.4% to 55.6%). In total, the samples from library code cover much more API methods and call sequences than the samples from Internet code. The results lead to our following finding:



(a) `accumulo` (b) `cassandra`

(c) `karaf` (d) `lucene`

(e) `poi` (f) total

Fig. 3: The number of code samples per API class

Finding 2. The code samples from library code cover 4.0 times more API classes, 4.7 times more methods, and 3.0 times more call sequences than the samples from Internet code.

In summary, the samples from library code cover much more API classes, methods, and call sequences than Internet code.

3.3 RQ3. The Characteristic of Samples

3.3.1 Protocol

For each sample in Table 3, CodeEx extracts the API classes it calls. Based on the results, for each API class, we reversely count the samples that call the API class. In this way, we count how many samples each API class has. Based on the results, we draw figures to show their accumulative percentage. These figures do not present which classes are the most popular. To handle this issue, we rank API classes in the descending order of their samples, and we compare whether their top API classes overlap. To show the uniqueness of samples, we merge samples whose call sequences are identical. Here, we take a rigorous criterion, since their arguments can be different even if two sequences are identical. We draw box plots to show the distribution of call sequences, as far as their number of samples is concerned.

3.3.2 Result

Figure 3 shows the number of code samples per API class. The horizontal axes show the number of samples, and the vertical axes show the accumulative percentages. Please note that the cumulative percentages start from zero, and

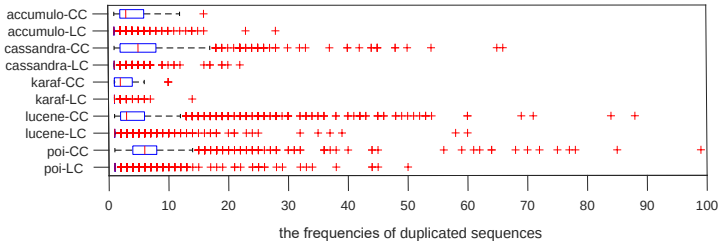


Fig. 4: The frequency of call sequences

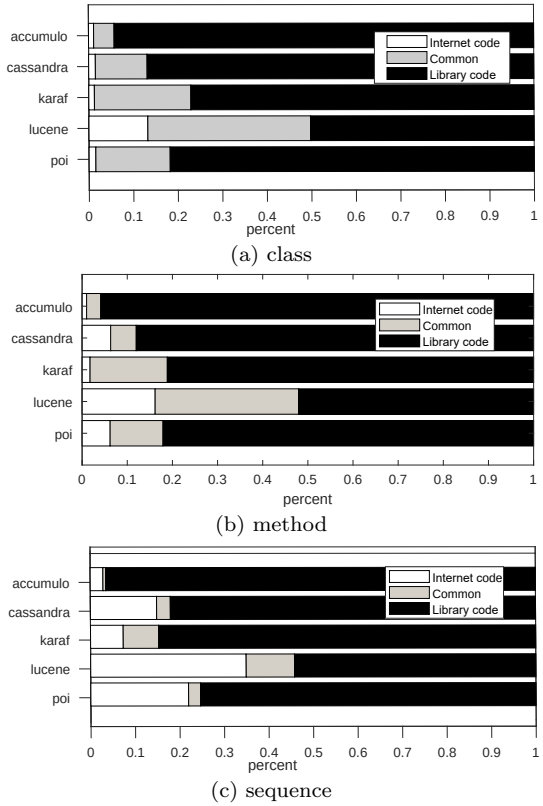
the horizontal axes are log scales. We find that the distributions of samples are consistent across projects. From both sources, a small portion of popular API classes have much more samples. Several popular API classes even have thousands of samples. Please note that when we use SearchCode to retrieve samples from Internet code, for each API class, we analyze only the top 20 pages. As a result, for those popular API classes, SearchCode can retrieve much more samples than what we collected. For about 10% of API classes, we retrieved more samples from Internet code than from library code since its repository is much larger than ours, but for the other classes, we retrieved more samples from library code. Meanwhile, 50% of API classes have fewer than ten samples. However, if we consider the areas under the curves, we find that the distributions of library code are less skewed than those of Internet code. For example, Figure 3f shows that in total, we retrieved at least one sample for 40% of API classes from library code, and the percent is only 10% when we retrieved samples from Internet code. From both sources, a few popular API classes (the top ten percent) have much more samples than the majority, but the distribution of library code is less skewed. For about 90% of API classes, samples from library code are more than those from Internet code.

