The Journal of Systems & Software 169 (2020) 110728

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In practice

# From API to NLI: A new interface for library reuse

Qi Shen [a,b], Shijun Wu [a,b], Yanzhen Zou [a,b,*], Zixiao Zhu [c], Bing Xie [a,b]

[a] Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China
[b] School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
[c] IBM Research, Beijing 100191, China

ARTICLE INFO

*Article history:*
Received 2 November 2019
Received in revised form 26 June 2020
Accepted 7 July 2020
Available online 9 July 2020

*Keywords:*
Library reuse
Code pattern
Program synthesis

ABSTRACT

Developers frequently reuse APIs from existing libraries to implement certain functionality. However, learning APIs is difficult due to their large scale and complexity. In this paper, we design an abstract framework NLI2Code to ease the reuse process. Under the framework, users can reuse library functionalities with a high-level, automatically-generated NLI (Natural Language Interface) instead of the detailed API elements. The framework consists of three components: a *functional feature extractor* to summarize the frequently-used library functions in natural language form, a *code pattern miner* to give a code template for each functional feature, and a *synthesizer* to complete code patterns into well-typed snippets. From the perspective of a user, a reuse task under NLI2Code starts from choosing a functional feature and our framework will guide the user to synthesize the desired solution. We instantiated the framework as a tool to reuse Java libraries. The evaluation shows our tool can generate a high-quality natural language interface and save half of the coding time for newcomers to solve real-world programming tasks.



## 1. Introduction

To implement certain functionality, developers often reuse existing libraries with the corresponding APIs. Yet discovering the correct subset of the APIs is a major obstacle for the API users (Robillard and DeLine, 2011). The obstacle not only comes from the large size of APIs, furthermore, a real-world programming task usually requires the cooperation of multiple APIs, and each API invocation should follow strict specifications. For example, for a simple functionality like "*set color for an Excel cell*", the desired API usage sequence using `apache-poi` is as follows:

*Workbook*.*createCellStyle*();
*CellStyle*.*setFillBackgroundColor*(*short*);
*CellStyle*.*setFillForegroundColor*(*short*);
*CellStyle*.*setFillPattern*(*FillPatternType*);
*Cell*.*setCellStyle*(*CellStyle*);

To address the issue, we promoted the concept of NLI (Natural Language Interface) for library reuse (Shen et al., 2019). With NLI, users reuse library functionalities with high-level natural language descriptions instead of directly manipulating the detailed APIs. Fig. 1 summarizes the key steps of how Alice, a Java programmer, reuses the library `apache-poi` with NLI. As Fig. 1(a) shows, Alice starts from selecting the desired functionality from a list of natural language descriptions, which is *set cell color* in this case. After the selection, the functionality is mapped to its corresponding implementation, which is a built-in code template in NLI. As Fig. 1(b) shows, Alice needs to provide three parameters (*i.e.* specific background color, foreground color and fill pattern) to fill the template. Each parameter to provide is annotated with an example expression in gray font, which is recommended by a synthesizer. In fact, there are more than three missing parameters in the code template, the synthesizer has automatically created trivial ones from the current context (*e.g.* creating a *Workbook* object with the constructor). After Alice fills the parameters, a well-typed code snippet is synthesized and inserted into the editor (as Fig. 1(c) shows), which perfectly solves Alice's task.

For the library reuse problem, we highlight the benefits of NLI from two aspects. The first benefit is the query composition. If the developer is not familiar with the library, it could be difficult to compose a high-quality query which accurately describes the desired functionality. For example, Table 1 displays a post[1] from Stack Overflow. The post title mistakenly mentioned the concept "*background color*", while the accepted answer shows the user actually desired "*foreground color*". In NLI, we summarize the library functionalities into functional features. We conjecture that, compared to composing free-form queries, the mechanism

* Correspondence to: Peking University, No.5, Yiheyuan Road, Beijing 100871, China.
*E-mail address:* zouyz@pku.edu.cn (Y. Zou).

[1] https://stackoverflow.com/questions/53052931.

https://doi.org/10.1016/j.jss.2020.110728
0164-1212/

```
public class Main {
  public static void main(string[] args) {
    Workbook workbook = new HSSFWorkbook();
    Cell cell = workbook.createSheet().createRow(0).createCell(0);
    set cell
  }
}
```

```
set cell alignment   (Statement in jetbrains.mps.baseLanguage)
set cell borders     (Statement in jetbrains.mps.baseLanguage)
set cell color       (Statement in jetbrains.mps.baseLanguage)
set cell font        (Statement in jetbrains.mps.baseLanguage)
```

(a) User types or selects the desired functional feature

```
public class Main {
  public static void main(string[] args) {
    Workbook workbook = new HSSFWorkbook();
    Cell cell = workbook.createSheet().createRow(0).createCell(0);
    set color for cell cell {
      background color: <IndexedColors.RED.getIndex()>
      foreground color: <IndexedColors.RED.getIndex()>
      pattern: <FillPatternType.BIG_SPOTS>
    }
  }
}
```

(b) The code pattern exposes three parameters for user

```
public class Main {
  public static void main(String[] args) {
    Workbook wb = new HSSFWorkbook();
    Cell cell = wb.createSheet().createRow( i: 0).createCell( i: 0);
    CellStyle style = wb.createCellStyle();
    style.setFillBackgroundColor(IndexedColors.RED.getIndex());
    style.setFillForegroundColor(IndexedColors.RED.getIndex());
    style.setFillPattern(FillPatternType.BIG_SPOTS);
    cell.setCellStyle(style);
  }
}
```

(c) The synthesized code snippet for the functional feature

**Fig. 1.** Application of NLI for reusing `apache-poi`.

**Table 1**
An example post from Stack Overflow.

| |
|---|
| **Title: Apache-POI : How to set background color of a cell when creating spreadsheet?** |
| **Question:** In Apache POI 4.0, I want to set an Excel cell background color. But all I get are black cells. I have tried many things, but result is always the same. How can I set the background color of an Excel cell in Apache POI 4.0 ? |
| **Answer:** Try to use below code for background style: setFillForegroundColor(IndexedColors.YELLOW.getIndex()); |

of selecting functional features is easier and can make users more confident about the results. The second benefit is code quality. An illustrative code example can help developers quickly understand how to implement certain functionality. However, many online code examples are only intended to express the main idea of a solution instead of being reused as-it. Previous studies (Treude and Robillard, 2017; Zhang et al., 2018) show that online code examples are often not self-explanatory and may have quality problems such as incorrect order of API calls. As Table 1 shows, the code snippet in the accepted answer contains only one API, which is not a complete solution for the task. In NLI, we mine code patterns by exploring more usage examples of the API. Our hypothesis is that unveiling how APIs are used in similar program contexts could improve the code quality.

To construct and use NLI, we designed an abstract framework NLI2Code, which consists of three components: a functional feature extractor, a code pattern miner, and a synthesizer. *Functional features* are natural language descriptions of the library functionalities. In this paper, we instantiated the extractor by

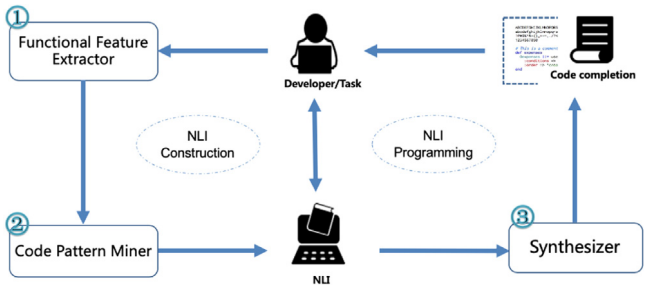


**Fig. 2.** NLI2Code framework.

mining Stack Overflow since a previous survey shows it is the first option for most developers to search for programming solutions (Ragkhitwetsagul et al., 2018). In the second component, we try to match each functional feature with a *code pattern*, which is a code template mined from multiple implementations of the feature. As code patterns usually lack customized information such as local parameters, a *synthesizer* is supposed to complete them into compilable snippets. The missing parameters could be synthesized from the current programming context or provided by the user. Finally, NLI2Code combines the three components and generates well-typed code snippets for users.

Around the central concept NLI, the main contributions of this paper are:

- an algorithm to extract verb phrases describing library functionalities from Stack Overflow.
- an approach to mine code patterns, with a self-designed intermediate representation for Java to eliminate coding style differences.
- an instantiation of NLI2Code to reuse Java libraries, with evaluation on real-world tasks to prove the feasibility of the framework.

The remainder of the paper is organized as follows. Section 2 demonstrates the abstract framework NLI2Code. Sections 3–5 explain our implementation of the framework, which is available from our online artifacts.[2] In Section 6, we conduct several experiments to check the accuracy of our algorithms and apply a controlled experiment to explain how NLI2Code works in real-world development. Section 7 introduces the related work. Section 8 briefly summarizes this paper.

## 2. Framework

As Fig. 2 shows, the NLI2Code framework consists of three components. In the offline part, we construct a natural language interface as pairs of functional features and code patterns. In the online part, the user solves tasks by selecting functional features and our synthesizer completes the corresponding code patterns into well-typed code snippets. In the rest of this section, we will discuss the three components separately.

### 2.1. Functional feature extractor

Extracting functional features is the first step in our framework. A functional feature is a brief description of certain library functionality in verb phrase form. Nowadays, libraries typically provide multiple platforms for developers and users to communicate, such as mailing lists, issue tracker system, and posts from online forums like Stack Overflow. These communication records

[2] https://github.com/nli2code/jss-artifact.

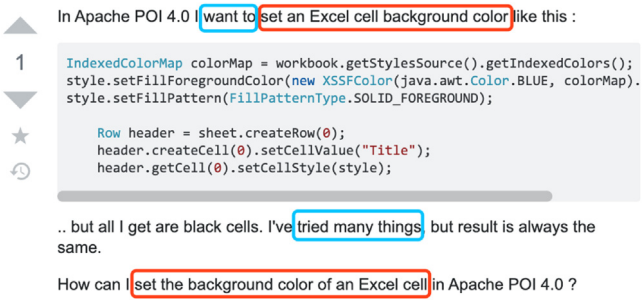


**Fig. 3.** Functional features in Stack Overflow.

are the natural corpus to extract functional features because they contain rich information about how libraries are used. In our framework, all verb phrases from the discussions are considered as candidate functional features. We issue two challenges to get usable functional features:

- Noises. As Fig. 3 shows, phrases like *want to* and *try many things* are unrelated to library functionalities and have little semantic information. Such phrases should be pruned off.
- Diversity. Functionalities could be expressed in different ways. *e.g. set an Excel cell color* and *set the color of an Excel cell* in Fig. 3. Furthermore, users could use different words which makes the phrases lexically different. *e.g. change the cell color*. Such phrases with the same semantic information need to be clustered and normalized. Otherwise, the generated natural language interface will be verbose and repetitive.

