CodeKernel: A Graph Kernel based Approach to the Selection of API Usage Examples

Xiaodong Gu[1,3], Hongyu Zhang[2], Sunghun Kim[1,3]
[1]Hong Kong University of Science and Technology, Hong Kong
[2]The University of New Castle, Australia [3]Clova AI Research, NAVER Corporation
[1]guxiaodong1987@126.com, hunkim@cse.ust.hk
[2]hongyu.zhang@newcastle.edu.au

***Abstract*—Developers often want to find out how to use a certain API (e.g., *FileReader.read* in JDK library). API usage examples are very helpful in this regard. Over the years, many automated methods have been proposed to generate code examples by clustering and summarizing relevant code snippets extracted from a code corpus. These approaches simplify source code as method invocation sequences or feature vectors. Such simplifications only model partial aspects of the code and tend to produce inaccurate examples.**

We propose CodeKernel, a graph kernel based approach to the selection of API usage examples. Instead of approximating source code as method invocation sequences or feature vectors, CodeKernel represents source code as object usage graphs. Then, it clusters graphs by embedding them into a continuous space using a graph kernel. Finally, it outputs code examples by selecting a representative graph from each cluster using designed ranking metrics. Our empirical evaluation shows that CodeKernel selects more accurate code examples than the related work (MUSE and eXoaDocs). A user study involving 25 developers in a multinational company also confirms the usefulness of CodeKernel in selecting API usage examples.

I. Introduction

API usage examples have shown importance in many software engineering tasks such as API documentation [27], [36], [46], [51], code search [21], and code completion [9], [32]. Developers frequently need to use APIs (e.g., *FileReader.read* in JDK library) that they are unfamiliar with or do not remember how to use. It is common practice for developers to search for usage examples (i.e., sample code) to understand the APIs. The API usage examples provide exemplar code that demonstrates the typical usage of an API. Accurate and understandable code examples can help developers overcome obstacles caused by unfamiliar APIs [17], [27], [51].

Yet acquiring accurate and understandable API usage examples is difficult. The most common way is to directly read manually written examples from API documentation. However, such examples are usually insufficient, covering only a small portion of common APIs. There are a large number of APIs (e.g., JDK has 86K+ APIs), which are constantly evolving. It is time consuming for library developers to manually write examples for all of them. Furthermore, API usage examples cannot answer programming questions that are not directly related to a specific API. Another way is to search from developer Q&A forums such as Stack Overflow [4]. However, it is often difficult to find relevant code for unpopular APIs or programming tasks. The answers could be either too general or too detailed, and might not be up-to-date [36]. Developers could also exploit code examples using code search tools such as GitHub Search [1]. Yet, the accuracy of answers is highly dependent on the search engine. Users may encounter too many project-specific code snippets extracted from open source projects. For example, a search of "*FileReader.read*" over GitHub returned 93,691 Java code snippets. It would be time-consuming to explore a large number of project-specific code snippets to understand how to implement this functionality. Therefore, it is desirable to be able to automatically select a small yet effective code example.

Many approaches have been proposed to generate API usage examples from a code corpus [10], [21], [35], [46], [51]. For example, MAPO [51] and UP-Miner [46] abstract code snippets into method call sequences and mine usage patterns by clustering similar sequences and mining frequent patterns in each cluster. Kim et al. [21] proposed eXoaDocs which approximates code snippets as AST element vectors. These vectors are clustered and ranked according to their vectorial similarities. The API usage examples are then selected from the clusters. The aforementioned approaches simplify source code as method call sequences or feature vectors. Such simplifications only model partial aspects of the source code. The structural information of the code such as control structures and data dependency is lost. Therefore, these source code representations could lead to imprecise code similarity measurement. The resulting code examples are often inaccurate and difficult for developers to reuse in programming practice. Nguyen et al. proposed GrouMiner [35], which is a graph-based approach to mine API usage patterns [32]. However, it utilizes frequent pattern mining, which tends to produce redundant results. Moreno et al. [27] proposed MUSE, which applies a code clone detection technique (Simian [2]) to group code snippets and select code examples. MUSE could produce redundant code examples as it is based on text-based clones of code [43] instead of a source code abstraction. We will describe more about the limitations of the existing approaches in Section II.

To address the limitations of existing approaches, we propose a novel approach called CodeKernel, which is a graph

kernel based approach to the selection of API usage examples from relevant code corpus. CodeKernel has two distinctive characteristics:

- First, instead of abstracting source code into method call sequences [46], [51], feature vectors [21] and raw code [27], CodeKernel represents source code as object usage graphs [35]. An object usage graph can be seen as an abstraction of source code. It abstracts away syntactical details a raw code representation brings, but it keeps complete information about code such as texts, structures, sequences, and data dependencies.
- Second, instead of using frequent pattern mining [35] or similarity heuristics [10], [21], CodeKernel clusters the similar graphs through graph kernel [7], [8] which embeds the graphs into a high-dimensional continuous space. Such an embedding conserves full aspects of the original graphs [6], thus is more accurate than methods that are based on feature extraction or similarity heuristics.

Given a code corpus (which consists of code snippets from open source projects), CodeKernel first builds object usage graphs for each function. It then clusters the graphs through graph embedding. Finally, CodeKernel selects the representative graph of each cluster using ranking metrics.

We empirically evaluated the accuracy of CodeKernel on 34 Java APIs. Our results show that CodeKernel's code clustering achieves an average F1-score of 0.79, outperforming two state-of-the-art approaches (MUSE [27] and EXOADOCS [21]). In a human study involving 25 developers in a multinational company, 69% of our code examples were preferred over the state-of-the-art technique, and 95% developers considered CodeKernel useful for selecting API usage examples. The results confirm the accuracy and usefulness of CodeKernel in programming practices.

The main contributions of our work are as follows:

- To our knowledge, we are the first to apply a graph kernel method to source code, which leads to more accurate code examples than the state-of-the-art techniques.
- We develop CodeKernel, a tool that generates API usage examples. Our evaluation confirms the accuracy and usefulness of the selected code examples.

II. BACKGROUND AND MOTIVATION

In this section we show the motivation behind our approach by reviewing the limitations of the state-of-the-art approaches.