Figure 4 shows the box plot of frequency. "CC" denotes the sequences of Internet code, and "LC" denotes those of library code. Here, if the frequency of a call sequence is two, it appears in two code samples. Table 4 shows that the samples from library code contain more unique call sequences than those from Internet code. For all the projects, Figure 4 shows that the median frequencies of library code are lower than those of Internet code. The observation leads to our following finding:

Finding 3. The call sequences from library code are less repetitive than those from Internet code.

Researchers proposed approaches to mine API patterns based on their frequency [11] or their repetitiveness [27]. As samples from library code are less repetitive, it is more challenging to mine API patterns from these samples, if researchers use frequencies to mine patterns. Instead, researchers can mine their usage patterns with other techniques. For example, instead of frequencies, Saied *et al.* [51] compare the source files of libraries to identify API call sets.

In summary, from both sources, popular API classes have much more samples than the majority. For about 10% API classes, Internet code contains more samples, but fewer samples than library code for other API classes. In



(a) class

(b) method

(c) sequence

Fig. 5: The overlapped API usages

addition, we find that code samples of library code are less repetitive. As a result, although it is more challenging to mine API patterns from the samples from library code, it can be more effective to recommend unique samples.

3.4 RQ4. Overlapped API Usages

3.4.1 Protocol

In this research question, we analyze to what degree API usages are overlapped. For each API, we check whether it appears in S_{lc}, S_{cc}, or $S_{lc} \cap S_{cc}$, from different levels such as classes, methods, and call sequences.

3.4.2 Result

Figure 5 shows the results: "Internet code" shows APIs that appear only in S_{cc}; "Library code" shows APIs that appear only in S_{lc}; and "common" shows APIs that appear in $S_{cc} \cap S_{lc}$. Figure 5a shows that at the granularity of API

classes, except `lucene`, S_{lc} covers almost the entire S_{cc}. Even for `lucene`, S_{lc} covers more than 70% S_{cc}. Comparing with API classes, Figure 5b shows that at the granularity of methods, S_{lc} covers fewer S_{cc}, but for all the projects, S_{lc} covers more than half of S_{cc}. However, at the granularity of call sequences, S_{lc} no longer covers most of S_{cc}. The observation leads to our finding:

Finding 4. The samples from library code cover about 90% of API classes and 80% of methods that Internet code covers, but the overlapped samples reduce to about 20%, when we consider their call sequences.

3.5 RQ5. Programming Tasks

In this research question, we conducted a controlled experiment to compare the two sources in their capability for assisting programming tasks. Although various approaches have been proposed to rank, mine, and summarize source files, we did not select these approaches for two considerations. First, the purpose of this research question is to compare the two sources, but not to compare techniques. Second, as many approaches are not designed to take API code as their inputs, their effectiveness can be significantly reduced on library code. Instead of comparing the above-mentioned approaches, we analyze their research opportunities in Section 4, according to our findings.

3.5.1 Setup

Human subjects. We invited four PhD students to attend this study. All the students are majoring in computer science, and none of them are authors of this paper. Table 5 shows their backgrounds. Rows "experience" and "Java" show their years of programming experience and years of using Java. Row "language" shows their known programming languages, ranked by their skills.

Programming tasks. We select `poi` as the subject library for two considerations. First, it is easier to prepare the environment. We do not have to install additional software for `poi`, but if we select other libraries like `cassandra`, we have to prepare the database and its tables for its tasks. Second, none of the students were familiar with `poi`. In our tasks, we must rule out impacts of the prior API knowledge, because it will introduce bias to our study. None of our students know `poi`, but they used other libraries such as `lucene`.