In this work, we applied a filtering pipeline to remove noise phrases, considering stop words, the structure, and the context of the phrases. To cluster similar phrases, we designed a normal form to extract the core action and objects in verb phrases. After normalization, phrases with the same content or merely lexically different are merged. Here we define two important properties for the extracted functional features:

- Accurate: Each functional feature should clearly correspond to certain library functionality.
- Complete: The set of all functional features should cover the library functionalities as much as possible.

### 2.2. Code pattern miner

Functional features organize library functionalities in a list of verb phrases. Although many posts from the user forums naturally provide code examples, these examples usually cannot be reused as-it. In fact, most code examples are only intended to describe the main idea of a solution to the original question, which could be difficult to be understood by others (Treude and Robillard, 2017). Furthermore, a recent analysis shows that online code examples usually have quality problems such as missing control constructs and incorrect order of API calls (Zhang et al., 2018). Another analysis on 914,974 Java code snippets from Stack Overflow shows that only 3.89% of them are parsable (Yang et al., 2016).

A practical way to improve the quality of code examples is to detect similar API usage in a larger codebase. A code pattern is a code template describing that in a certain usage scenario, some API elements are frequently called together. Compared to a single code example in the original post, a code pattern exploits the commonalities among similar programs, which reduces the risk of unknown consequences. Moreover, code patterns naturally hint users which part of the code to modify because they leave variations among the programs as unfilled parts. Common variations include hard-coded strings and magic numbers. A common procedure for code pattern mining is as follows:

- construct a code corpus
- abstract code into a certain data structure (*e.g.*, call sequence, abstract syntax tree, data flow graph)
- apply the corresponding frequent pattern mining algorithm on the corpus and transform the frequent items back to code

### 2.3. Synthesizer

Code patterns are incomplete because they usually miss local information. Existing IDEs (Integrated Development Environment) usually provide a simple code completion feature. However, such completion typically only considers one step of computation, which means that the recommendation result is a single variable or method. In fact, a missing parameter may require a method chain to get the desired result. Furthermore, each method in the chain may require new parameters to synthesize. These efforts suggest a general direction for the synthesizer in NLI2Code: given a programming context $\Gamma$ and the desired type $\tau$, synthesize the entire type-correct expression with type $\tau$ from the context. Formally, find expression $e$ such that $\Gamma \vdash e : \tau$.

We conclude two solutions for the synthesizer. The first one is called the type-directed search (Perelman et al., 2012; Gvero et al., 2013), which enumerates all possible expressions with the desired type. Since the searching space is usually large, heuristic functions are often used to guide the search process. The second solution recommends expressions according to the statistical analysis of a large code corpus. To synthesize the desired expression, users can benefit if they are recommended with expressions frequently used under a similar context.

To understand the potential and feasibility of our framework, we instantiated it as a tool NLI4J to reuse Java libraries. In the following three sections, we will separately introduce our implementation of the three components.

## 3. Extracting functional features

As the first component of NLI2Code, we need a list of functional features to summarize frequently-used library functionalities. Recall that a functional feature is defined as a brief description of certain functionality in verb phrase form. Given a library, our extractor takes Stack Overflow threads as input and outputs the functional feature list.

Fig. 4 shows the workflow of our approach to extract functional features. We firstly extract verb phrases from Stack Overflow threads by leveraging the syntax parsing techniques. Then, a set of heuristic rules is used to filter out unlikely phrases. Considering that the same functionality can be expressed in different ways, we propose a normalized functional feature representation grammar to ensure the correct clustering of phrases. At last, a frequent subgraph mining algorithm gSpan (Yan and Han, 2002) is applied to mine functional features from the clustered phrases.

### 3.1. Candidate functional features

Our data source is the Q&A threads from Stack Overflow containing the specific tags, such as *"apache-poi"* for the POI project. According to the definition, we extract all verb phrases as candidate functional features. Similar to the state-of-the-art works, we use Stanford NLP toolkits (Manning et al., 2014) to extract verb phrases from the raw data. A big problem for applying NLP tools to software documentation is that there are many code-like terms, which are error-prone in the POS (Part-of-Speech) tagging

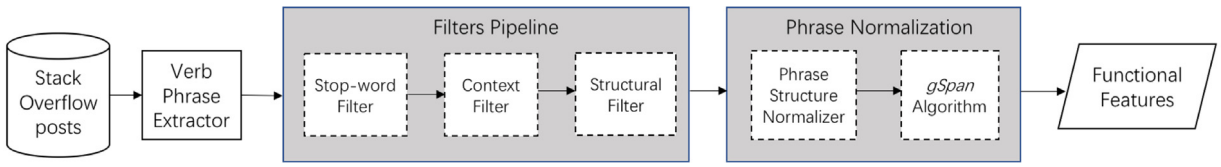


**Fig. 4.** The process of extracting functional features.

and might cause failure in the syntax tree parsing. Thus, we replace the code-like terms with special placeholders to ensure the correct POS tagging. For reproducibility, we briefly explain how we recognize code-like terms here. Stack Overflow threads usually label code fragments with consecutive ⟨*pre*⟩⟨*codea*⟩ tags or ⟨*code*⟩ tag for the inline code elements. For those code-like terms that are not annotated with HTML tags, we employ a set of regular expressions, which is provided by Treude et al. (2015) to identify them from the natural language content.

After the preprocessing, we split the natural language text into sentences and feed each sentence to the Stanford NLP toolkit. The toolkit returns a tree-structured parsing result. Fig. 5 displays the parsing tree of a long sentence, which contains seven verb phrases (subtrees tagged with VP). All the verb phrases from the parsing tree are extracted and form the initial candidate functional features, which will be filtered by a filtering pipeline. The pipeline filters out phrases not related to library functions, for example, in the sentence from Fig. 5, only one phrase out of seven subtrees is valuable, which is the seventh phrase *set up the print area for the excel file*.

### 3.2. Filtering pipeline

The filtering pipeline consists of three rule-based phrase filters. If a phrase matches the rule of a filter, one piece of evidence will be added to the phrase. A piece of evidence might be counted as one vote up or one vote down or veto to accept the phrase. Then we collect all the evidence added to a phrase and count the vote. Intuitionally, we remove a phrase when the upvotes are less than the downvotes.

The first filter is based on a handcrafting stop word list. We downvote three types of verb phrases because they are not likely to appear in a meaningful functional feature:

- Special grammatical ingredients such as auxiliary verbs (*e.g.*, be, do, have), modal verbs and pronouns usually do not have actual meanings.
- Q&A special words. The sentences from Stack Overflow often contain trivial words for describing the questioners' requirements (*e.g.*, ask, try, need).
- Programming special terms. Some programming terms, keywords in programs, or development special words are usually not part of valid functional features. (*e.g.*, extend, return and stack trace).

The second filter judges the phrases based on information from the context. Though the phrases containing Q&A special expressions are considered invalid, the phrases following some special Q&A expressions are very likely to refer to the library functionalities. For example, in Fig. 5, the 5th verb phrase *"need to ..."* should be filtered, but the 7th verb phrase *"set up the print areas for the excel file"* following the Q&A phrase *"need to"* is a functional feature. For each phrase, we analyze its preceding content in the same sentence. If we find a match with Q&A special expressions before the phrase, we upvote the phrase.

The third filter is based on the structure of the phrase in the syntax tree. We use syntactic structure characteristics to filter out invalid verb phrases. For example, the 3rd and the 6th phrases in Fig. 5 do not contain any verbs as direct children and will be filtered out with the structural filter. Besides, there are usually some complex sub-clauses in the verb phrase. We hope to keep our generated features as concise as possible, therefore we remove the sub-clauses. Another important purpose of filtering parse tree structures is to get the candidate phrases ready for the later normalization. The structural filter ensures the phrase candidates are compatible with the normal form.

**Table 2**
The normal form of feature phrases.

| | | |
|---|---|---|
| Feature | ::= | Action Object [Condition] |
| Action | ::= | verb [particle] |
| Object | ::= | dt adj noun |
| Condition | ::= | prep [verb] Object |

### 3.3. Phrase normalization

To cluster verb phrases with similar meaning, we define the normal form of feature phrases as Table 2 shows. The symbol "[]" denotes that a component is optional. Generally speaking, a functional feature consists of at least an *Action* and an *Object*, which could be modified by a *Condition* (usually a prepositional phrase).

Our pilot study summarizes the common parsing tree types that are compatible with our normal form. To put this straight, Table 3 lists 6 types and their transformation rules to the normal form. Case #1 is the most common case, denotes the verb phrases that consist of a verb and a noun phrase. Particles for the intransitive verbs are presented in case #2. Case #3 is another popular case that contains a verb, noun phrase (NP), and prepositional phrase (PP). Case #4 denotes the verb phrases that do not contain a direct noun phrase. Case #5 is for the noun phrases that consist of a word chain headed by a noun. Case #6 is a typical prepositional phrase.

After normalization, we rebuild the tree representation for the phrase and apply *gSpan* algorithm to mine frequent subgraphs as our final functional features. Furthermore, we merge two phrases if they share the same objects and their action words are synonyms judged by WordNet.[3] Fig. 6 explains why normalization is necessary. Fig. 6(a) is the parse tree of the verb phrase *set the print area* and Fig. 6(b) depicts another candidate phrase *set up the print areas for the excel file*. The original parsing trees contain many detailed grammatical ingredients, which prevent us from mining valuable common subgraphs. The two largest common subgraphs between tree (a) and tree (b) are (VP (VB set) (NP)) (in red color and bold font) and (NP (DT the) (NN print)) (in blue color and underscored), which are meaningless. In contrast, Fig. 6(c) and (d) are rebuilt from our normalized phrases, which omit unnecessary details like POS tags and unify the structures of the top layers. Their common parts (in red and bold font) show us a reasonable result.

[3] We use APIs from nltk.corpus.wordnet.

**Table 3**
Transformation rules for common types of verb phrases.

| ID | Grammar pattern | Phrase example | Transformation rule to normal form |
|---|---|---|---|
| 1 | VP := VB NP | Get the cached formula value<br>VP(VB)(NP(DT)(JJ)(NN)(NN)) | VB := verb; NP := Object |
| 2 | VP := VB PRT NP | Set up the print areas<br>VP(VB)(PRT(RP))(NP(DT)(NN)(NN)) | VB := verb;PRT := particle;<br>NP := Object |
| 3 | VP := VB NP PP | Delete documents from lucene index<br>VP(VB)(NP(NN))(PP(IN)(NP(NN)(NN))) | VB := verb;NP := Object;<br>PP := Condition |
| 4 | VP := VB PP | Iterate through the terms in a document<br>VP(VB)(PP(IN)(NP(NP(DT)(NN))(PP(IN)(NP(DT)(NN))))) | VB := verb;IN(in PP) := particle;<br>NP(in PP) := Object |
| 5 | NP := word NN | Case #1-#3 | Map word to dt adj noun in Object |
| 6 | PP := IN NP | Case #3-#4 | IN ::= prep (in Condition);<br>NP ::= Object (in Condition) |

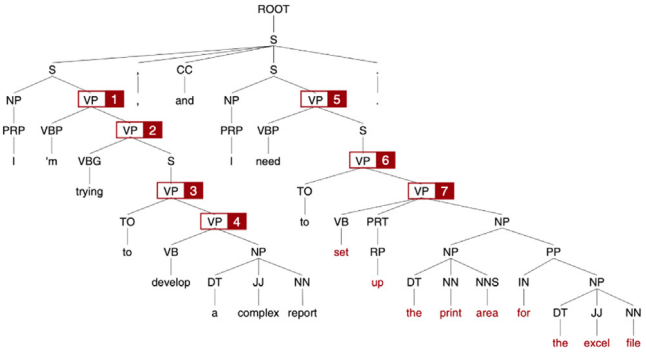


**Fig. 5.** The parsing tree of a long sentence. The seventh verb phrase is a functional feature and the others need to be filtered.