A. Call Sequence based Methods

A number of techniques such as MAPO [51] and UP-Miner [46] represent source code as method call sequences. Figure 1 shows a screenshot of UP-Miner [46], an API usage pattern mining approach based on call sequences. The example in Figure 1 shows that, when both *SqlConnection.new* and *SqlConnection.createCommand* occur, it is highly probable that the API *SqlConnection.Open* will occur next. For a set of call sequences that include an API method, UP-Miner first performs clustering of the call sequences. It then mines API

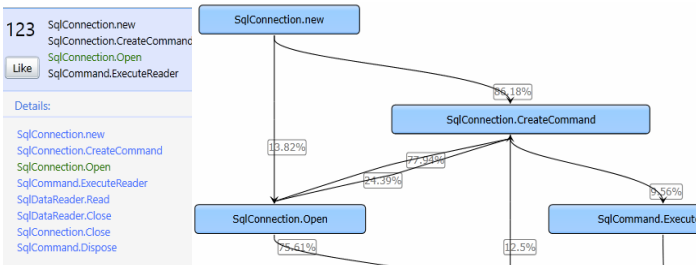

Fig. 1: an API usage pattern mined by UP-Miner

usage patterns from each cluster using a frequent sequence mining algorithm, and performs clustering again to group the frequent sequences into patterns. Given a usage pattern, UP-Miner also returns a list of code examples that contains the pattern, ranked by their similarity value.

Although call sequences can characterize API usage patterns, they fails to capture the structural information of the source code such as loops, branches and third-party method invocations. Missing such information could cause inaccurate calculation of code similarity, thus producing inaccurate API usage patterns.

B. Feature Vector based Methods

Instead of abstracting source code as API sequences, a considerable number of existing approaches use feature extraction and similarity heuristics for source code [10], [21]. One typical approach EXOADOCS [21] approximates the semantic features of code as AST element vectors before clustering. The AST element vector characterizes a fragment as occurrence counts of single AST node types. However, it is also insufficient to capture structural information [33]. Figure 2 shows two code fragments of different structures [33]. Unfortunately, they have very similar AST element vectors (e.g., both fragments have four "type declarations", one "for statement", one "if statement" and four "identifier names" in their AST trees). They even have similar identifiers ("x" and "n") and data types ("int"). Thus they would be incorrectly clustered by EXOADOCS's vector-based approach.

```
int sum (int x, n){
   int s = 0;
   for(int i = x; i<n; i++)
      if (i%2==0)
        s = s + i;
   return s;
}
```

```
int power (int x, n){
   int p = 0;
   p = 1;
   for( int i = x; i<n; i++)
      p = p * x;
   return p;
}
```

Fig. 2: Different fragments with similar element vectors of single node types [33]

C. Code Clone based Methods

MUSE [27] is a typical code example selection approach which utilizes program slicing and text-based clone detection technology. However, methods based on text based clone detection could produce many redundant examples, as they detect type-1 and type-2 clones [43] over raw code instead

Fig. 3: Patterns minined by GrouMiner for the API *StringUtils.isBlank*

of an abstraction. The similarity measures between code snippets could be adversely affected if the example contains too much information that is specific to local context. Figure 4 shows code examples produced by MUSE for the Java API *FileUtils.writeStringToFile*, which are directly extracted from its website[1]. As we can see, the examples selected by

Example 1

```
final FileInfo template;
final FileInfo filter;
final String outputBasePath;
String outputDir = createOutputDirectory(
                                template,filter,outputBasePath);
final String templateFilename = template.getFile().getName();
final String outputFilename = FilenameUtils.separatorsToSystem(
                                outputDir + templateFilename);
final String rawTempl = FileUtils.readFileToString(template.getFile());
final Properties properties = readFilterIntoProperties(filter);
final String processedTemplate = StrSub.replace(rawTemplate,properties);
//newFile(outputFilename) -> the file to write
//processedTemplate -> the content to write to the file
FileUtils.writeStringToFile(new File(outputFilename),processedTemplate);
```

Example 6

```
public JobManagerConfiguration jobManagerConfiguration;
StringWriter results = new StringWriter();
File tempPBSFile = null;
String scriptContent = results.toString().replaceAll("^[ |\t]*\n$","");
if(scriptContent.startsWith("\n"))
{
    scriptContent = scriptContent.substring(1);
}
int number = new SecureRandom().nextInt();
number = (number < 0 ? -number : number);
tempPBSFile = new File(Integer.toString(number)
                + jobManagerConfiguration.getScriptExtension());
//tempPBSFile -> the file to write
//scriptContent -> the content to write to the file
FileUtils.writeStringToFile(tempPBSFile,scriptContent);
```

Example 10

```
/**
* Location of repository.
*/
private final transient String path;
@NotNull final String name;
@NotNull final String content;
final File dir = new File(this.path);
final File file = new File(dir,name);
//file -> the file to write
//content -> the content to write to the file
FileUtils.writeStringToFile(file,content);
```

Fig. 4: Parts of usage examples for the Java API *FileUtils.writeStringToFile* selected by MUSE [27]

MUSE contain much redundancy. In their results, example 1, 6 and 10 are presented as different examples because they prepare file names (String) and contents (String) in different

[1]https://github.com/lmorenoc/icse15-muse-appendix/blob/master/commons-io-2.4/examples/writeStringToFile_29.html

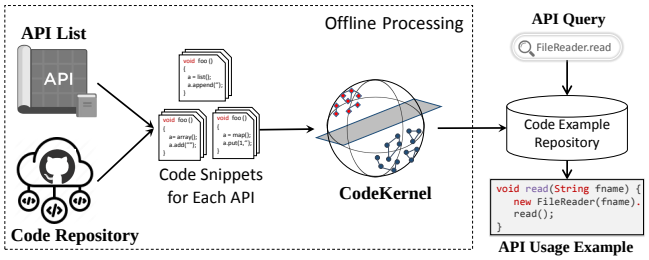

Fig. 5: The application of CodeKernel to the selection of API usage examples

ways. Example 1 reads a string content, and replaces it with a filter. Example 6 removes special characters from a string. Example 10 directly writes a string. However, from developers' perspective, different ways of preparing contents (e.g., *String.replaceAll*, *String.substring*) are not related to the API usage. They are specific to local context. In fact, all the examples follow the same usage of the API: creating a file with a filename (String), preparing a context (String), then invoking the API *FileUtils.writeStringToFile* to write the context to the file. Clone detection techniques are often specific to such local context and could produce redundant examples. A better approach should treat these examples as similar ones and merge them into one example.

D. Graph based Frequent Pattern Mining

Representing source code as graphs could alleviate the aforementioned problems as graphs are effective to carry structural information. GrouMiner [35] is a typical graph-based approach that is successful for mining API usage patterns [32]. However, it is based on frequent pattern mining which tends to suffer from the "high redundancy" problem, that is, patterns could be subsets of other larger patterns [5], [46]. Figure 3 shows patterns returned by GrouMiner for the Java API *StringUtils.isBlank* in the same code corpus of MUSE [27]. More results can be found at http://codekernel19.github.io/codekernelpre/preliminary.html. We can see that many patterns are redundant: Pattern 22 is a subset of Pattern 23, and Pattern 12 is a subset of Pattern 27. Such redundancy incurs extra effort for developers in finding patterns of interest. This indicates that more improvement is required for graph-based approaches, and graph clustering could be a better choice than frequent pattern mining.