In total, we prepare twenty tasks. We consider the popular features of the `poi` library so that (1) the tasks reflect more common real-world development scenarios, and (2) there are more overlaps between extracted usage samples from different sources and thus we can have a basis to compare their effectiveness. Table 6 shows the tasks. As listed in "Description", our tasks cover various functionalities of `poi`. Before the study, the authors wrote the gold standards and the test cases for all the tasks. As shown in Column "LOC", it typically needs only several lines of code to implement our tasks. Our tasks do not involve complicated algorithms, because they are designed to evaluate

Table 5: The backgrounds of our students.

	student 1	student 2	student 3	student 4
experience	7	10	6	6
Java	1	7	2	5
language	C Python Java	C Java Python	C C++ Java	C C++ Java

two API learning sources. Columns "Class" and "Method" list the number of required POI classes and methods, respectively. As shown in Column "Test", a task can have more than one test case. For example, the first task sets the username and password to a record. One test case checks the username, and the other checks the password. We released our programming tasks on our website. In some programming tasks, programmers need to manipulate files, and the correctness of their outputs depends on the contents of these files. To support the replication of our study, we release these files with our tasks.

Compared treatments. This evaluation explores the impacts of two treatments. In the first treatment, the students are asked to search code samples from SearchCode. In the second treatment, the students are asked to search for code samples from a local repository. This repository contains our samples that were synthesized from the API code of `poi`. Researchers [59] report that programmers may not find the useful APIs of a programming task, but they cannot build a query without knowing a useful API. In practice, programmers usually know which library they use and search API usages for this library. As a start, to allow the students building queries, we give a useful `poi` class for each task. Indeed, each task name is ended with the `poi` class name. As no student is familiar with `poi`, to help them construct queries, they are allowed to browse the API documentation of `poi`.

Onsite setting. Before the evaluation, the first author introduced the desired functionalities of all the tasks, SearchCode, and Junit to the students. The students were told that all the programming tasks involved no complicated algorithms, but were designed to call proper `poi` APIs. The students are asked not to modify or fool the test code. The students were grouped into two teams based on their Java programming experiences. The students and tasks have differences. For example, a student can construct better queries than other students. To eliminate the differences, we asked two teams to complete half of the tasks under both treatments. In particular, this evaluation is split into two phases. In the first phase, for the top ten tasks, Team A applied the first treatment, and Team B applied the second treatment. In the second phase, for the bottom ten tasks, we switched the treatments of the two teams. In each phase, students were allowed to ask questions about the desired functionalities of tasks, and they were asked to implement the tasks independently.

We set the time limit of each phase as 90 minutes. On average, each task has only 9 minutes. In industrial developments, programmers often have to learn API usages in a short time. We set a short time limit to put more pressure, which is close to the situation of industrial developments.

Table 6: Our programming tasks.

Task	Description	LOC	Class	Method	Test
1	set the username and password to a record	2	2	3	2
2	add the data as a record	7	3	3	3
3	get the footer and header of a record stream	8	5	2	2
4	read an encrypted file	8	3	2	3
5	get the type of a file	6	1	2	1
6	get the styles of a shape	3	1	7	3
7	get the styles of a file	12	3	5	2
8	get the sound and info in a powerpoint file	8	1	4	4
9	parse the constants of the data	2	3	1	4
10	get the contents of a sheet	9	3	7	4
11	get the comment at a cell	3	3	2	2
12	analyze an ecripted word file	20	5	9	4
13	obtain the id and content of a record	5	3	5	3
14	set the password of a workbook	2	1	2	1
15	obtain the fragment names of a sheet	3	2	5	2
16	add a picture to a powerpoint file	23	3	5	2
17	get the place holder of a shape	1	1	1	1
18	get the document summary of a file	12	5	2	4
19	find the chart inside a slide	6	3	1	3
20	get the summary of a word file	21	6	4	4

3.5.2 Result

Table 7 shows the results. In Column "Result", $\surd$ denotes that the second treatment is better than the first treatment; $\times$ denotes that the first treatment is better than the second treatment; and - denotes that there are no differences. In four tasks, the students achieved better results with samples from library code than those with Internet code. We interviewed the students. Their feedbacks are as follows:

1. The students failed to retrieve useful samples for some tasks, although SearchCode has a huge repository. For example, in the 5th task, the students used `FileMagic` to query both tools. SearchCode retrieved 17 samples, but none calls `poi` APIs. Meanwhile, from library code, the students retrieved 51 samples for this API. As a result, a student learns how to complete this task in the second setting.

2. The students retrieved useful samples for other tasks, but they are not ranked at the top. For example, in the 8th task, the students used `HSLFSlideShow` to query SearchCode. It retrieved three pages of code samples, but the useful ones were not ranked on the first page.