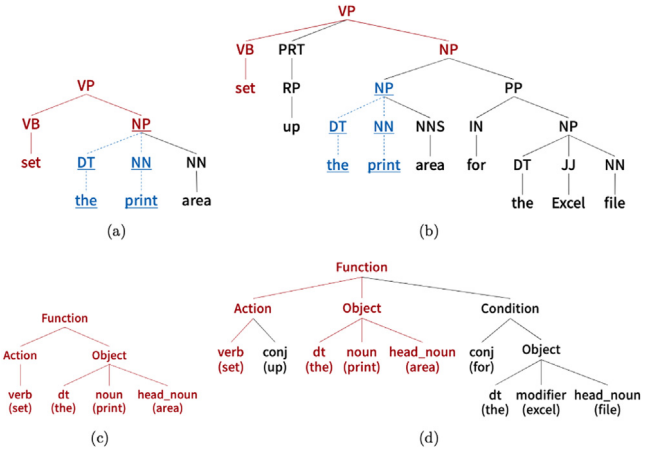


**Fig. 6.** Comparison between parse tree and normalized tree in mining frequent subtrees. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

## 4. Mining code patterns

After getting the list of functional features, we map the features to their implementation. Although Stack Overflow often provides direct code snippets along with the descriptions, such examples are usually incomplete (*i.e.* only mentioned the key APIs instead of the complete solution) and may have quality problems such as incorrect order of API calls (Zhang et al., 2018). To augment these code examples, our main idea is to unveil what has been done in more similar programs. To be specific, we first map each functional feature to a related API and construct a code corpus containing usage examples of the API. Then, we abstract each code example in the corpus into a data flow graph and apply existing frequent subgraph mining algorithm to mine the patterns. Finally, we transform the mined patterns (*i.e.*, frequent subgraphs) back to the text-form code.

```
// snippet 1
style.setFillForegroundColor(short);
style.setFillPattern(SOLID_FOREGROUND);
// snippet 2
style.setFillPattern(SOLID_FOREGROUND);
style.setFillForegroundColor(short);
```

**Fig. 7.** Example snippets where the sequence model fails.

### 4.1. Code corpus construction

We first match each functional feature with a related API and then construct a code corpus by searching usage examples of the API. The rationale behind this design decision is that although code snippets on Stack Overflow suffer the quality problems, they often mentioned the correct API to use. Such APIs could be a starting point to find the complete solution.

Given a functional feature, we view all the code elements mentioned in the same Stack Overflow thread (*i.e.*, contents inside the ⟨*code*⟩ tag) as candidate APIs to match. The metric we use to select the related API is based on the lexical similarity. First, we split the API names according to the camel-case rule and stem the splitted tokens. For each API, we calculate the number of overlapped tokens between its name and the functional feature. (*e.g.*, the number is 2 for the API *"setFillForegroundColor"* and the feature *"set cell color"* since the overlapped tokens are *"set"* and *"color"*). The API with the most overlapped tokens is selected as the matched one and we break the tie by counting the occurrence number of certain API in the thread.

After selecting a related API, we further extract usage examples of the API from client repositories downloaded from Github in advance. If a source code file from the repositories contains the desired API, we add it to the corpus.

### 4.2. Code abstraction

Source code can be viewed as plain text, however, such simple representation is sensitive to trivial differences (*e.g.*, variable names, indentations) and affects the performance of pattern mining. Thus, before applying the frequent pattern mining algorithm on the constructed code corpus, we need to abstract the code into a certain data structure. Common abstractions include the AST (abstract syntax tree) and the method call sequence. As the natural representation of source code, the AST is sensitive to the coding style differences (*e.g.*, using different keywords *"for"* and *"while"* to implement loops). Recently, Raghothaman et al. (2016)

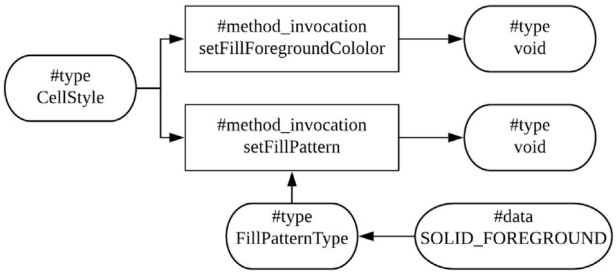


**Fig. 8.** An example data flow graph with type annotations.

and Zhang et al. (2018) applied the *structured call sequence* as the code abstraction to mine API-centric code patterns. The sequence model allows users to define the interested parts of code such as API invocations and guard conditions. However, sometimes changing the order of certain API calls does not affect the program behavior. For example, the two snippets in Fig. 7 behave in the same way. In this case, the sequence model is too sensitive to capture the complete pattern.

Compared to the AST and method call sequence, the graph model is more expressive to describe interactions between variables (Nguyen et al., 2009). In this paper, we augment the data flow graphs by annotating the data nodes with API types. Vertices in a data flow graph can be divided into data and operations. To better fit the library reuse problem, we annotate each data node with the corresponding API type name. Fig. 8 displays the same dataflow graph generated for the two snippets from Fig. 7. The annotations *"CellStyle"* and *"FillPatternType"* are API types from the library `apache-poi`. Also, the different order of method invocations did not affect their abstractions because they share the same data flow.

We follow the common workflow to generate data flow graphs from source code. First, we generate a self-designed intermediate representation (IR) from Java code. The IR is independent of the source language and is designed to be conducive for further processing. Second, we generate control flow graphs from the IR, furthermore, the graphs are refined into the static single assignment (SSA) form. Third, the control flow graphs are transformed into the data flow graphs. The last two steps are the implementation of existing algorithms (Braun et al., 2013) and will not be discussed here. The rest of this subsection will discuss our self-designed IR, which is shown in Fig. 9.

There are two reasons to design our own intermediate representation: First, most existing tools to generate Java IR behave poorly on the incomplete code snippets. *e.g.*, The famous tool Soot[4] requires all dependencies of the current file to generate the corresponding intermediate code. While our tool only requires that the input snippets can be taken as a compilation unit, which can be a method without the wrapper class, or even just a block containing several method invocations. Second, the syntax of Java is complex, there are multiple ways to write code sharing the same behavior. As Fig. 10 shows, to increments a variable, one can write $cnt{+}{+}$ or $cnt \mathrel{+}= 1$. To iterate a list of strings, a for-each loop or an iterator are both correct. Such details have not been normalized in existing tools, while our IR can eliminate some common coding style differences. As a result, the two snippets will result in the same representation in our intermediate code. To be specific, both increment operations are represented by $\langle PstOp \rangle$'++' defined in Fig. 9.

After generating data flow graphs for the corpus, we apply gSpan algorithm again to mine frequent subgraphs as code patterns.

[4] https://github.com/Sable/soot.

### 4.3. Skeleton code

As the last step of pattern mining, we recover the graph-form code patterns into the skeleton code.

**Definition 1.** Skeleton code is an incomplete syntax tree, which is obtained by removing trees rooted at $v_1, v_2, \ldots, v_n$ from a complete syntax tree. Each $v_i$ is a node from the complete syntax tree and we name such nodes as holes in the skeleton code.

For example, Fig. 11 is the skeleton code recovered from the graph in Fig. 8. Nodes wrapped in the dotted line are holes in the syntax tree. Fig. 12 shows the text form of the skeleton code.

During the generation of data flow graphs, we record the corresponding nodes from the syntax tree. To construct the skeleton code from a data flow graph, we first list all the tree nodes included in the graph. Then we randomly select a syntax tree of the original source code and search the least common ancestor (LCA) of the nodes in the tree. After the search, we recover a complete syntax tree containing all the nodes from the graph. Naturally, the missing parts (*i.e.*, nodes not covered by the graph) in the recovered syntax tree become holes in the skeleton code.

## 5. Synthesizer

As the last component of our framework, the synthesizer completes the skeleton code into a well-typed code snippet under the current programming context. We explain the details in this section for reproducibility, but we do not claim the synthesizer as a contribution.

Consider each hole in the skeleton code is annotated with the corresponding type, the synthesis problem can be stated as: given a programming context, how to create an expression with the desired type $\tau$. Here are the three strategies we use:

- pick a variable of $\tau$ from the current context
- call the constructor function of $\tau$
- invoke a method chain and the return type of the last method is $\tau$

The last strategy is a search process. Fig. 13 displays an example of the search tree. Each node in the tree is an API type from the library and an edge connects two nodes if they are separately the caller and the return type of a method. The root of the tree is the type of a declared variable and the leaves are the target type. A path from the root to a leaf represents a method chain which returns the desired type. As Fig. 13 shows, there are four method chains to create a variable with type *"Cell"* from the starting type *"Workbook"*. During the search, we also considered type casting between types by analyzing the inheritance between APIs.