III. Approach

Our primary lever for selecting code examples is to cluster source code according to their usages and select typical snippets from clusters. The limitations of the existing approaches motivate us to develop a novel approach that allows manipulation of structural data accurately while being computationally cheap [6], [44]. Our approach, named *CodeKernel*, uses graphs to model source code and directly clusters original graphs using the graph kernel method [28].

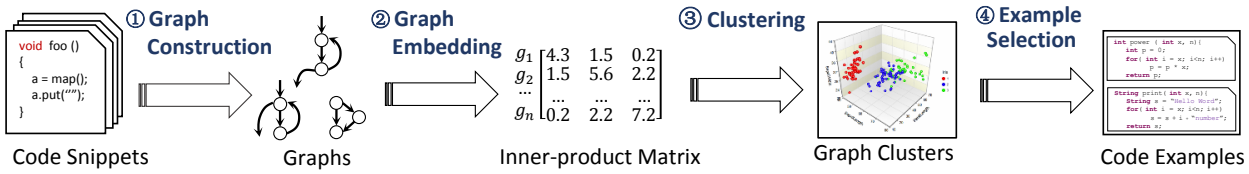

Fig. 6: The workflow of CodeKernel

Figure 5 illustrates the application scenario of CodeKernel. The offline processing is responsible for selecting code examples. It gathers relevant code snippets for each API and selects API usage examples using CodeKernel. At runtime, for a given API query (such as $FileReader.read$), the relevant code examples are identified and presented to users. The overall pipeline of CodeKernel is shown in Figure 6. It takes as input raw code corpus (i.e., code snippets from open source projects or code search results) and outputs code examples. The raw code is first transformed into object-usage graphs [35]. Then, graphs are embedded into a continuous space using a graph kernel method, resulting in an inner product matrix. CodeKernel clusters the graphs in the new space by applying a clustering algorithm to the inner product matrix. Finally, the representative graph of each cluster is selected with ranking metrics and recovered as code examples. These procedures are offline and the selected code examples are returned in response to user's queries (Figure 5).

We describe our approach using pseudo code in Algorithm 1. The details are explained in the following sections.

A. Graph Representation for Source Code

We choose to use a graph model which contains information about text, sequence, structures, and data dependencies so that it is capable of representing the full aspects of source code. Meanwhile, it ignores syntactical details so that it is not sensitive to local contexts. In particular, in our approach we choose to use the object usage graph [35], which is a graph model for source code. Object usage graph has proven to be successful in many software engineering tasks such as object usage pattern mining [35], code completion [32] and API recommendation [31]. An object usage graph is a directed acyclic graph defined as $G = (V, E)$ where V stands for a set of nodes (controls, actions and data) and $E \subseteq V \times V$ denotes a set of edges representing call sequences or data dependencies [35]. Each node has a label which is a class/method name or a control unit [31].

Figure 7 illustrates an example of an object usage graph. The action nodes such as *StringBuffer.new* and *BufferedReader.readLine* stand for method calls or field accesses. The data nodes such as *StringBuffer* and *BufferedReader* represent objects of a class. The control nodes such as *while* represent controls for branches or loops. There are two types of edges, sequential edges and data edges. Sequential edges connect nodes who have strict orders among them. For example, *BufferedReader.new* must be executed before *BufferedReader.readLine*. Data edges connect a data node with

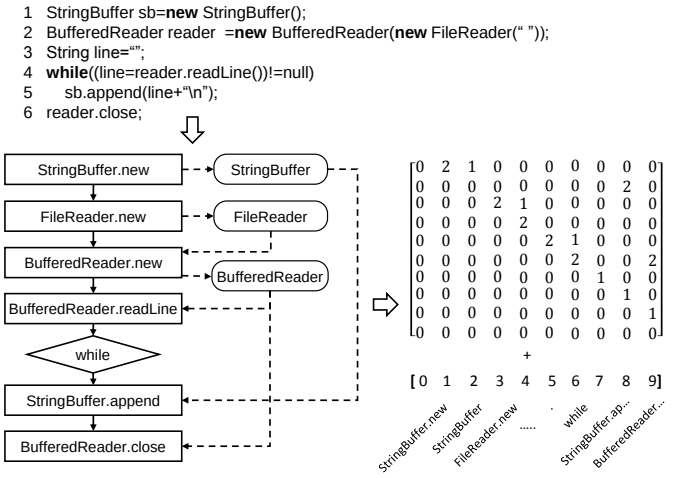

Fig. 7: Schematic illustration of graph generation from source code

action nodes if the action node uses objects or parameters of the data node. For example, both *BufferedReader.readLine* and *BufferedReader.close* use the object *br* which is defined as a data node *BufferedReader*, therefore, both of them are connected with the data node.

To ease further computation, we represent each graph as an adjacency matrix accompanied with a label vector. The adjacency matrix is an $n \times n$ matrix, where n is the number of nodes and each entry is the edge type between the corresponding nodes. We set the entry to 0 if there is no edge between two nodes, 1 if there is a sequential edge, or 2 if there is a data edge. The label vector $\ell = \langle c_1, ..., c_n \rangle$ is an n-dimensional vector, where n is the number of nodes in the graph, and each c_i is the global index of the label for the i-th node.

In our experiments, we generate object usage graphs by applying GrouMiner [35] at the method level (Line 1, Alg. 1).

B. Graph Embedding

Our next goal is to compute graph similarities for clustering (Line 3-5, Alg. 1). Instead of extracting graph features [33] such as AST element vectors [21], n-grams [46], [31], and statement sequences [10], we try to directly manipulate the graphs. Specifically, we embed the original graphs into a high-dimensional, continuous space where their inner products can

Algorithm 1 High-level pseudo code of **CodeKernel**

Input:
Code Corpus *Corp*
Output: Code Examples *examples*
1: *graphs*← BuildGraphs(*Corp*)
2: **let** $K_{n\times n} \leftarrow [0]_{n\times n}$
3: **for all** graph pair $< g_i, g_j > \in$ *graphs* **do**
4: $K_{i,j} \leftarrow$ GraphKernel(g_i, g_j)
5: **end for**
6: *clusters*← SpectralClustering(K)
7: **let** *examples*← ∅
8: **for all** C∈*clusters* **do**
9: repr←SelectRepr(C,K)
10: *examples*←*examples*∪ CodeRecover(C,repr)
11: **end for**
12: *examples*←Rank(*examples*)
13: **return** *examples*

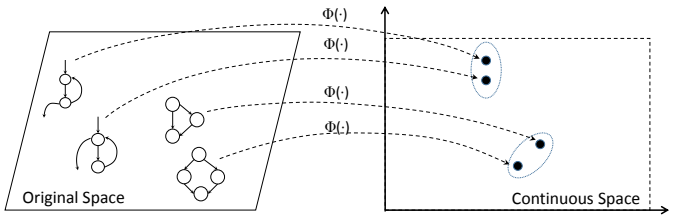

Fig. 8: Illustration of the kernel-based graph embedding

be calculated accurately and is computationally cheaper. Then, conventional clustering methods can be applied directly to the embedded data points.