The results lead to our finding:

Finding 5. In tasks where both sources can provide API usage samples, there is no significant difference between complete tasks.

Researchers have proposed various approaches to retrieve useful code samples from code repositories (see Section 5). The purpose of our study is to compare two sources of API usages, but not those recommendation approaches (see Sections 4 and 5 for more discussions). As a result, in our study, we provide

Table 7: The results from the 4 students.

Task	Team A		Team B		Result
	student 1	student 2	student 3	student 4	
1	1	2	2	2	√
2	2	1	0	0	×
3	0	0	0	0	—
4	0	0	0	0	—
5	0	0	0	1	—
6	0	0	0	0	—
7	0	0	0	0	—
8	0	0	0	1	√
9	0	0	0	0	—
10	0	0	0	0	—
11	2	2	0	0	√
12	0	0	0	0	—
13	0	0	0	0	—
14	0	0	0	0	—
15	0	0	0	0	—
16	0	0	0	0	—
17	1	0	0	0	√
18	0	0	0	0	—
19	0	0	0	0	—
20	0	0	0	0	—

The cells with the grey background show the passing test cases of the first treatment (Internet code), and the cells with the white background show the results of the second treatment (library code).

only the state-of-the-practice tool support. In particular, when the students search API documents and library code, they use "Find in Files" that is provided by text editors. Although SearchCode has a textual interface, it is not designed for natural language queries. For example, we use "poi set the given user and password" as the keyword to search SearchCode, but from the first page, we find no code samples that use `poi`. To retrieve useful code samples from both sources, programmers must construct effective queries by themselves, and identify useful API names. The students tried many queries, but seldom successfully identified all the useful API names for our programming tasks. As there were so many, we did not record all the queries.

As shown in Table 6, 5 out of our 20 tasks involve only single API classes. API documents can provide more useful information for the 5 tasks (5, 6, 8, 14, and 17), but Table 7 shows that the students resolved only the 5th task. We find two possible reasons. First, API documents are organized in a concise format, but can be unfriendly to readers. For example, the 6th task involves the following call chain:

```
1 public void getcolors(XSLFAutoShape as) {
2  fillStyle = as.getFillStyle().getPaint();...
3 }
```

The API document of `XSLFAutoShape` lists only two irrelevant methods. As the `getFillStyle()` method is declared by a superclass, `XSLFSimpleShape`, the students did not notice this method. Second, API documents can contain some

specialized vocabulary. For example, the 5th task involves the `FileMagic` class. The API document of the `valueOf()` method is as follows:

```
1 Get the file magic of the supplied File
```

Only a student understands that the file magic refers to the type of a file. As a result, only this student successfully completed this task.

3.6 Threat to Validity

The internal threat to validity includes the internal techniques of CodeEx. For library code, as CodeEx aggressively removes API calls, we can underestimate covered APIs from library code. Most code samples from code search engines are not compilable, since they are partial and do not ship with their dependencies. As a result, CodeEx cannot remove their internal usages by removing compilation errors. As the internal usages from Internet code are not removed, internal calls in Internet code can be misleading in our human study, and we can overestimate covered APIs from Internet code. Still, the impacts shall be minor, since Table 3 shows that about 80% of library source files do not have any internal calls and most Internet samples are client code. As CodeEx extracts API usages through static analysis, it ignores dynamic calls [17], and this limitation reduces covered API usages from both sides. It is difficult to extract such usages from Internet code, since it is often infeasible to execute such code due to compilation errors. Nevertheless, we did not conclude that library code is the replacement of Internet code or vice versa. Instead, our study shows that the two sources complement each other. The internal threat to validity also includes the programming skills of our students. Our students are not professional programmers, and their programming skills are insufficient to unleash the potential of either tool. As a result, 14 tasks are not accomplished in either treatment. The threats to external validity include our limited subjects, students, and tasks. As human studies are too expensive, we can only invite four students and prepare limited tasks. However, the trend of our study is already clear, and the trend may not change, if the scale is enlarged. Our quantity studies show that libraries contain richer API usages than the samples from code search engines, but our human study shows that richer API usages alone are insufficient to guarantee better assistance results. We have released all the programming tasks on our project website. Other researchers can replicate our study to reduce the threat. The threats to external validity also include our selected code search engine. Although SearchCode is popular, other code search engines (*e.g.*, GitHub) can achieve better results. This thread can be reduced by introducing more code search engines.