To guide the search process, we define a cost model as the heuristic rule. The model evaluates the goodness of different ways for variable synthesis by mapping them to integers. Using existing variables in context is encouraged, with zero cost. If there are multiple variables with the same type, we choose the one created most recently due to software localness. If a variable is the return value of a certain method, it costs 2 when the method is a constructor and 1 for else. The process for variable synthesis could be recursive, which means in the process of synthesizing the current variable, the invocations require parameters that are not in the context. Our cost model adds the costs for synthesizing these parameters to the total cost.

$$cost(t) = 0, \quad t \text{ in context or } t \text{ is constant} \tag{1}$$

$$cost(f(t_1, t_2, \ldots, t_k)) = price(f) + \sum_{i=1}^{k} cost(t_i) \tag{2}$$

⟨*Goal*⟩ ::= { ⟨*Block*⟩ }

⟨*Block*⟩ ::= ⟨*If*⟩ | ⟨*Label*⟩ | ⟨*Switch*⟩ | ⟨*Goto*⟩ | ⟨*Try*⟩ | { ⟨*Stmt*⟩ }

⟨*If*⟩ ::= `'if'` ⟨*Expr*⟩ { ⟨*Block*⟩ }
| `'if'` ⟨*Expr*⟩ { ⟨*Block*⟩ } `'else'` { ⟨*Block*⟩ }

⟨*Label*⟩ ::= `'Label'` int

⟨*Switch*⟩ ::= `'switch'` ⟨*Expr*⟩ { ⟨*Case*⟩ }
| `'switch'` ⟨*Expr*⟩ { ⟨*Case*⟩ } `'default'` { ⟨*Block*⟩ }

⟨*Case*⟩ ::= `'case'` ⟨*Expr*⟩ { ⟨*Block*⟩ }

⟨*Goto*⟩ ::= `'goto'` ⟨*Label*⟩

⟨*Try*⟩ ::= `'try'` { ⟨*Expr*⟩ } { ⟨*Block*⟩ } `'finally'` { ⟨*Block*⟩ }

⟨*Stmt*⟩ ::= ⟨*Assert*⟩ | ⟨*Throw*⟩ | ⟨*Field*⟩
| ⟨*Assign*⟩ | ⟨*Method*⟩ | ⟨*Return*⟩
| ⟨*Phi*⟩ | ⟨*Ref*⟩ | ⟨*ArrInit*⟩
| ⟨*ArrCall*⟩ | ⟨*PstOpe*⟩ | ⟨*PreOpe*⟩
| ⟨*BinOpe*⟩

⟨*Assert*⟩ ::= `'assert'` ⟨*Expr*⟩

⟨*Throw*⟩ ::= `'throw'` ⟨*Expr*⟩

⟨*Field*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩ '.' string

⟨*Assign*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩

⟨*Method*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩ '.' string '(' ⟨*Args*⟩ ')'
| ⟨*Var*⟩ '=' ⟨*String*⟩ '(' ⟨*Args*⟩ ')'

⟨*Args*⟩ ::= ε | ⟨*Expr*⟩ { ',' ⟨*Expr*⟩ }

⟨*Return*⟩ ::= `'return'` [ ⟨*Expr*⟩ ]

⟨*Phi*⟩ ::= ⟨*Var*⟩ '=' `'phi'` '(' ⟨*Var*⟩ { ',' ⟨*Var*⟩ } ')'

⟨*Ref*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩ '::' string

⟨*ArrInit*⟩ ::= ⟨*Var*⟩ '=' `'new'` string '[' { ⟨*Expr*⟩ } ']'
| ⟨*Var*⟩ '=' `'new'` string '[' { ⟨*Expr*⟩ } ']' '{' ⟨*Args*⟩ '}'

⟨*ArrCall*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩ '[' ⟨*Expr*⟩ ']'

⟨*PstOpe*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩ ⟨*PstOp*⟩

⟨*PreOpe*⟩ ::= ⟨*Var*⟩ '=' ⟨*PreOp*⟩ ⟨*Expr*⟩

⟨*BinOpe*⟩ ::= ⟨*Var*⟩ '=' ⟨*Expr*⟩ ⟨*BinOp*⟩ ⟨*Expr*⟩

⟨*Expr*⟩ ::= ⟨*Boolean*⟩ | ⟨*Number*⟩ | ⟨*Var*⟩ | ⟨*Keyword*⟩ | string | char |

⟨*PstOp*⟩ ::= '++' | '--'

⟨*PreOp*⟩ ::= '++' | '--' | '+' | '-' | '!' | '~'

⟨*BinOp*⟩ ::= '+' | '-' | '*' | '/' | '%' | '<<' | '>>' | '>>>' | '<' | '>' |
'<=' | '>=' | '==' | '!=' | '^' | '|' | '&' | '||' | '&&' |
`'instanceof'`

⟨*Boolean*⟩ ::= `'true'` | `'false'`

⟨*Number*⟩ ::= int | double

⟨*Keyword*⟩ ::= `'super'` | `'this'` | `'null'`

⟨*Var*⟩ ::= string | '#' int

**Fig. 9.** Grammar of intermediate representation for Java code.

```
// for-each style iteration
for (String s: lst) {
    cnt++; foo(cnt, s);
}
// iterator style iteration
Iterator<String> iter = lst.iterator();
while (iter.hasNext()) {
    cnt += 1; foo(cnt, iter.next());
}
```

**Fig. 10.** Two example snippets of different coding styles.

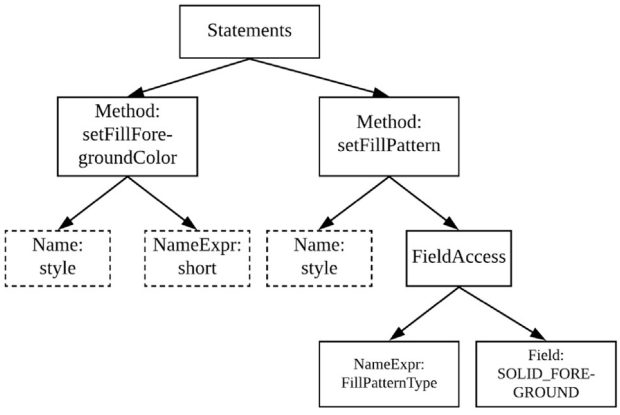


**Fig. 11.** Example skeleton code.

```
<HOLE1>.setFillForegroundColor(<HOLE2>);
FillPatternType fillPatternType1 =
    FillPatternType.SOLOD_FOREGROUND;
<HOLE1>.setFillPattern(fillPatternType1);
```

**Fig. 12.** Text-form of the example skeleton code.

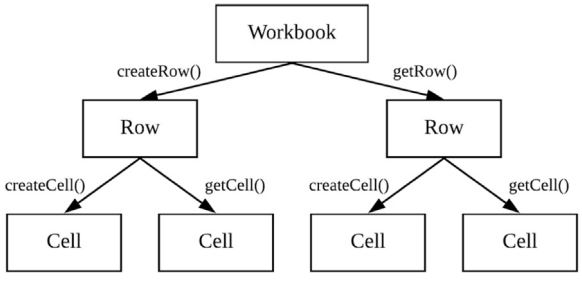


**Fig. 13.** Type-directed search tree.

$$price(f) = \begin{cases} 2 & \text{f is constructor} \\ 1 & \text{else} \end{cases} \quad (3)$$

If two expressions get the same score under the cost model, we break the tie by referring to the code corpus. Recall that each skeleton code comes from a code corpus, we first select instances of the skeleton code from the corpus. For each instance, we extract the variable to fill the hole and locate the definition of the variable by analyzing the *"def-use"* relationship. The process of extraction can be recursive because the definition of a variable may use other undeclared variables. The recursion terminates when we find all definitions of the variables or we meet a variable defined outside the current method body (*i.e.*, parameters of the method, global variables). As a result, from each instance in the corpus, we extract an expression (*i.e.*, a method chain) to fill the hole. For the two synthesized expressions with the same score, we calculate their frequency in the expressions extracted from the corpus and recommend them in the order of decreasing frequency.

## 6. Evaluation

In this section, we evaluate NLI4J from three perspectives, corresponding to the three components of the framework. First, to evaluate the accuracy and completeness of the functional features, we compare the extracted functional features with the

library functionalities provided by the official tutorials. Second, to evaluate the quality of code patterns, we use the code examples from the official tutorials as benchmarks and compare our mining algorithm with two existing pattern mining tools (Gvero and Kuncak, 2015; Zhang et al., 2018). Third, we evaluate the synthesizer with a controlled experiment on `apache-poi`. In the study, we implement an IDE plugin by putting all the three components together and investigate whether the plugin could save programmers' time to solve real-world tasks.

Our research questions are as follows:

- $RQ_1$ : *How well does our filtering pipeline perform on selecting functional features from user discussions?* This question aims at accessing whether our filtering pipeline is effective to filter unrelated verb phrases. Furthermore, we investigate the importance of each filter in the process.
- $RQ_2$ : *To what extent is* NLI4J *able to provide accurate and complete functional features?* This question evaluates the accuracy and completeness of the normalized functional features. Here, accuracy refers to each functional feature should clearly correspond to a functionality. The completeness refers to the capability of the generated functional features to cover frequently-used library functionalities.
- $RQ_3$ : *How does our code pattern mining algorithm perform compared to existing mining tools?* This research question is related to the quality of the mined code patterns. Given the same codebase, we compare our mined code patterns with two existing mining algorithms, which separately abstract source code into syntax trees and sequences.
- $RQ_4$ : *To what extent is* NLI4J *able to promote the efficiency to solve real-world programming tasks?* Finally, this research question directly investigates the usefulness of NLI4J in real-world development.

In the following, we first introduce our datasets and benchmarks. Then, for each research question, we detail our evaluation methodology and results in an individual subsection.

### 6.1. Datasets and benchmarks

To answer the research questions, we collect data for five Java libraries: an html extraction library (`jsoup`), a source code parser (`eclipse-jdt`), a library manipulating Microsoft documents (`apache-poi`), a deep learning toolkit (`deeplearning4j`) and a graph database platform (`neo4j`). In addition to being widely used, these five libraries cover different domains of programming, from the front-end html parsing to the back-end database manipulation.

To construct NLI for a given library, our tool requires (1) related threads from Stack Overflow, and (2) client code reusing the library APIs.

Stack Overflow provides a tag for each of the five libraries (*e.g.* tag *"jsoup"* for the `jsoup` library). For each library, we crawl all the threads containing tag *"java"* and the library-specific tag. Since our functional feature extractor processes a single sentence at a time, we extract textual contents of the threads and split the text into sentences using the Stanford NLP toolkit. The sentences form our first dataset $SO_{large}$. Table 4 lists the number of the threads and the split sentences in $SO_{large}$. Furthermore, we extract a smaller dataset $SO_{small}$ by randomly sampling 100 sentences for each of the five libraries. During the sampling, we remove sentences that are shorter than 15 characters, since such sentences are usually mistakenly split and seldom contain functional features. As a result, the dataset $SO_{small}$ contains 500 sentences. Based on our theoretical definition of the extraction process, the first author manually labels the functional features for each sentence in $SO_{small}$.

**Table 4**
Overview of the Stack Overflow dataset.

| Library | # threads | # sentences |
|---|---|---|
| jsoup | 649 | 2,780 |
| apache-poi | 2496 | 8,046 |
| neo4j | 1600 | 8,144 |
| deeplearning4j | 290 | 1,310 |
| eclipse-jdt | 805 | 3,461 |
| **All** | **5840** | **23,741** |

**Table 5**
Overview of the client code dataset.

| Library | # repositories | # source files |
|---|---|---|
| jsoup | 119 | 5,077 |
| apache-poi | 239 | 21,601 |
| neo4j | 291 | 37,428 |
| deeplearning4j | 48 | 7,470 |
| eclipse-jdt | 26 | 34,254 |
| **All** | **723** | **105,830** |

**Table 6**
Benchmarks from the official tutorials.

| Library | # functionalities | Average LoC |
|---|---|---|
| jsoup | 13 | 4.1 |
| apache-poi | 46 | 12.3 |
| neo4j | 9 | 2.9 |
| deeplearning4j | 21 | 10.9 |
| eclipse-jdt | 6 | 18.0 |
| **All** | **95** | **10.3** |

For client code, we build the dataset by downloading all the client repositories using the Github APIs.[5] Given a library, the query we used is restricted as follows: the body is the name of the library (*e.g.*, jsoup), the programming language is specified as Java, and each repository should have at least five stars. Table 5 lists the number of the client repositories we download and the number of the source files from the repositories.