The kernel method is an efficient and well-studied approach to achieve such embedding [6], [44]. Figure 8 illustrates the basic idea. Suppose we have data in a space whose coordinates are too difficult or expensive to compute (e.g., sequential data or graphs). We then assume that there exists a transformation function $\Phi : S \rightarrow T$ that maps data in the original space S into a continuous space T. As we do not know the explicit formulation of Φ, the transformed data in space T is still implicit. Fortunately, there exists an important principle that, the inner products of data in the space T can be calculated simply by a kernel function defined in the source space S [44]. That is, $k(g_1, g_2) = \Phi(g_1) \cdot \Phi(g_2)$. A kernel function $k : S \times S \rightarrow \mathbb{R}$ is a function that satisfies mercer's conditions [44]. It allows us to operate non-vectorial data in a continuous space by simply defining a function on the original data. The embedding conserves full aspects of the original data, thus is more accurate than methods that extract features from data instances [6]. Kernel functions have been introduced for sequence data, graphs, text, images, as well as vectors [8], [14], [49]. Most commonly used kernel functions includes Gaussian kernel [6], [44], linear kernel [6], [44] and polynomial kernel [6], [44].

Especially, for graph data, there is a class of kernel function named graph kernel [7], [8]. Graph kernels are kernel functions that compute inner products on graphs [8]. They can be intuitively understood as functions measuring the similarities of pairs of graphs. They allow kernelized learning algorithms such as support vector machines to work directly on graphs, without having to do feature extraction to transform them to fixed-length, real-value feature vectors. Graph kernels have seen successful applications in many areas such as chemoinformatics (e.g., molecule kernels [41]), bioinformatics [8], and social network analysis [45].

In our approach, we adopt a highly efficient and widely used graph kernel, the shortest path graph kernel [7]. Given two graphs, G_1 and G_2, their kernel is defined as:

$$k(G_1, G_2) = \sum_{e_1 \in SD(G_1)} \sum_{e_2 \in SD(G_2)} k_{\text{walk}}(e_1, e_2) \quad (1)$$

where $SD(G)$ is a new graph having the same nodes as G. Each edge $e = (u, v)$ in $SD(G)$ is a new edge with a weight as the shortest distance between u and v in the original graph G. The path kernel $k_{walk}(e_1, e_2)$ is defined as:

$$k_{walk}(e_1, e_2) = k_{node}(u_1, u_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(v_1, v_2) \quad (2)$$

where k_{node} and k_{edge} are kernel functions for comparing two nodes or edges. We define node kernel as:

$$k_{node}(u_1, u_2) = \begin{cases} 1, & \text{if label}(u_1)=\text{label}(u_2), \quad (3) \\ 0, & \text{otherwise.} \quad (4) \end{cases}$$

which means that we assign a kernel value of one to two nodes with identical labels, and assign zero to two nodes with different labels. We define an edge kernel as a Brownian bridge kernel [8] of edge weights. The Brownian bridge kernel has shown good performance in many graph kernel studies [7], [8]. It is defined as:

$$k_{edge}(e_1, e_2) = max(0, c - |w(e_1) - w(e_2)|) \quad (5)$$

where w returns the weight of an edge, and c is a positive constant. The kernel means that we assign the highest kernel value to edges that are identical in weight and assign zero to edges that differ in weight by more than a constant c. We empirically set c = 2 as it was validated and performed well in [7], [8].

For each pair of graphs, we compute their kernel value using Equation 1. Finally, we get a positive definite kernel matrix $K_{n\times n}$ where n is the number of graphs. Kernel matrix is also known as inner-product matrix, which can be viewed as a similarity matrix. It represents pair-wise inner products of graphs in the new continuous space. The inner-product matrix can be directly manipulated by machine learning algorithms such as classification [8], [11] and clustering [24]. For the implementation, we adopted Borgwardt's code in Matlab [45].

C. Graph Clustering

After embedding graphs into continuous space, our next step is to cluster graphs in the new space (Line 6, Alg. 1).

We adopt a typical clustering algorithm in machine learning, namely spectral clustering [12]. The most important reason we choose Spectral Clustering is that it suits our data well. In our problem, data in the continuous space is not vectorial.

Therefore, algorithms that require vectorial input such as K-means, Gaussian Mixture Model and EM are not applicable. Spectral clustering, on the other hand, is based on similarity pairs instead of vectors. The pairwise inner products embedded in the new space are exactly suitable for the algorithm. Spectral clustering also often outperforms other clustering algorithms in many domains [12], [50].

In our approach, the spectral clustering algorithm takes as input the inner product matrix generated by graph embedding and performs clustering. We adopt a tool named *Spectral Clusterer for Weka* [3] in our implementation.

D. Example Selection

After clustering, CodeKernel selects code examples from clusters (Line 7-13, Alg. 1). For each cluster, it first selects a representative graph according to the designed ranking metrics. Then, it presents a code example by recovering the original code of the selected graph.

1) Rank Metrics: We design two rank metrics for selecting a representative graph for each cluster.

Centrality We first want the representative graph as common as possible in the cluster. That is to say, the graph should have high similarity to other graphs in the cluster. Inspired by a clustering algorithm, K-medoids [38], we define a metric *Centrality*, which measures the average distance from an instance to other instances in the cluster. For each graph g_i in a cluster C, the centrality is defined as:

$$\text{centrality}_i = 2 \times \text{sigmoid}(\frac{1}{|C|} \sum_{g_j \in C, j \neq i} K_{i,j}) - 1 \quad (6)$$

where $sigmoid$ is a commonly used function to normalize values to the interval of [0, 1] [18]. The higher the centrality of g_i, the more common g_i is in the cluster.

Specificity The graphs with high centralities may tend to be the larger graphs since they have more chance of being similar to others. Unfortunately, larger graphs tend to have more specific elements (i.e., edges that are rare in the cluster), making the code example difficult to understand. To penalize graphs with too many specific edges, we also design the *Specificity* metric. For each graph g_i in a cluster C, the *Specificity* is defined as:

$$\text{specificity}_i = 2 \times \text{sigmoid}(\frac{1}{|E_{g_i}|} \sum_{e \in g_i} w_{idf}(e)) - 1 \quad (7)$$

where $|E_{g_i}|$ is the number of edges in g_i, $sigmoid$ is a normalization function [18], and each $w_{idf}(e)$ is the IDF (Inverse Document Frequency) weight of an edge. We use IDF to measure the rareness of each graph edge in the cluster. For each graph edge e in a cluster C, the IDF weight is defined as

$$w_{idf}(e) = \log(\frac{|C|}{N_e}) \quad (8)$$

where $|C|$ is the cluster size, N_e is the number of times the edge e occurs in the whole cluster C. The more specific the edges, the higher specificity a graph has.