4 Benefit and Challenge

In this Section, we discuss the challenges and benefits:

Benefit 1. Libraries provide API samples for those new and less popular APIs. Our study explains why programmers often cannot find useful API samples, since Finding 1 shows that only 9.3% API classes have up-to-date code samples on average. Finding 2 shows that the samples from library code cover four times more API classes, which are useful to learn API usages, especially for those new and less popular APIs.

Benefit 2. Different sources complement each other, and are beneficial to various approaches. Researchers have proposed various approaches that recommend samples from local [75,59,28] and Internet [42,13,31,46] repositories. Furthermore, some approaches can mine and recommend API patterns [75,53,52] from such repositories. Besides Internet and library code, test code [77,23,25], StackOverflow [67,69,68], API tutorials [63,50,57], the past code in commit histories [74], and API documents [72] also illustrate API usages. Zhong *et al.* [71] show that library code and its documents often define different API usages. Finding 2 shows that libraries cover more API usages than Internet repositories, and Finding 4 shows that API usages from libraries and Internet repositories complement each other. All the above studies and our findings show that the API usages from multiple sources complement each other, and some early explorations already use multiple sources (*e.g.*, Zeng *et al.* [66]). Furthermore, Tung *et al.* [61] propose an approach to recommend libraries for a given project. Their approach is useful, if users do not know which libraries should be included in their scenarios.

Challenge 1. It is challenging to trim down source files to illustrate the usages of specific APIs. Compared to samples from tutorials or StackOverflow, it is more challenging for programmers to learn API usages from source files, since these files present many irrelevant implementation details. A useful code sample shall be concise [18], complete [30], and easy to understand [65], but most code samples are not written to illustrate API usages. The prior approaches [18,30] can derive better code samples. After some extensions, the above approaches can also improve the code quality of API samples from library code.

Challenge 2. It is challenging to improve the quality of code samples. As the source files from code search engines and local libraries are not written to illustrate API usages, API usages in these files are often incomplete and scattered in multiple methods. In our study, we remove obsolete usages from code search engines and internal calls from library code. Although our treatments ensure the fairness of our comparison, there are research opportunities to improve the quality of code samples. For example, it can be feasible to replace internal calls with corresponding APIs. As another example, although the API usages from individual source files are incomplete, Buse and Weimer [18] infer API tutorials from multiple source files. The above directions can improve the code quality of source files as API examples, so programmers can better learn API usages.

5 Related Work

Our study is related to the following topics.

Empirical studies on API usages. Researchers have conducted various empirical studies on API usages. Linares-Vásquez *et al.* [35] analyze the API patterns that are related to energy usages. Zhong and Mei [70] analyze API usages to learn their suitable formats of specifications. McDonnell *et al.* [41] analyze API adoptions in Android applications. Piccioni *et al.* [47] analyze the usability of APIs. Brito *et al.* [16] analyze the replacements of deprecated APIs. Monperrus *et al.* [44] analyze the directives in API documents. The prior studies analyze API usages in client code, and we are the first to analyze API usages inside library code. It can be interesting to replicate their studies on library code, and the replication can lead to insightful results.

Code search engines. Researchers have proposed various code search engines [13,42,19,48,34,38]. The above search engines provide richer interfaces to query code samples. For example, Reiss [48] allows searching through a user-defined template, and McMillan *et al.* [42] support searching through a given example. As research tools, it is difficult to build a repository as large as SearchCode has. As a result, some researchers build their tools on commercial code search engines. For example, Asyrof *et al.* [12] improve the GitHub code search with type matching, and Thummalapenta and Xie [60] improve the Google code search (a retired code search engine) to collect code samples whose input and output types are as expected. As the repository of SearchCode is larger than most of the above engines, we are unlikely to underestimate APIs that are covered by Internet code, but with these tools, our students can complete more tasks.