To evaluate the generated NLI, a list of library functionalities and their implementations are required as benchmarks. We turn to the official tutorial for each of the five libraries. The names of the tutorials vary between libraries, (*e.g.* cookbook, developers' guide), and we organize each tutorial as a list of functionalities. Each functionality is a pair consisting of a concise description and a code example. We filter the functionalities with too long code examples (*i.e.* more than 20 lines of code after removing the comments) because instead of discussing a specific feature, such long examples are more likely to describe a topic or a complete procedure to reuse the library. After the filtering, we treat all the left official functionalities as benchmarks in our evaluation. For each library, Table 6 lists the number of the functionalities and the average lines of a code example (LoC) in the benchmarks.

### 6.2. RQ1: Filtering pipeline

As the first step of our algorithm, NLI4J extracts all the verb phrases from user discussions and then select functional features by filtering unrelated phrases. This subsection discusses the output of our filtering pipeline on the labeled dataset $SO_{small}$.

#### 6.2.1. Methodology

We first combine all the three filters (*i.e.* stop word filter, context filter, and structure filter) to filter verb phrases. The results are compared to the manually labeled results on the

[5] https://api.github.com/search/repositories.

**Table 7**
Filtering results for the five libraries.

| Library | # phrases | # features | # correct sentences |
|---|---|---|---|
| jsoup | 245 | 55 | 96 |
| apache-poi | 277 | 81 | 92 |
| neo4j | 290 | 75 | 87 |
| deeplearning4j | 284 | 68 | 96 |
| eclipse-jdt | 264 | 36 | 94 |
| **All** | **1360** | **315** | **465** |

**Table 8**
Results for different combinations of the three filters.

| Filter combination | # correct sentences | # wrong features | # missing features |
|---|---|---|---|
| context + structure | 120 | 478 | 21 |
| word + structure | 414 | 10 | 103 |
| word + context | 389 | 97 | 36 |
| All filters | 465 | 12 | 41 |

dataset $SO_{small}$. We automatically compare the results with a script that simply matches the textual contents. To avoid the mistakes caused by trivial details in natural language (*e.g.* tenses of verbs, the plural form), instead of asking the annotator to label the benchmarks from scratch, we provide the extracted verb phrases and let the annotator select the functional features from the phrases. If the provided phrases already missed certain functional features, the verdict of this sentence will be a failure even before comparison.

Furthermore, to evaluate the importance of each filter, we created three new filtering pipelines by removing one filter at a time. Then, we evaluated the three modified pipelines using the same script.

#### 6.2.2. Results

From the 500 sentences, our extractor extracted 1,360 verb phrases using the Stanford NLP toolkit. We fed the phrases to our filter pipeline and got 315 functional features. For 93% (465 out of 500) sentences, the automatically extracted functional features matched the labeled ones in the benchmark. Our tool missed 41 functional features and gave 12 wrong features in the remaining 35 sentences. Table 7 summarizes details of the results for each library. For each library, we list (1) the number of verb phrases mined from the sentences, (2) the number of functional features after filtering, and (3) the number of sentences which the filtered results match the benchmarks. From the result, we did not notice significant variations between different libraries. However, it is possible that our fixed stop words can cause some false negatives when a new library is specified, since a stop word could be a domain-specific concept or action for the new library. In that case, a customized stop word list is recommended.

We checked each of the 35 failed sentences and summarized two main reasons for the mistakes, which resulted in both missing and wrong functional features:

- *Preprocessing of natural language.* We found in more than half of the failed sentences, the NLP toolkit did not produce the correct verb phrase list as expected.
- *Tangled votes.* Some phrases were upvoted and downvoted at the same time, and our current weights for the filters lead to a wrong decision for these phrases.

The first reason is an external factor to our tool. We found the Stanford NLP toolkit sometimes failed to split the sentences correctly when the punctuation characters are not correctly used. Also, a common case for failed POS tagging is when verbs appear at the beginning of sentences. For the second reason, as there were both upvotes and downvotes in our filter pipeline, sometimes they are tangled and bring mistakes in the functional feature recognition. For example, the phrase *"return the node of the highest score"* was missing from the sentence *"With Cypher, I'm trying to return the node of the highest score."* because it was downvoted for using a stop word *return* and upvoted for the context (with a preceding Q&A expression *I'd like to*). Machine learning approaches could help in such a scenario by assigning proper weights for the three filters (all one point in our current implementation). However, consider the small size of the annotated sentences and the fact that the current algorithm is accurate for most sentences, we did not apply machine learning approaches at present.

To evaluate the importance of each filter, we create three new filtering pipelines by removing one filter at a time. The results are displayed in the first three rows of Table 8 and the last row combines all the three filters. When the stop word filter is removed, the number of wrong features rapidly raises to 478 as the first row shows. The context filter upvotes the verb phrases following Q&A expressions, as a result, the filter pipeline tends to give a lower score for each phrase after removing this filter. As the second row shows, the number of missing features without the context filter is the largest. The third row depicts the result for removing the structure filter, which also brings more incorrect features.

*Answer for RQ1:* On a labeled dataset containing five hundred sentences, our filtering pipeline correctly filters unrelated verb phrases for 465 (93%) sentences. All the three filters contribute to the performance and the stop word filter is proved to be the vital factor.

### 6.3. RQ2: Functional features

In this subsection, we generate functional features for each library from the dataset $SO_{large}$. Our benchmark is the lists of functionality descriptions from the official tutorials. The evaluation checks both (1) whether each functional feature is accurate, and (2) whether each functionality in the tutorial is covered.

#### 6.3.1. Methodology

Given the Stack Overflow corpus of a library, the output of our functional feature extractor is a list of functional features in verb phrase form.

To evaluate the accuracy, we ask two annotators to rate the extracted functional features. They are requested to give a score for each feature: two points for an actual library functionality, one point for a likely functionality that requires further information to make it clear, zero point for a meaningless phrase. Two annotators mark the functional features separately and afterwards discuss to reach an agreement. We count all the ratings by annotators and calculate the average score for all the functional features.

To evaluate the completeness, we ask the annotators to review the functionalities in the benchmark one by one and judge whether the functionality is included in our generated functional features. Again, we ask the annotators to give a score for each functionality. If a functionality is included in our generated features, our result gets two points. If our output includes a similar functional feature but not precise, our result gets one point. Otherwise, our result gets zero point for the functionality.

**Table 9**
Accuracy of the generated functional features.

| Library | # features | Score | | | Average |
|---|---|---|---|---|---|
| | | 2 | 1 | 0 | score |
| jsoup | 86 | 41 | 27 | 18 | 1.26 |
| apache-poi | 190 | 116 | 48 | 26 | 1.47 |
| neo4j | 119 | 61 | 40 | 18 | 1.36 |
| dl4j | 33 | 16 | 14 | 3 | 1.39 |
| eclipse-jdt | 103 | 48 | 47 | 8 | 1.39 |
| **All** | **531** | **282** | **176** | **73** | **1.39** |

**Table 10**
Completeness of the generated functional features.

| Library | # functions | Score | | | Average |
|---|---|---|---|---|---|
| | | 2 | 1 | 0 | score |
| jsoup | 13 | 10 | 3 | 0 | 1.77 |
| apache-poi | 46 | 30 | 12 | 4 | 1.56 |
| neo4j | 9 | 7 | 1 | 1 | 1.67 |
| dl4j | 21 | 10 | 3 | 8 | 1.09 |
| eclipse-jdt | 6 | 6 | 0 | 0 | 2.00 |
| **All** | **95** | **63** | **19** | **13** | **1.52** |

#### 6.3.2. Results

Table 9 displays the results for the accuracy of the functional features. The first column is the name of the library and the second column is the number of the normalized functional features extracted from the $SO_{large}$ dataset. The third column lists the number of functional features marked with three different scores and the last column is the average score for all the functional features in this library. As the last row shows, for a total of 531 functional features, 282 (53.1%) of them are annotated with two points, 176 (33.1%) are annotated with 1 point and the remaining 73 (13.7%) are irrelevant to library functionalities. The average score shows that our functional features get approximately 1.39 points out of two.

Table 10 displays the results for the completeness of the functional features. Instead of rating a functional feature, we rate each functionality from the official tutorial in Table 10. The result shows that our generated functional features can cover 86.3% (82 out of 95, 66.3% with two points, 20.0% with one point) of the functionalities in the benchmark. For the functions which get one point, our annotators reported that the majority of them are caused by the fact that the tutorial summarizes several tasks into one functionality. For example, the last function in the tutorial of `apache-poi` is *"cells with multiple styles"*, which mentioned three tasks (setting color, font and cell style) at the same time. There is little chance that a user will discuss the three functionalities together in a verb phrase. We carefully analyzed all the 13 missing functionalities (rated as zero point) by manually searching them on Stack Overflow. As a result, seven of them are never mentioned on Stack Overflow, the rest six functionalities are discussed fewer than three times in the whole corpus. Since we only keep the frequent normalized syntax trees during normalizing functional features, phrases with low frequency will not be included in our final result.

The results show a little fluctuation among different libraries, especially for the completeness. In Table 10, the `eclipse-jdt` library gets the highest score of 2.00, while the `deeplearning4j` library is rated the lowest (1.09 points). The fluctuation comes from the different sizes of related threads on Stack Overflow. In fact, as Table 4, the number of threads under the tag `deeplearning4j` is the smallest in our dataset. The small size of discussions obviously affects the completeness of functional features, which is an external threat to our algorithm.

```
// parse text from html
Document document_1 = Jsoup.parse(<$HOLE1>);
document_1.select(<$HOLE2>).first().text();
```

```
// create an embedded database
GraphDatabaseFactory factory_1 =
    new GraphDatabaseFactory();
GraphDatabaseService service_1 =
    factory_1.newEmbeddedDatabase(<$HOLE1>);
```

```
// configure a network
MultiLayerConfiguration configuration_1 =
    new NeuralNetConfiguration.Builder()
    .seed(<$HOLE1>).iterations(<$HOLE2>)
    .list().layer(<$HOLE3>).build();
```

```
// merge cells
CellRangeAddress address_1 = new
    CellRangeAddress(
    <$HOLE1>, <$HOLE2>,
    <$HOLE3>, <$HOLE4>
);
<$HOLE5>.addMergedRegion(address_1);
```

```
// save workbook
Workbook wb_1 = new HSSFWorkbook();
try {
    wb.write(<$HOLE1>);
} catch (IOException e) { <$BODY> }
```

**Fig. 14.** Example code patterns for functional features.

*Answer for RQ2:* By comparing the functional features with the official tutorials, we found that 86.2% (458 out of 531) functional features are accurate. Furthermore, the features can cover 86.3%(82 out of 95) of the functionalities listed in the official tutorials.

### 6.4. RQ3: Code pattern

In this section, we focus on evaluating the performance of our code pattern mining algorithm. We start the evaluation from some examples and then compare our miner with two existing pattern mining tools.

#### 6.4.1. Examples

Fig. 14 shows five code patterns that NLI4J mines. A symbol with $ denotes that there is a missing part in the code pattern. To be more specific, a ⟨$*HOLE*⟩ represents a missing variable and a ⟨$*BODY*⟩ represents a missing code block. The reader will observe the immediate usefulness of the code patterns for learning API usage.