2) Representative Graph Selection: With the two ranking metrics, we select a representative graph from each cluster. We first define a ranking score for each instance in a cluster as:

$$\text{score} = \text{centrality} - \gamma \cdot \text{specificity} \quad (9)$$

where γ stands for a parameter to control the penalty of specificity. We empirically set γ=0.2. Then, we rank instances in a cluster according to their ranking scores and select the graph with the largest score as the representative graph.

Finally, for the selected graphs from clusters, we recover their original code and rank them according to the sizes of clusters they belong to. The examples from larger clusters are ranked with higher priorities than those from smaller ones.

IV. Empirical Evaluation

We evaluate our framework from two perspectives: accuracy and usefulness. Specifically, our evaluation addresses the following research questions:

- **RQ1: How accurate are the API usage examples selected by CodeKernel?**
- **RQ2: How useful is CodeKernel for selecting API usage examples?**
- **RQ3: Does graph kernel help improve the graph clustering performance ?**

A. Accuracy of Selected Examples(RQ1)

Accuracy is the key aspect for the evaluation of API example selection. Inaccurate examples could have large redundancy and low recall. Hence, developers have to examine a large number of results to find useful API examples.

1) Accuracy of Code Clustering: We first evaluate CodeKernel's accuracy in code clustering, namely, assigning relevant code snippets to the same example. This is important because it determines the succinctness and recall of final examples.

To evalate the clustring accuracy, we selected a few typical Java APIs, run CodeKernel for each API, and compare the clustering accuracy against the baseline methods. Table I lists the selected APIs for RQ1 and their statistics. They are widely used in the corpus provided by our baseline methods. The colunm Code Corpus shows the code corpus that the API usage examples are selected from. The # snippets column shows the number of methods in the code corpus that use the corresponding API. The last column # API usage shows the number of usages of each API in the code corpus. They are determined according to our manual labels to be introduced.

Accuracy Measure: We measure the clustering accuracy with F1-score. The F1-score is a widely used accuracy measure for clustering in the data mining literature [25], [26], [39], [42]. It treats clustering results as a series of decisions, one for each of the $N(N-1)/2$ pairs of the instances [25]. For example, if there are 4 snippets $\{s_1, s_2, s_3, s_4\}$, which belong to cluster {A, A, B, C}, respectively. To evaluate a clustering method, we compare 4×3/2=6 times, for the pairs $\langle s_1,s_2\rangle$, $\langle s_1,s_3\rangle$, $\langle s_1,s_4\rangle$, $\langle s_2,s_3\rangle$, $\langle s_2,s_4\rangle$ and $\langle s_3,s_4\rangle$. If a clustering method outputs {A, A, A, C}, we can see that the pairs $\langle s_2,s_3\rangle$ and $\langle s_1,s_3\rangle$ are

TABLE I: Summary of selected APIs for evaluating the accuracy of code clustering (RQ1-task1)

Selected APIs	Library	Code Corpus	# of snippets	#API usages
FileUtils.writeStringToFile	commons-io2.4	86 projects used in [27]	12	4
IOUtils.toString			23	11
FilenameUtils.normalize			6	2
FileUtils.forceMkdir			8	2
IOUtils.toByteArray			10	5
StringUtils.isBlank	commons-lang3	53 projects used in [27]	25	2
StringUtils.isNotBlank			24	1
Servant._poa	CORBA	top 200 results by [21]	78	8
Window.pack	java.awt		48	12
Driver.connect	java.sql		40	13
Properties.loadFromXML	java.util		32	11
PrinterJob.pageDialog	java.awt		67	11
Graphics2D.fill	java.awt		51	14
SelectableChannel.register	java.nio		49	7

grouped incorrectly. A clustering algorithm aims to assign two snippets to the same cluster if and only if they are similar. A true positive (TP) decision assigns two similar snippets to the same cluster whereas a true negative (TN) decision assigns two dissimilar snippets to different clusters. There are two types of errors it can make. A false positive (FP) decision assigns two dissimilar snippets to the same cluster. A false negative (FN) decision assigns two similar snippets to different clusters. F1-score is defined as:

$$F1 = \frac{2 \times P \times R}{P + R} \tag{10}$$

where $P = \frac{TP}{TP+FP}$ and $R = \frac{TP}{TP+FN}$ [25]. The P-value measures the precision of assigning snippet pairs to clusters. A higher precision means less FPs, indicating that a smaller number of dissimilar snippets are assigned to the same cluster. Therefore, a higher P-value indicates higher coverage of clustering. The R-value measures the recall of cluster assignments of snippet pairs. A higher recall means less FNs, which indicates that a smaller number of similar snippets are assigned to different clusters. Therefore, a higher R-value indicates less redundancy in clustering.

To evaluate the accuracy of the clustering methods, we need the ground truth clusters for each API. In our experiments, we manually labeled ground truth clusters for the raw code snippets that contain the APIs under study. To reduce the labeling bias, two developers independently labeled examples in the original corpus. Then, they discussed for disagreements and relabeled again until agreements are reached.

Baselines: We compare the accuracy of our approach against MUSE [27] and EXOADOCS [21]. Clone detection is a widely studied work utilizing code similarity measure and MUSE is a successful clone-based approach for code example selection. EXOADOCS is the state-of-the-art code example selection approach which clusters and ranks code snippets with similarity heuristics such as distances between AST element vectors. As we cannot obtain the original implementation of MUSE and EXOADOCS[3], to facilitate comparison, we collected the code corpus stated in their papers as well as the raw results produced by their tools[4]. We then run CodeKernel to select code example for the selected APIs (Table I) from the same code corpus provided by each paper. Finally we compare our results with the published code examples of MUSE and EXOADOCS. For MUSE, as we can only obtain the selected code example for each cluster from its published results, we cannot compute the exact P, R and F1 values. To this end, we make a relaxation by assuming that all the missing examples are assigned to a correct group. Specifically, we assign missing code examples to the corresponding groups according to their ground truth labels. Therefore, the P, R and F1 values we compare to are upper bounds.

[3]The EXOADOCS website was down and the authors no longer maintain the code, but one of them kept the code corpus as well as the raw results.