Code recommendation. To retrieve better samples, researchers proposed various query interfaces. For example, Tansalarak and Claypool [59] allow querying with a pair of input and output types. It is difficult for novice programmers to construct queries. To assist such programmers, researchers explored recommending code samples by the programming contexts under development [28,75]. When programmers are writing code, their programming contexts can be incomplete. Zhou *et al.* [76] use a language model to synthesize code, and search for the clones of synthesized code. Although synthesized code can contain errors, they served as the programming contexts to search for real code. Ghafari and Moradi [24] implement a benchmark to evaluate code recommendation tools. Nguyen *et al.* [45] recommend code samples for handling exceptions. Besides code repositories, researchers also explored other sources like StackOverflow threads [31]. Given code repositories as inputs, researchers further mine various models to synthesize code [15,62,21] or code hints (*e.g.*, next APIs [36]). Based on our results, library code is a useful source for the above approaches, which can unleash the potential of library code.

6 Conclusion

Programmers mainly use Internet code from code search engines to learn API usages. Although code search engines provide Internet-scale repositories, programmers still complain that it is difficult to learn API usages, especially for those new and unpopular ones. As code search engines already have millions of projects in their repositories, it is less useful to add more projects. Instead, researchers have illustrated that library code also contains interesting and concise API usages. Although library code is beneficial, it is challenging to analyze its API usages, since library code can contain internal usages. As a result, analyzing library code directly overestimates the API usages from library code. To handle the problem, in this paper, we implement a support tool, and conduct the first empirical studies on self-API usages inside five libraries. We summarize our results in six findings, and provide our insights on the benefits and challenges. As for its benefits, library code contains much richer API usages than Internet code, but as challenges, it needs more advanced techniques to unleash the potential of API usages inside libraries.

Acknowledgments

We appreciate reviewers for their insightful comments. Hao Zhong is sponsored by the National Nature Science Foundation of China No. 62232003 and 62272295. Xiaoyin Wang is supported in part by NSF Grant CCF-1846467.

References

1. accumulo. `https://accumulo.apache.org` (2019)
2. cassandra. `http://cassandra.apache.org` (2019)
3. Guice. `https://searchcode.com/api/` (2019)
4. JDT. `http://www.eclipse.org/jdt/` (2019)
5. karaf. `https://karaf.apache.org` (2019)
6. lucene. `https://lucene.apache.org` (2019)
7. poi. `https://poi.apache.org` (2019)
8. The searchcode engine. `https://searchcode.com/` (2019)
9. cassandra archive. `http://archive.apache.org/dist/cassandra` (2020)
10. Tha API documents of accumulo 1.9. `https://accumulo.apache.org/1.9/apidocs/` (2020)
11. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proc. 29th POPL, pp. 4–16 (2002)
12. Asyrofi, M.H., Thung, F., Lo, D., Jiang, L.: Ausearch: Accurate API usage search in github repositories with type resolution. In: Proc. SANER, pp. 637–641 (2020)
13. Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. In: Companion to Proc. OOPSLA, pp. 681–682 (2006)
14. Bian, P., Liang, B., Shi, W., Huang, J., Cai, Y.: Nar-miner: discovering negative association rules from code for bug detection. In: Proc. ESEC/FSE, pp. 411–422 (2018)
15. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proc. PLDI, pp. 467–481 (2017)