As Fig. 14 shows, a code pattern usually describes the frequent combination of API elements. The fourth code pattern (*i.e., "merge cells"*) denotes that a cell region is managed by the class *CellRangeAddress*, which is usually invoked with another method *addMergedRegion*. To instantiate a *CellRangeAddress* object, four parameters are required to specify the left top and the right bottom corners of the region. Some patterns contain control flow statements besides API invocations, such as the last example *"save workbook"* in Fig. 14. The code pattern not only summarizes the correct APIs to invoke, but also hints that the method *write* needs to handle an exception. The specific way to handle the exception is left to the user in a ⟨$*BODY*⟩ block.

**Table 11**
The number of functionalities to mine code patterns.

| Library | # covered functions | # functions to mine patterns |
|---|---|---|
| jsoup | 13 | 12 |
| apache-poi | 42 | 36 |
| neo4j | 8 | 7 |
| deeplearning4j | 13 | 10 |
| eclipse-jdt | 6 | 4 |
| **All** | **82** | **69** |

**Table 12**
Comparison of three pattern mining tools.

| Library | anyCode | ExampleCheck | NLI4j |
|---|---|---|---|
| jsoup | 0.33 | 0.23 | 0.15 |
| apache-poi | 0.42 | 0.27 | 0.21 |
| neo4j | 0.35 | 0.29 | 0.29 |
| deeplearning4j | 0.79 | 0.68 | 0.56 |
| eclipse-jdt | 0.85 | 0.81 | 0.81 |
| **Average** | **0.48** | **0.36** | **0.29** |

#### 6.4.2. Methodology

To evaluate the performance of our pattern mining algorithm, we use code examples from the official tutorials as the benchmark. From the total 95 functionalities in the benchmarks, we first remove the 13 functionalities that are not covered by our functional features (*i.e.* the functionalities with zero point in Table 10). Then we removed another 13 functionalities because our algorithm failed to match a correct API from the corresponding functional feature. Table 11 shows the 69 left functionalities.

As there is no universal metric to measure the quality of code patterns, we approximated the quality by calculating the *Jaccard distance*. To be more specific, we built a set of the invoked APIs in the code pattern and another set of the invoked APIs in the official example. Jaccard distance is the metric to calculate the differences between two sets as follows:

$$dis(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|} \tag{4}$$

We compare our pattern mining algorithm with two existing tools, *i.e.*, anyCode and ExampleCheck. anyCode expands an API element into a Java expression with a pre-trained PCFG (Probabilistic Context Free Grammar) model. The second tool ExampleCheck is designed to check API misuse from the Stack Overflow. The rationale behind is to compare API examples from Stack Overflow with the mined API usage patterns from Github. We choose the two tools because their abstractions for source code are representative. anyCode abstracts code into tree-based structure PCFG, and ExampleCheck abstracts code into the sequence structure SCS (*i.e.*, Structured Call Sequence). To make the comparison meaningful, we configure the settings for all three tools as follows:

- *The same codebase.* All three tools are provided with the same codebase, which is all the usage examples for a given API. On average, the codebase for each API contains 217 source code files.
- *The same threshold.* We set the threshold (5%) for the minimum frequency of a pattern to be mined from the codebase.

#### 6.4.3. Results

Table 12 shows the results of the experiments. Given a tool and a library, we list the average Jaccard distance between the mined code patterns and the code examples from the benchmarks. Overall, NLI4j achieves the minimum average Jaccard distance (0.29), which proves the code patterns mined by our tool are more similar to the official code examples. In the experiment, we found that anyCode can only synthesize quite short patterns. Some code examples in our benchmark contain more than ten API invocations, as a result, the performance of anyCode on these cases are not as good as the other two tools. ExampleCheck and NLI4j can generate more complete and complex code patterns. However, as we explained before, in such cases, the sequence structure ExampleCheck used is too strict for pattern mining. For example, for the color setting task in Apache POI, we found that two APIs (*i.e.*, *"setFillForegroundColor"* and *"setFillPattern"*) could be swapped. However, swapping two APIs will result in two different subsequences for ExampleCheck and it failed to produce the complete pattern.

Besides, we observed the fluctuation among different libraries. For `deeplearning4j` and `eclipse-jdt`, we found the Jaccard distances of all the three tools are significantly larger than the rest three libraries. In fact, we found the style of API usage varies among different libraries. For example, `deeplearning4j` often requires a long method chain to configure the network from all aspects. However, users of `deeplearning4j` may skip some aspects in their client code, as a result, the mined patterns are visibly shorter than the official code examples. For `eclipse-jdt`, many functionalities of the library apply the visitor pattern (a design pattern). All the code abstractions of the three tools are designed to analyze code snippets inside a method, which could not represent the visitor pattern well.

*Answer for RQ3:* Given the same codebase, our generated code patterns are more complete and accurate than two existing pattern mining tools.

### 6.5. Controlled experiment

We conducted a controlled experiment on the library *apache-poi* to see whether our tool can improve the efficiency of reusing libraries in real-world programming. We also recorded the ranking of the expressions accepted by users to evaluate the performance of our synthesizer.

#### 6.5.1. Methodology

To evaluate the utility and effectiveness of programming with NLI4j, we invited 8 participants to solve real-world programming tasks using the tool. All the participants were familiar with the Java programming language and were divided into two groups. The newcomer group consisted of five participants new to the library *apache-poi*. The rest three participants once built client projects with *apache-poi* and they formed the expert group. The logic and usage of NLI4j were briefly introduced to all the participants in advance.

We prepared five specific programming tasks for the participants to solve. Prototypes of the tasks were randomly picked from an online tutorial website.[6] We concretized the tasks for two reasons. First, some tasks require specific configurations to be automatically validated. For example, we concretized the task *"create blank workbook"* by specifying the file name and the path to save it. Second, some tasks in the tutorial are supposed to teach users how to manipulate a class for multiple subtasks. Separating the subtasks makes it more executable for composing solutions and validators. Table 13 lists the tasks with a brief description and the number of API elements invoked in the code example from the tutorial. On average, one task in our experiment

[6] https://www.tutorialspoint.com/apache_poi.

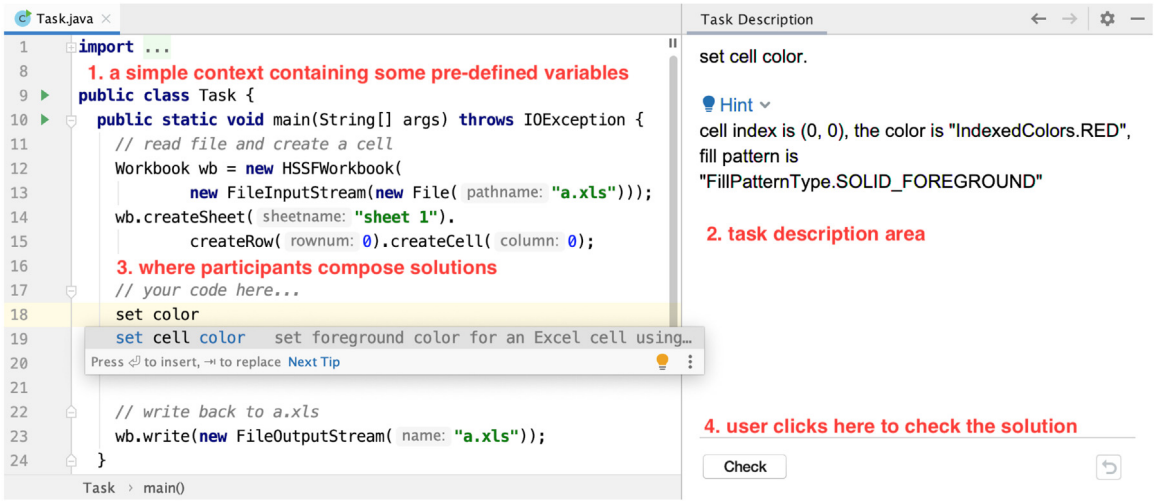


**Fig. 15.** UI for our controlled experiment. The user is invoking a functional feature.

**Table 13**
Five tasks for participants to solve with Apache-POI.

| Id | Task description | #invoked APIs |
|---|---|---|
| 1 | Create blank workbook | 4 |
| 2 | Write into a spreadsheet | 3 |
| 3 | Set cell color | 6 |
| 4 | Set italic font and font color | 7 |
| 5 | Create hyperlink to URL | 7 |

invokes 5.4 API elements. Fig. 15 shows the user interface of our controlled experiment. Each task has three components: a task description with a detailed hint, a solution file with some pre-defined variables, and a validator program. All tasks consider the fill-in-the-blanks approach, which meant the participants needed to fill the solution file by implementing the missing functions. A task is considered to be accomplished if the validator returns the accepted page. The tool for our controlled experiment is available in the published online artifacts.

We allowed all participants to visit online resources such as Q&A forums and search engines when solving tasks, but we recorded the number of pages they opened in the process. Two settings were configured for the coding environment, one equipped with the NLI4J plugin and the other without it. Participants were assigned at random to each programming task and each coding environment, and thus there was no proper balance. No participants were assigned to the same task with different coding environments. For the participants who used NLI4J, their interaction with the plugin was recorded. Recall that our synthesizer would recommend synthesized expressions to users, we recorded whether the user accepted the recommendation and the ranking of the expression that they used. Finally, the overall task duration and the number of websites viewed was recorded to facilitate data analysis.

### 6.5.2. Results

Table 14 shows the results of the controlled experiment. Columns (1) and (2) display the participant's index and the reported programming expertise (N stands for the newcomer, and E for the expert). Columns (3) and (4) display the task and the code environment (STD stands for the standard IDE and NLI stands for the IDE equipped with NLI4J plugin). Finally, Column (5) refers to the overall duration of the task, and Column (6) displays the number of web pages the participant opened for the task.