TABLE II: F1-scores of CodeKernel and MUSE

API	MUSE			CodeKernel		
	P	R	F1	P	R	F1
FileUtils.writeStringToFile	$\leq$0.60	$\leq$0.21	$\leq$0.31	0.68	0.45	0.54
IOUtils.toString	$\leq$0.51	$\leq$0.53	$\leq$0.52	0.80	0.56	0.66
FilenameUtils.normalize	0	0	0	0.60	0.60	0.60
IOUtils.toByteArray	$\leq$0.80	$\leq$0.44	$\leq$0.57	1.0	0.67	0.80
FileUtils.forceMkdir	$\leq$1.0	$\leq$0.38	$\leq$0.55	0.71	0.94	0.81
StringUtils.isBlank	$\leq$1.0	$\leq$0.51	$\leq$0.67	0.92	1.00	0.96
StringUtils.isNotBlank	$\leq$1.0	$\leq$0.44	$\leq$0.61	1.0	1.0	1.0
Average	$\leq$0.70	$\leq$0.36	$\leq$0.46	0.82	0.75	0.77

TABLE III: F1-scores of CodeKernel and EXOADOCS

API	EXOADOCS			CodeKernel		
	P	R	F1	P	R	F1
Servant._poa	0.58	0.48	0.53	0.92	0.97	0.94
Window.pack	0.49	0.80	0.61	0.82	0.93	0.87
Driver.connect	0.42	0.85	0.56	0.90	0.99	0.94
Properties.loadFromXML	0.08	0.34	0.13	1.0	0.55	0.71
PrinterJob.pageDialog	0.21	0.83	0.34	0.92	0.94	0.93
Graphics2D.fill	0.12	0.79	0.20	0.74	0.53	0.62
SelectableChannel.register	0.28	0.62	0.38	1.0	0.46	0.63
Average	0.31	0.67	0.39	0.90	0.77	0.81

Results: Table II and III show the accuracy results of MUSE, EXOADOCS and CodeKernel. As the results indicate, CodeKernel produces code examples with higher coverage and less redundancy, with average P and R values of 0.86 and 0.76 respectively, which are greater than those of MUSE (P$\leq$0.7, R$\leq$0.36) and EXOADOCS (P=0.31, R=0.67). Overall, CodeKernel outperforms MUSE and EXOADOCS for all studied APIs, with an average F1-score of 0.79, which is significantly greater than that of MUSE (0.46) and EXOADOCS (0.39). The results confirm the effectiveness of the clustering method used by CodeKernel.

2) Accuracy of Example Selection: We also evaluate the accuracy of CodeKernel in selecting representative examples from each code cluster(Section III-D2). As the ranking of code examples could be subjective, we conducted a user study to evaluate the accuracy. The user study involved 25 developers in a multinational company M, all having more than 2 years of programming experiences.We randomly selected 10 Java

[4]MUSE published an API documentation in their website https://github.com/lmorenoc/icse15-muse-appendix. The documentation includes their raw results of API examples. Results and code corpus of EXOADOCS are provided by its authors, and are the same as what used in their paper [21].

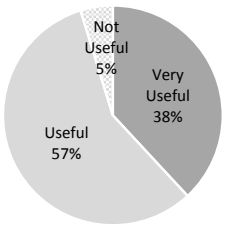

Fig. 9: The results of user study

APIs that are not too simple nor too common[5]. Participants were asked to read API examples selected by CodeKernel as well as the code snippets in the corresponding clusters where the examples were selected. Then, they were asked to rate the representativeness of the selected representative example in each cluster. Possible answers fall in a five-point Likert scale (5 very accurate, 4 accurate, 3 neither, 2 inaccurate, and 1 totally inaccurate).

The results show that developers gave high ratings for the accuracy. 94% developers graded a high accuracy (with a score of 4 or 5). The average rating score was 4.1, indicating an overall positive feedback.

The code examples selected by CodeKernel have less redundancy and higher coverage than those selected by the state-of-the-art techniques, and are representative.

B. Evaluation of the Usefulness of the Selected API Usage Examples(RQ2)

We conducted a user study to investigate developers' perceived usefulness of API usage examples selected by CodeKernel. The study involved the same participants as described in Section IV-A2. It consists of two tasks on 20 randomly selected APIs[6]:

Task 1: (Questionnaire) Each participant was required to read API usage examples selected by CodeKernel[6]. These APIs were selected randomly from those who have examples in JDK or in popular tutorial websites. Then, they were required to answer the following question about the usefulness of the examples: *Overall, are the selected examples useful for understanding API usages?* It has five answer options (5 very useful, 4 useful, 3 neither, 2 not useful and 1 totally not useful).

Figure 9 shows the statistic of developers' perceived usefulness in this task. Overall, developers gave high ratings for the usefulness. The average rating was 4.5, indicating an overall positive feedback from developers. 95% of the developers thought that CodeKernel is useful for understanding API usages. Among them, 38% strongly agreed with the usefulness. This feedback indicates that developers appreciate our CodeKernel tool.

[5]The full list of Java APIs is in our project website at https://codekernel19.github.io

[6]The studied Java APIs are in our project website at https://codekernel19.github.io/appendix.html

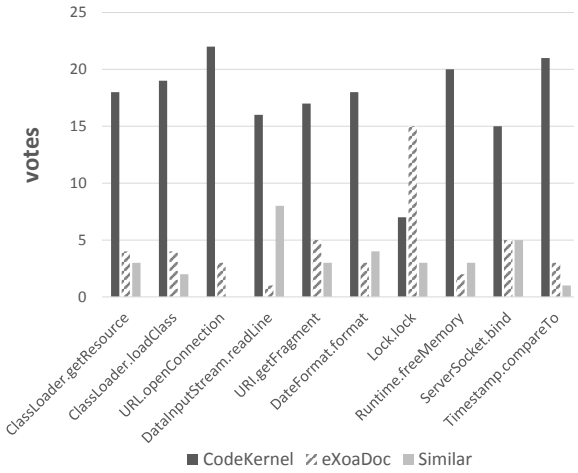

Fig. 10: Feedback on tool comparison

Task 2: (Tool Comparison) Each participant was required to read API usage examples selected by two tools: CodeKernel and EXOADOCS. We hid the names of these two tools and asked participants to evaluate 10 pairs of API usage examples[6], each corresponding to a randomly selected API. They were required to select a tool that produces better examples for the corresponding API and select "similar" if they consider both producing examples of the same quality. Figure 10 shows the statistic of tool comparison feedback. For 9 out of 10 pairs, the usage examples generated by CodeKernel are considered more useful by the developers. CodeKernel has overwhelming votes for most of the APIs. Among all the votes, 69% developers considered CodeKernel's code examples better than those of EXOADOCS, and 13% considered they were similar. Only 18% developers thought that EXOADOCS's code examples were better. The results show that developers consider CodeKernel more useful than the state-of-the-art techniques.

Developers feedback indicates that the API usage examples selected by CodeKernel are useful.

C. Graph Kernel's Performance on Graph Clustering (RQ3)

As the most distinctive feature of our approach is the graph kernel based clustering method, we also evaluate whether the graph kernel technique helps improve the graph clustering performance. To this end, we replace the graph kernel component (described in Section III-B) in CodeKernel with a component which directly measures graph similarities using a conventional similarity measure:

$$Sim(G_1, G_2) = \frac{|E_1 \bigcap E_2|}{min(|E_1|, |E_2|)} \tag{11}$$

where E is the set of edges in G. This measure is used in [22]. Basically, this equation measures the ratio of common edges of two graphs.