16. Brito, G., Hora, A., Valente, M.T., Robbes, R.: On the use of replacement messages in api deprecation: An empirical study. Journal of Systems and Software **137**, 306–321 (2018)
17. Bruce, B.R., Zhang, T., Arora, J., Xu, G.H., Kim, M.: Jshrink: In-depth investigation into debloating modern java applications. In: Proc. ESEC/FSE, pp. 135–146 (2020)
18. Buse, R.P., Weimer, W.: Synthesizing api usage examples. In: Proc. ICSE, pp. 782–792 (2012)
19. Chatterjee, S., Juvekar, S., Sen, K.: Sniff: A search engine for java using free-form queries. In: Proc. FASE, pp. 385–400 (2009)
20. Dagenais, B., Hendren, L.J.: Enabling static analysis for partial Java programs. In: Proc. OOPSLA, pp. 313–328 (2008)
21. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: Proc. PLDI, pp. 420–435 (2018)
22. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: Proc. ESEC/FSE, pp. 339–349 (2008)
23. Ghafari, M., Ghezzi, C., Mocci, A., Tamburrelli, G.: Mining unit tests for code recommendation. In: Proc. ICPC, pp. 142–145 (2014)
24. Ghafari, M., Moradi, H.: A framework for classifying and comparing source code recommendation systems. In: Proc. SANER, pp. 555–556 (2017)
25. Ghafari, M., Rubinov, K., Pourhashem K, M.M.: Mining unit test cases to synthesize api usage examples. Journal of software: evolution and process **29**(12), e1841 (2017)
26. Hassan, F., Wang, X.: HireBuild: An automatic approach to history-driven repair of build scripts. In: Proc. ICSE, pp. 1078–1089 (2018)
27. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proc. 34th ICSE, pp. 837–847 (2012)
28. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: Proc. 27th ICSE, pp. 117–125 (2005)
29. Kawrykow, D., Robillard, M.P.: Improving API usage through automatic detection of redundant code. In: Proc. ASE, pp. 111–122 (2009)
30. Keivanloo, I., Rilling, J., Zou, Y.: Spotting working code examples. In: Proc. ICSE, pp. 664–675 (2014)
31. Kim, K., Kim, D., Bissyandé, T.F., Choi, E., Li, L., Klein, J., Traon, Y.L.: Facoy: a code-to-code search engine. In: Proc. ICSE, pp. 946–957 (2018)
32. Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies? Empirical Software Engineering **23**(1), 384–417 (2018)
33. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: Proc. ASE, pp. 81–92 (2015)
34. Lemos, O.A.L., Bajracharya, S.K., Ossher, J., Morla, R.S., Masiero, P.C., Baldi, P., Lopes, C.V.: Codegenie: using test-cases to search and reuse source code. In: Proc. ASE, pp. 525–526 (2007)
35. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., Poshyvanyk, D.: Mining energy-greedy api usage patterns in android apps: an empirical study. In: Proc. MSR, pp. 2–11 (2014)
36. Liu, X., Huang, L., Ng, V.: Effective api recommendation without historical software repositories. In: Proc. ASE, pp. 282–292 (2018)
37. Lo, D., Khoo, S.C.: Smartic: Towards building an accurate, robust and scalable specification miner. In: Proc. ESEC/FSE, pp. 265–275 (2006)
38. Lv, F., Zhang, H., Lou, J.g., Wang, S., Zhang, D., Zhao, J.: Codehow: Effective code search based on api understanding and extended boolean model (e). In: Proc. ASE, pp. 260–270 (2015)
39. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Proc. PLDI, pp. 48–61 (2005)
40. Maoz, S., Ringert, J.O.: GR(1) synthesis for LTL specification patterns. In: Proc. ESEC/FSE, pp. 96–106 (2015)
41. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the android ecosystem. In: Proc. ICSM, pp. 70–79 (2013)
42. McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., Xie, Q.: Exemplar: A source code search engine for finding highly relevant applications. IEEE Transactions on Software Engineering **38**(5), 1069–1087 (2011)