**Table 14**
Summary of experiment results.

| (1) Id | (2) Expertise | (3) Task | (4) Environment | (5) Time (s) | (6) # pages |
|---|---|---|---|---|---|
| 1 | N | 1 | NLI | 144 | 0 |
| | | 2 | STD | 377 | 4 |
| | | 3 | NLI | 180 | 2 |
| | | 4 | STD | 520 | 6 |
| | | 5 | STD | 1021 | 8 |
| 2 | N | 1 | STD | 212 | 2 |
| | | 2 | STD | 419 | 4 |
| | | 3 | NLI | 306 | 3 |
| | | 4 | NLI | 729 | 5 |
| | | 5 | STD | 741 | 9 |
| 3 | N | 1 | NLI | 165 | 0 |
| | | 2 | NLI | 265 | 2 |
| | | 3 | STD | 764 | 10 |
| | | 4 | STD | 1189 | 10 |
| | | 5 | STD | 812 | 7 |
| 4 | N | 1 | STD | 315 | 5 |
| | | 2 | NLI | 197 | 2 |
| | | 3 | STD | 610 | 6 |
| | | 4 | NLI | 576 | 3 |
| | | 5 | NLI | 382 | 3 |
| 5 | N | 1 | NLI | 190 | 0 |
| | | 2 | STD | 598 | 8 |
| | | 3 | NLI | 247 | 3 |
| | | 4 | STD | 1186 | 5 |
| | | 5 | NLI | 431 | 5 |
| 6 | E | 1 | NLI | 90 | 0 |
| | | 2 | STD | 197 | 2 |
| | | 3 | NLI | 91 | 0 |
| | | 4 | NLI | 410 | 1 |
| | | 5 | STD | 547 | 1 |
| 7 | E | 1 | STD | 122 | 1 |
| | | 2 | NLI | 109 | 2 |
| | | 3 | STD | 169 | 1 |
| | | 4 | STD | 623 | 4 |
| | | 5 | NLI | 315 | 0 |
| 8 | E | 1 | STD | 176 | 1 |
| | | 2 | NLI | 138 | 0 |
| | | 3 | STD | 201 | 1 |
| | | 4 | NLI | 484 | 5 |
| | | 5 | NLI | 206 | 0 |

Figs. 16 and 17 summarize the data from Table 14 for newcomers. It compares the average time (minutes) used and the number of web pages opened by the newcomers between two

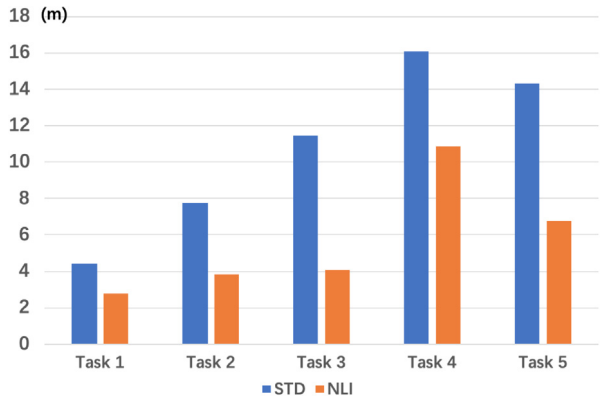


**Fig. 16.** Comparison between the average time newcomers spent in two coding environments.

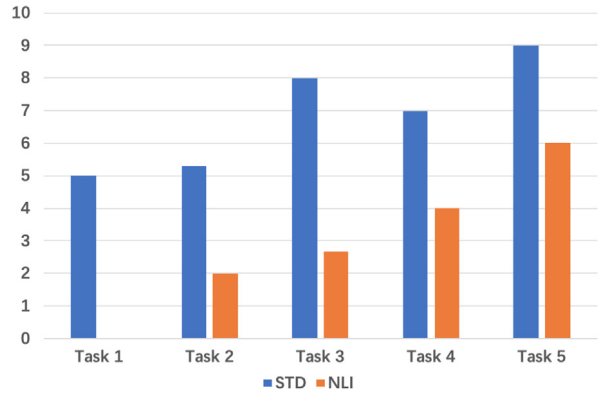


**Fig. 17.** Comparison between the average number of web pages newcomers visited in two coding environments.

coding environments. On average, newcomers without NLI4J spent 674 s and visited 6.5 web pages for each task, which is significantly larger than the number for participants using the plugin (317.7 s and 2.3 pages). Notice that when newcomers using NLI4J met the first task (*create blank workbook*), all of them solved the task without referring to any web sources.

However, such a difference is not that obvious in the expert group. On average, an expert with NLI4J solved a task in 230.3 s, an expert without NLI4J solved a task in 290.7 s. From later communication with the three experts, we found they were familiar with how to read and search library documentation, which can explain the number of web pages they opened was much smaller than the newcomer group. However, all of the three experts confirmed the plugin is convenient when they forgot how to use a certain API. Many participants reported that when they used the plugin, they could accomplish most tasks without external information and the visited web pages were only to confirm the solution.

For participants who solved tasks with NLI4J, we also asked them to record the rankings of the expressions they chose to complete the code patterns. We wanted to know whether our synthesizer recommended useful expressions to the participants. Table 15 shows the result, the second column lists the number of interactions for each task. An interaction means there is a missing variable for users to provide or select from the list of recommended expressions. We use two metrics, *i.e.* MRR (Mean Reciprocal Rank) and Hit@1 to evaluate the quality of the recommendation. Our synthesizer could not recommend useful expressions if the missing part must be specified by users. For

**Table 15**
Recommendation performance of the synthesizer.

| Task ID | # interactions | MRR | Hit@1 |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
| 2 | 2 | 0.25 | 0 |
| 3 | 4 | 1.00 | 1.00 |
| 4 | 4 | 0.375 | 0.25 |
| 5 | 2 | 0.50 | 0.50 |
| **Average** | **2.6** | **0.54** | **0.46** |

example, the only interaction in the first task is the name of the workbook, which could be any valid string. NLI4J failed to recommend the desired string and get 0 for both MRR and Hit@1 metrics. Actually, among all 13 interactions for the tasks, such conditions (arbitrary values of built-in types) happened 5 times. For all the other interactions, NLI4J successfully recommended the desired expressions at a top-2 position. In the third task, all four desired variables were recommended as the first choice. On average, each task requires 2.6 interactions and the average MRR value is 0.54 and the value for Hit@1 is 0.46.

### 6.5.3. Discussion

All participants were asked to fill a simple survey after the controlled experiment. The survey form is available in our published artifacts. Form the survey result, we can see all participants agree that using NLI4J could improve their coding efficiency. When asked to compare the input form of functional features with free-form natural language, most participants (6 out of 8) reported that they preferred functional features. However, some participants raised the concern that for those functions not included in functional features, they could only turn to the free-form queries. Besides, one of our participants mentioned that although free-form queries are flexible, however, composing such queries from scratch could be difficult for a newcomer. He mentioned that some hint like auto-completion or our functional features would be very helpful when users described their requirements.

We also asked the participants to compare code patterns used in NLI4J with concrete code examples. Overall, most of the participants (5 out of 8) preferred code patterns with two main reasons. First, code patterns gave a more clear hint for where to modify. Second, participants believed that code patterns had higher quality and were more reliable since they were mined from multiple concrete examples.

*Answer for RQ4:* The result of the controlled experiment shows that NLI4J can save half of the coding time for newcomers of a library. For experienced developers, NLI4J can play the role of a prompter when they forget the usage of certain APIs. Given a programming context, the recommended expressions from our synthesizer can effectively help developers fill the missing parts.

## 6.6. Threats to validity

**Internal validity:** Our four research questions covered the key steps in constructing NLI (*i.e.*, functional feature extraction, code pattern mining, and the synthesizer). However, we could not evaluate all the details in the implementation because our framework has a quite long workflow. For example, we did not discuss parameter tuning for our frequent pattern mining algorithm. In our current implementation, we set the frequency threshold as 5% to mine code patterns and it works well on our datasets. However, the best threshold may vary under different datasets.

For the case study, although we have considered the help of NLI4J varies for different users. The total number of participants is relatively small. We plan to put our tool in the daily development of developers and collect more user data in our future work.

**External validity:** We selected five libraries from different domains, which covered the front-end parsing tool, the back-end database, and popular toolkits. The evaluation shows that our tool can mine accurate functional features (accuracy of 86.2%) and high-quality code patterns. However, since our tool is feature-oriented, its performance on libraries with clear features are usually better than the libraries which are designed as frameworks. Furthermore, API invocation is not the only way of library reuse. Some libraries heavily rely on other design or syntax, such as design patterns and annotations (*e.g.*, the OGM mechanism in *neo4j*). Thus, the first external validity is the generalization of our framework to other libraries.

We carefully chose the datasets in our experiment so the findings could be generalized as much as possible. We selected Stack Overflow to extract functional features because it is one of the most popular platforms to search for programming tasks (Ragkhitwetsagul et al., 2018). There are a lot of discussions about API usage from the site, and many previous studies encourage us to select it as the corpus (*e.g.*, Treude and Robillard, 2016; Zhang et al., 2019). Nonetheless, not all the libraries are active on Stack Overflow. Although most of our design is not specific to Stack Overflow, the performance may differ when other forms of user discussions are used as input. Regarding the codebase for mining code patterns, we downloaded all repositories with at least 5 stars from Github and the number of source code files is more than 105K. In our experiment, we found a code corpus containing one hundred files is good enough to mine high-quality patterns. However, we only evaluated on Java APIs and it may not be representative to all the languages and libraries.

## 7. Related work

The idea of NLI2Code contributes to the large body of work on API comprehension and software reuse. In addition, each of the three components has benefited from related work in the corresponding domain, which will be summarized in this section separately.

### 7.1. Information extraction from software artifacts

Several researchers have succeeded in extracting high-quality software specifications from software artifacts using NLP techniques. Zhong et al. (2009) proposed an approach for inferring specifications from API documentation by detecting actions and resources through machine learning. Their evaluation showed relatively high precision, recall, and F-scores for five software libraries, and indicated potential uses in bug detection. Abebe and Tonella (2010) presented an NLP-based approach to extract and organize concepts from software identifiers in a WordNet-like structure through tokenization, part-of-speech tagging, dependency sorting, and lexical expansion. Jiang et al. (2017) introduced an approach to select relevant tutorial fragments for APIs, which combined the topic model and the PageRank algorithm. More closely to our goal, summarizing software artifacts with functional explanations, Henß et al. (2012) designed an approach to extract FAQs from mailing lists and forums. The approach applied the LDA algorithm to extract topic models from the data which are used for the creation of topic-specific FAQs. Treude et al. (2015) defined the concept of task as a specific programming action that has been described in the documentation. Indexing long documents with high-level tasks can help users quickly locate the part they care about. Furthermore, Campbell and Treude (2017) developed a tool that can map tasks to code snippets from Stack Overflow answers. In NLI2Code, we normalize the free-form tasks into a set of pre-defined functional features and enhance concrete code examples into abstract code patterns, considering the quality of Stack Overflow code examples are controversial (Zhang et al., 2018).

### 7.2. Code pattern mining

Code patterns are abstract code examples with metavariables or other components to be completed by users. Modern IDEs usually integrate relevant features to define widely-used code patterns, such as live template feature in IntelliJ IDEA and SnipMatch in Eclipse. Several studies (Wang et al., 2013; Xie and Pei, 2006) applied statistical methods to automatically mine code patterns since source code was shown to be highly repetitive (Hindle et al., 2012). The common workflow for code pattern mining first abstracts source code into a well-designed data structure and then apply the corresponding frequent pattern mining algorithm. Allamanis and Sutton (2014) presented Haggis, a system for mining code patterns that was built on techniques from statistical natural language processing. Haggis transformed source code into abstract syntax trees and applied Bayesian probabilistic tree substitution technique to get code patterns. The mined patterns were proved to be accurate and meaningful and the author mentioned part of the patterns were accepted by the Eclipse SnipMatch project. To detect API misuse in online forums, Zhang et al. (2018) developed a tool ExampleCheck to compare API usages in the forum with code patterns mined from large codebases. The authors designed a data structure called the structured call sequence, which enriched API invocations with syntax like guard conditions and control flow statements. Such enrichment is vital because most API misuses in online code examples suffer from missing guard conditions and exception handling and ExampleCheck could effectively find these misuses.