We compare the clustering performance of both schemes, that is, CodeKernel with graph kernel and CodeKernel using

TABLE IV: F1-scores of code clustering by different graph similarity methods

API	Baseline			Graph Kernel		
	P	R	F1	P	R	F1
FileUtils.writeStringToFile	0.72	0.45	0.55	0.68	0.45	0.54
IOUtils.toString	0.44	0.19	0.27	0.80	0.56	0.66
FilenameUtils.normalize	0	0	0	0.60	0.60	0.60
IOUtils.toByteArray	1.0	0.11	0.20	1.0	0.67	0.80
FileUtils.forceMkdir	1.0	0.31	0.48	0.71	0.94	0.81
StringUtils.isBlank	0.92	1.0	0.96	0.92	1.00	0.96
StringUtils.isNotBlank	1.0	0.25	0.40	1.0	1.0	1.0
Average	0.73	0.33	0.41	0.82	0.75	0.77

the baseline graph similarity measure. We use the same experimental setup as in RQ1.

Table IV shows the accuracy results of both schemes. As the results indicate, CodeKernel with graph kernel leads to better performance than using the baseline graph similarity measure. The graph kernel technique obtains an 88% relative improvement in terms of F1-score over the baseline method.

Graph kernel can significantly improve code clustering performance.

V. Discussion

A. An Example

We now present a concrete API usage example selected by CodeKernel. We will also discuss the limitations and present ideas for future improvement.

Figure 11 lists an excerpt of code example selected by CodeKernel for the API *FileUtils.writeStringToFile*. These results are from a cluster consisting of 6 instances. The Example 1 at the top is the selected representative of the cluster. The code snippets below (Instances 1 to 3) are instances in that cluster. These instances are clustered together as they all follow the same pattern *File.new, FileUtils.writeStringToFile*. The first instance is selected as an example as it has high similarities to other instances and does not contain many project-specific nodes. We can see improvement when comparing our examples against those selected by MUSE (Figure 4). First, all the instances we consider to be the same are clustered together by CodeKernel, which means CodeKernel can provide less redundant API usage examples to developers. In addition, the representative graph selected by CodeKernel contains less context-specific information, which means our examples are more readable.

Still, CodeKernel has limitations and could produce incomplete results. It may not show the complete data flow. For example, in Example 1 shown in Figure 11, the definition of the field reference *this.path* is not included in the sample code, developers need to browse the original source code file to understand its definition. Furthermore, the selected examples could contain project-specific identifiers and statements, such as *this.git.exec(...)* and *content*. The project-specific statements should be trimed and the name of the identifiers can be normalized. In the future, we will perform more advanced program analysis on the sample code to further improve the completeness and readability of the code. Particularly, we will investigate the synthesis of sample code directly from the selected object usage graph.

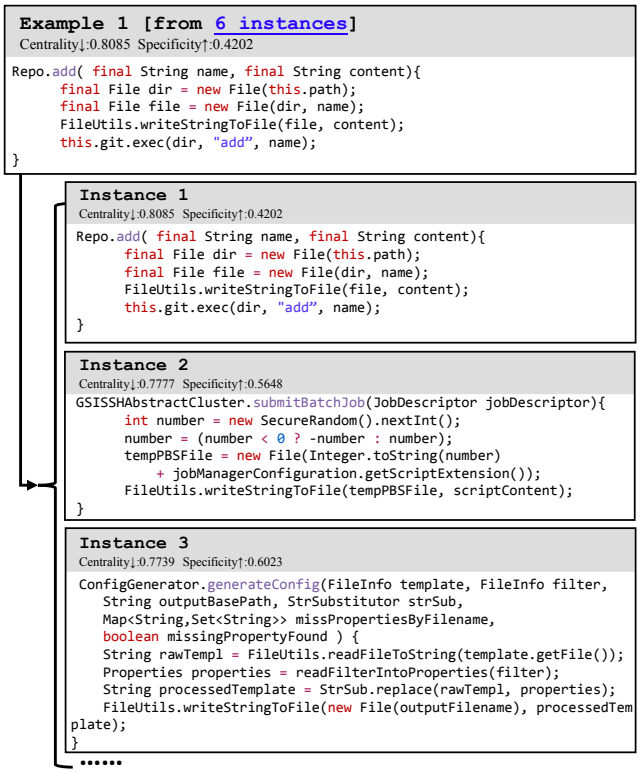

Fig. 11: The code example for the API *FileUtils.writeStringToFile* produced by CodeKernel

B. Why does CodeKernel work?

A fundamental challenge to mining source code is that, source code is not continuous data. It is discrete, structural and composite. There is no explicit coordinate and vector that can fully characterize it. Therefore, code similarities are difficult to define and compute. Existing approaches try to make it continuous either by doing feature extraction to transform it to fixed-length, real value feature vectors, or by some similarity heuristics. However, any feature vector they extracted just approximate partial information of the code (e.g., tokens [46], AST elements [21], orders [10] and topics [23]). Therefore, these approaches often lead to inaccurate code examples.

Our approach addresses such challenge by directly embedding the graph representation of source code into a continuous space without explicit feature extraction. The graph embedding conserves full aspects of the original graphs and is more accurate than methods that extract feature vectors from code.

VI. Threats to Validity

As a proof of concept, all APIs and related projects investigated in our experiments are written in Java. Although Java is one of the most popular programming languages, it might not be representative of APIs written in other languages such

as Python. However, CodeKernel is not limited to a certain language as it operates on software graphs which can be extracted from most languages. Evaluating our tool for other languages remains our future work.

In the evaluation of cluster accuracy, we compared our results with those of related methods (MUSE and eXoaDocs). However, the tool implementations of the related methods were unavailable to us. So in our comparison, we had to use the published results and datasets as the related methods used, which are relatively small in scope. In the future, we will re-implement the related methods and conduct empirical studies on more datasets to further evaluate these tools.

In our work, we perform user studies to evaluate the accuracy and usefulness of the selected API usage examples. Although our user studies involved 25 developers, the scope of the experiments is still limited. Furthermore, the participants examined a small number (10) of APIs. Therefore, our user studies could introduce bias. In the future, we will perform large-scale user studies involving more participants and APIs.

VII. Related Work

A. Code Example Selection

Code example selection has shown to be important in many software engineering tasks, such as API documentation [27], [36], [46], [51], code search [20], [21], and code completion [9], [32]. The mainstreaming technical direction to select code example is to cluster related code snippets according to some similarity heuristics, and rank or synthesize an example from each cluster. For example, Kim et al. [21] proposed eXoaDocs, which clusters and ranks code snippets according to their distance between AST element vectors. Buse and Weimer proposed to represent code snippets as CFGs, and cluster them according to their similarities of statement ordering and data type usages [10]. However, the simplified feature vectors can only approximate partial aspects of the code (e.g., tokens [46], AST elements [21], orders [10], and topics [23]). Therefore, their approaches often lead to inaccurate code examples. Different from these techniques, CodeKernel clusters graphs by embedding them to a continuous space. The graph embedding keeps full aspects of original graphs [6] and is more accurate than methods that extract feature vectors from code.