43. Michail, A.: Data mining library reuse patterns using generalized association rules. In: Proc. ICSE, pp. 167–176 (2000)
44. Monperrus, M., Eichberg, M., Tekes, E., Mezini, M.: What should developers be aware of? an empirical study on the directives of api documentation. Empirical Software Engineering **17**(6), 703–737 (2012)
45. Nguyen, T., Vu, P., Nguyen, T.: Code recommendation for exception handling. In: Proc. ESEC/FSE, pp. 1027–1038 (2020)
46. Niu, H., Keivanloo, I., Zou, Y.: Learning to rank code examples for code search engines. Empirical Software Engineering **22**(1), 259–291 (2017)
47. Piccioni, M., Furia, C.A., Meyer, B.: An empirical study of API usability. In: Proc. ESEM, pp. 5–14 (2013)
48. Reiss, S.P.: Semantics-based code search. In: Proc. ICSE, pp. 243–253 (2009)
49. Robillard, M.P., DeLine, R.: A field study of API learning obstacles. Empirical Software Engineering **16**(6), 703–732 (2011)
50. Sadowski, C., Stolee, K.T., Elbaum, S.: How developers search for code: a case study. In: Proc. ESEC/FSE, pp. 191–201 (2015)
51. Saied, M.A., Abdeen, H., Benomar, O., Sahraoui, H.: Could we infer unordered api usage patterns only using the library source code? In: Proc. ICPC, pp. 71–81 (2015)
52. Saied, M.A., Ouni, A., Sahraoui, H., Kula, R.G., Inoue, K., Lo, D.: Improving reusability of software libraries through usage pattern mining. Journal of Systems and Software **145**, 164–179 (2018)
53. Saied, M.A., Raelijohn, E., Batot, E., Famelis, M., Sahraoui, H.: Towards assisting developers in api usage by automated recovery of complex temporal patterns. Information and Software Technology **119**, 106213 (2020)
54. Sawant, A.A., Robbes, R., Bacchelli, A.: On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In: Proc. ICSME, pp. 400–410 (2016)
55. Scaffidi, C.: Why are APIs difficult to learn and use? Crossroads **12**(4), 4–4 (2005)
56. Sim, S.E., Umarji, M., Ratanotayanon, S., Lopes, C.V.: How well do search engines support code retrieval on the web? ACM Transactions on Software Engineering and Methodology **21**(1), 1–25 (2011)
57. Stolee, K.T., Elbaum, S., Dobos, D.: Solving the search for source code. ACM Transactions on Software Engineering and Methodology **23**(3), 1–45 (2014)
58. Sven, A., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M.: Investigating next steps in static API-misuse detection. In: Proc. MSR, pp. 265–275 (2019)
59. Tansalarak, N., Claypool, K.: XSnippet: Mining for sample code. Proc. 21st OOPSLA pp. 413–430 (2006)
60. Thummalapenta, S., Xie, T.: PARSEWeb: a programmer assistant for reusing open source code on the web. In: Proc. 22nd ASE, pp. 204–213 (2007)
61. Thung, F., Lo, D., Lawall, J.: Automated library recommendation. In: Proc. WCRE, pp. 182–191 (2013)
62. Wang, Y., Dong, J., Shah, R., Dillig, I.: Synthesizing database applications for schema refactoring. In: Proc. PLDI, p. to appear (2019)
63. Xia, X., Bao, L., Lo, D., Kochhar, P.S., Hassan, A.E., Xing, Z.: What do developers search for on the web? Empirical Software Engineering **22**(6), 3149–3185 (2017)
64. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proc. 28th ICSE, pp. 282–291 (2006)
65. Ying, A.T., Robillard, M.P.: Selection and presentation practices for code example summarization. In: Proc. ESEC/FSE, pp. 460–471 (2014)
66. Zeng, H., Chen, J., Shen, B., Zhong, H.: Mining API constraints from library and client to detect API misuses. In: Proc. APSEC, pp. 161–170 (2021)
67. Zhang, H., Wang, S., Chen, T.H.P., Zou, Y., Hassan, A.E.: An empirical study of obsolete answers on stack overflow. IEEE Transactions on Software Engineering (2019)
68. Zhang, N., Zou, Y., Xia, X., Huang, Q., Lo, D., Li, S.: Web APIs: Features, issues, and expectations–a large-scale empirical study of Web APIs from two publicly accessible registries using stack overflow and a user survey. IEEE Transactions on Software Engineering (2022)
69. Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M.: Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In: Proc. ICSE, pp. 886–896 (2018)

70. Zhong, H., Mei, H.: An empirical study on API usages. IEEE Transactions on Software Engineering **45**(4), 319–334 (2019)
71. Zhong, H., Meng, N., Li, Z., Jia, L.: An empirical study on API parameter rules. In: Proc. ICSE, pp. 899–911 (2020)
72. Zhong, H., Su, Z.: Detecting API documentation errors. In: Proc. OOPSLA, pp. 803–816 (2013)
73. Zhong, H., Wang, X.: Boosting complete-code tools for partial program. In: Proc. ASE, pp. 671–681 (2017)
74. Zhong, H., Wang, X., Mei, H.: Inferring bug signatures to detect real bugs. IEEE Transactions on Software Engineering **48**(2), 571–584 (2022)
75. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending API usage patterns. In: Proc. 23rd ECOOP, pp. 318–343 (2009)
76. Zhou, S., Shen, B., Zhong, H.: Lancer: Your code tell me what you need. In: Proc. ASE, pp. 1202–1205 (2019)
77. Zhu, Z., Zou, Y., Xie, B., Jin, Y., Lin, Z., Zhang, L.: Mining API usage examples from test code. In: Proc. ICSME, pp. 301–310 (2014)