Compared with existing works, which are designed to solve a particular problem, NLI2Code is designed as an abstract framework, which does not specify the approach to mine code patterns. Code abstraction designed in existing tools may rely on properties of their problems and cannot be easily generalized to others.

### 7.3. Program synthesis from natural language

Program synthesis is the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification (Gulwani et al., 2017). This problem has been considered the holy grail of computer science since the inceptions of AI in the 1950s. Program synthesis works diverse in the form of specification, including partial data structures (Loncaric et al., 2018), test cases (Feng et al., 2017), natural language (Nguyen et al., 2016; Little and Miller, 2007) and their combination (Raza et al., 2015). Despite its ambiguity, the natural language specification is the most flexible one and requires the smallest effort to compose. Existing synthesis tools with natural language input either recommend related APIs (Yessenov et al., 2017; Zamanirad et al., 2017) or compilable snippets (Buse and Weimer, 2012). Gvero and Kuncak (2015) defined a free-form specification that allowed users to write natural language queries and use names of local variables. Given a specification, they mapped it to a method and expanded the method with a PCFG model trained from large codebases. Galenson et al. (2014) developed a dynamic and interactive program synthesis tool CodeHint, which was integrated into the Eclipse IDE. CodeHint allowed users to execute the recommended code snippets and refine the snippets iteratively.

## 8. Conclusion

This paper promotes the concept of NLI (Natural Language Interface) for library reuse. To construct and use NLI, we design a framework with three components (*i.e.*, functional feature extractor, code pattern miner, and synthesizer). We instantiate the three components as a tool NLI4J to reuse Java libraries. The accuracy of our extracted functional features is 86.2% and can cover 86.3% of functionalities provided by the official tutorials. By comparing with existing code pattern miner, NLI4J can mine more accurate and complete code patterns. Finally, a controlled experiment with eight participants on five real-world tasks shows that our tool can save half of the coding time for newcomers of the library. From the practical perspective, our framework promotes the efficiency of reusing libraries. From the academic perspective, our framework lays out a design space of building the natural language interface for libraries, which would hopefully inspire research in this area.

## CRediT authorship contribution statement

**Qi Shen:** Conceptualization, Methodology, Software, Data curation, Writing - original draft. **Shijun Wu:** Methodology, Software, Data curation, Validation. **Yanzhen Zou:** Conceptualization, Methodology, Writing - review & editing. **Zixiao Zhu:** Software, Investigation. **Bing Xie:** Conceptualization, Supervision, Writing - review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This paper is supported by National Natural Science Fund for Distinguished Young Scholars, China (No. 61525201) and General Program of National Natural Science Foundation of China (61972006).

## References

Abebe, S.L., Tonella, P., 2010. Natural language parsing of program element names for concept extraction. In: The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010. pp. 156–159. http://dx.doi.org/10.1109/ICPC.2010.29.

Allamanis, M., Sutton, C.A., 2014. Mining idioms from source code. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 472–483. http://dx.doi.org/10.1145/2635868.2635901.

Braun, M., Buchwald, S., Hack, S., Leißa, R., Mallon, C., Zwinkau, A., 2013. Simple and efficient construction of static single assignment form. In: Jhala, R., Bosschere, K.D. (Eds.), Compiler Construction - 22nd International Conference, CC 2013, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. In: Lecture Notes in Computer Science, vol. 7791, Springer, pp. 102–122. http://dx.doi.org/10.1007/978-3-642-37051-9_6.

Buse, R.P.L., Weimer, W., 2012. Synthesizing API usage examples. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 782–792. http://dx.doi.org/10.1109/ICSE.2012.6227140.

Campbell, B.A., Treude, C., 2017. NLP2Code: Code snippet content assist via natural language tasks. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017. pp. 628–632. http://dx.doi.org/10.1109/ICSME.2017.56.

Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W., 2017. Component-based synthesis for complex APIs. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 599–612.

Galenson, J., Reames, P., Bodík, R., Hartmann, B., Sen, K., 2014. Codehint: dynamic and interactive synthesis of code snippets. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014. pp. 653–663. http://dx.doi.org/10.1145/2568225.2568250.

Gulwani, S., Polozov, O., Singh, R., 2017. Program synthesis. Found. Trends Program. Lang. 4 (1–2), 1–119. http://dx.doi.org/10.1561/2500000010.

Gvero, T., Kuncak, V., 2015. Synthesizing java expressions from free-form queries. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 416–432. http://dx.doi.org/10.1145/2814270.2814295.

Gvero, T., Kuncak, V., Kuraj, I., Piskac, R., 2013. Complete completion using types and weights. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 27–38. http://dx.doi.org/10.1145/2491956.2462192.

Henß, S., Monperrus, M., Mezini, M., 2012. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 793–803. http://dx.doi.org/10.1109/ICSE.2012.6227139.

Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.T., 2012. On the naturalness of software. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 837–847. http://dx.doi.org/10.1109/ICSE.2012.6227135.

Jiang, H., Zhang, J., Ren, Z., Zhang, T., 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 38–48. http://dx.doi.org/10.1109/ICSE.2017.12.

Little, G., Miller, R.C., 2007. Keyword programming in java. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. pp. 84–93. http://dx.doi.org/10.1145/1321631.1321646.

Loncaric, C., Ernst, M.D., Torlak, E., 2018. Generalized data structure synthesis. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 958–968. http://dx.doi.org/10.1145/3180155.3180211.

Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J.R., Bethard, S., McClosky, D., 2014. The stanford CoreNLP natural language processing toolkit. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations. pp. 55–60.

Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N., 2009. Graph-based mining of multiple object usage patterns. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, pp. 383–392.

Nguyen, T.V., Rigby, P.C., Nguyen, A.T., Karanfil, M., Nguyen, T.N., 2016. T2API: synthesizing API code usage templates from English texts with statistical translation. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 1013–1017. http://dx.doi.org/10.1145/2950290.2983931.

Perelman, D., Gulwani, S., Ball, T., Grossman, D., 2012. Type-directed completion of partial expressions. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 275–286. http://dx.doi.org/10.1145/2254064.2254098.

Raghothaman, M., Wei, Y., Hamadi, Y., 2016. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. pp. 357–367. http://dx.doi.org/10.1145/2884781.2884808.

Ragkhitwetsagul, C., Krinke, J., Oliveto, R., 2018. Awareness and experience of developers to outdated and license-violating code on stack overflow: An online survey. CoRR, abs/1806.08149. URL: http://arxiv.org/abs/1806.08149.

Raza, M., Gulwani, S., Milic-Frayling, N., 2015. Compositional program synthesis from natural language and examples. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 792–800.

Robillard, M.P., DeLine, R., 2011. A field study of API learning obstacles. Empir. Softw. Eng. 16 (6), 703–732. http://dx.doi.org/10.1007/s10664-010-9150-8.

Shen, Q., Xie, B., Zou, Y., Zhu, Z., Wu, S., 2019. NLI2Code: Reusing libraries with natural language interface. In: Peng, X., Ampatzoglou, A., Bhowmik, T. (Eds.), Reuse in the Big Data Era - 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26-28, 2019, Proceedings. In: Lecture Notes in Computer Science, vol. 11602, Springer, pp. 168–184. http://dx.doi.org/10.1007/978-3-030-22888-0_12.

Treude, C., Robillard, M.P., 2016. Augmenting API documentation with insights from stack overflow. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. pp. 392–403. http://dx.doi.org/10.1145/2884781.2884800.

Treude, C., Robillard, M.P., 2017. Understanding stack overflow code fragments. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017. IEEE Computer Society, pp. 509–513. http://dx.doi.org/10.1109/ICSME.2017.24.

Treude, C., Robillard, M.P., Dagenais, B., 2015. Extracting development tasks to navigate software documentation. IEEE Trans. Softw. Eng. 41 (6), 565–581. http://dx.doi.org/10.1109/TSE.2014.2387172.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D., 2013. Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013. pp. 319–328. http://dx.doi.org/10.1109/MSR.2013.6624045.

Xie, T., Pei, J., 2006. MAPO: mining API usages from open source repositories. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006. pp. 54–57. http://dx.doi.org/10.1145/1137983.1137997.

Yan, X., Han, J., 2002. GSpan: Graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan. pp. 721–724. http://dx.doi.org/10.1109/ICDM.2002.1184038.

Yang, D., Hussain, A., Lopes, C.V., 2016. From query to usable code: an analysis of stack overflow code snippets. In: Kim, M., Robbes, R., Bird, C. (Eds.), Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016. ACM, pp. 391–402. http://dx.doi.org/10.1145/2901739.2901767.

Yessenov, K., Kuraj, I., Solar-Lezama, A., 2017. DemoMatch: API discovery from demonstrations. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 64–78. http://dx.doi.org/10.1145/3062341.3062386.

Zamanirad, S., Benatallah, B., Barukh, M.C., Casati, F., Rodríguez, C., 2017. Programming bots by synthesizing natural language expressions into API invocations. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. pp. 832–837. http://dx.doi.org/10.1109/ASE.2017.8115694.

Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M., 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 886–896. http://dx.doi.org/10.1145/3180155.3180260.

Zhang, T., Yang, D., Lopes, C., Kim, M., 2019. Analyzing and supporting adaptation of online code examples. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. pp. 316–327. http://dx.doi.org/10.1109/ICSE.2019.00046.

Zhong, H., Zhang, L., Xie, T., Mei, H., 2009. Inferring resource specifications from natural language API documentation. In: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009. pp. 307–318. http://dx.doi.org/10.1109/ASE.2009.94.

**Qi Shen** is a Ph.D. candidate at the Software Institute, Peking University, under the supervision of Prof. Bing Xie and Associate Prof. Yanzhen Zou. His research interests include software reuse and program synthesis. He got a bachelor degree in computer science from Peking University in June, 2016.

**Shijun Wu** is a Ph.D. candidate at the Software Institute, Peking University. His main research interests are software engineering and human computer interaction. He got a bachelor degree in computer science from Peking University in June, 2019.

**Yanzhen Zou** received her Ph.D. degree from Peking University in 2010. She is currently an associate professor in the School of Electronics Engineering and Computer Science, Peking University. Her research interests focus on software engineering and information retrieval.

**Zixiao Zhu** is currently a research scientist at IBM Research — China lab. He received the B.Eng. degree in software engineering from East China Normal University, Shanghai, China, in 2011, and the Ph.D. degree in computer software and theory from Peking University, Beijing, China, in 2018. His research interests include unstructured data mining and analytics, AIOps, log analytics, knowledge graph, and software reuse.

**Bing Xie** is a profess or in the School of Electronics Engineering and Computer Science, Peking University. He is leading the Software Institute at Peking University. He received his Ph.D. degree from the National University of Defense Technology. He has more than 80 publications at major conferences in software engineering including FSE and POPL. His research interests include software reuse and formatting system.