Another line of work has investigated marrying state-of-the-art code clone and sample selection techniques. For example, Moreno et al. proposed MUSE [27] that selects usage examples of a given method by slicing out relevant snippets from code corpus and identifying similar examples through text-based clone detection [43]. CodeKernel differs from MUSE in that it clusters similar code snippets at an abstract usage level.

Recently, there is also much work that utilizes statistical machine learning and deep learning [37], [29]. For example, Nguyen et al. proposed API code recommendation using statistical learning from fine-grained changes [29]. They also proposed a deep neural network language model with contexts for source code [30]. CodeKernel differs from these approaches in that it directly embeds exact graphs without learning and statistically approximating the graph features. The latter is often computational expensive and cannot represent the exact original graph.

B. Mining API Usage Pattern

Instead of selecting code example for an API, a large number of approaches focus on mining API usage patterns [34], [15]. Usage patterns are method call sequences [13], [46], [48], [51] or even statistical models [31], [36]. Xie et al. [48] proposed MAPO, which is one of the first work on mining API patterns from code corpus. MAPO represents source code as call sequences and clusters them according to similarity heuristics such as method names. It finally generates patterns by mining and ranking frequent sequences in each cluster. UP-Miner [46] is an improvement of MAPO, which removes the redundancy among patterns by two rounds of clustering of the method call sequences. Nguyen et al. [36] proposed SALAD, a statistical model to learning API usages from bytecode. Similar to CodeKernel, it represents bytecode as a graph-based model that captures method call sequences, control and data flows. It learns API usages from graphs using a Hidden Markov Model (HMM) [40]. Fowkes and Sutton [13] proposed probabilistic algorithm for mining the most informative and parameter-free API call patterns.

While such sequential or statistical patterns have shown to be useful for API recommendation and code completion, they are insufficient for developers to understand the detailed usage of the APIs. It is difficult to reuse an API usage pattern without code structures. Different from the above techniques, CodeKernel can select code examples that exhibit code structures, as it models source code as graphs instead of call sequences or statistical models.

C. Graph based Object Usage Pattern

Besides our work, there have been other work that use graph to mine API patterns [31], [35]. GrouMiner [35] introduced the concept of object usage graph, and applied it to mine object usage patterns. As indicated in Section II-D, GrouMiner could lead to redundant patterns since it is based on frequent pattern mining. CodeKernel utilizes the concept of object usage graph proposed by GrouMiner, it addresses the limitation of GrouMiner by leveraging a graph kernel based clustering technique, thus producing less redundant code examples.

Gralan [31] is another approach that uses an usage graph for mining object usage patterns. It proposes a graph-based statistical language model for code suggestion. Different from Gralan, CodeKernel mines an explicit code example from a code corpus.

VIII. Conclusion

We have proposed CodeKernel for the selection of API usage examples. Instead of approximating source code as feature vectors or sequences, we represent source code as object usage graph, and cluster the graphs by embedding them into a continuous space. Our evaluation results show

that CodeKernel provides more accurate and understandable examples than the state-of-the-art techniques. Feedback from developers is also very encouraging: 69% of our examples were preferred to the state-of-the-art technique, 95% developers considered CodeKernel useful for selecting API usage examples, and all developers considered CodeKernel useful for summarizing code search results. The code examples selected by CodeKernel can be found at our website at: https://codekernel19.github.io.

Graph embedding could also be applied to other tasks that require feature extraction on source code, such as code retrieval [16], [47] and code clone detection [19], which will be our future work. In the future, we will also investigate deep learning based techniques to further improve the completeness and readability of the code examples.

REFERENCES

[1] Github Search. https://github.com/search?type=code.

[2] S. Harris. (2003) Simian - Similarity Analyzer.[Online]. Available: http://www.harukizaemon.com/simian/.

[3] Spectral Clusterer for WEKA. http://www.luigidragone.com/software/spectral-clusterer-for-weka/.

[4] Stack Overflow. http://stackoverflow.com/.

[5] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM, 2014.

[6] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[7] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005.

[8] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. Vishwanathan, A. J. Smola, and H.-P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl 1):i47–i56, 2005.

[9] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. 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*, pages 213–222. ACM, 2009.

[10] R. P. Buse and W. Weimer. Synthesizing API usage examples. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 782–792. IEEE, 2012.

[11] C. Cai, L. Han, Z. Ji, and Y. Chen. Enzyme family classification by support vector machines. *Proteins: Structure, Function, and Bioinformatics*, 55(1):66–76, 2004.

[12] D. Cai, X. He, and J. Han. Document clustering using locality preserving indexing. *Knowledge and Data Engineering, IEEE Transactions on*, 17(12):1624–1637, 2005.

[13] J. Fowkes and C. Sutton. Parameter-free probabilistic api mining at github scale. *arXiv preprint arXiv:1512.05558*, 2015.

[14] T. Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.

[15] M. Ghafari, K. Rubinov, and M. M. Pourhashem K. Mining unit test cases to synthesize api usage examples. *Journal of Software: Evolution and Process*, 29(12):e1841, 2017.

[16] X. Gu, H. Zhang, and S. Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[17] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[18] J. Han and C. Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *From Natural to Artificial Neural Computation*, pages 195–201. Springer, 1995.

[19] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9. IEEE, 2010.

[20] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675. ACM, 2014.

[21] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[22] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 486–493. IEEE, 2011.

[23] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[24] B. Kulis, S. Basu, I. Dhillon, and R. Mooney. Semi-supervised graph clustering: a kernel approach. *Machine learning*, 74(1):1–22, 2009.

[25] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[26] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178. ACM, 2000.

[27] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can I use this method? In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE, 2015.

[28] A. Narayanan, G. Meng, L. Yang, J. Liu, and L. Chen. Contextual weisfeiler-lehman graph kernel for malware detection. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4701–4708. IEEE, 2016.

[29] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522. ACM, 2016.

[30] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen. A deep neural network language model with contexts for source code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 323–334. IEEE, 2018.

[31] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE, 2015.

[32] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[33] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Fundamental Approaches to Software Engineering*, pages 440–455. Springer, 2009.

[34] P. Nguyen, J. Di Rocco, D. Ruscio, L. Ochoa, T. Degueule, and M. Di Penta. Focus: A recommender system for mining API function calls and usage patterns. In *41st ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019.

[35] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. 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*, pages 383–392. ACM, 2009.

[36] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Learning API usages from bytecode: A statistical approach. *arXiv preprint arXiv:1507.07306*, 2015.

[37] H. Niu, I. Keivanloo, and Y. Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, 22(1):259–291, 2017.

[38] H.-S. Park and C.-H. Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2):3336–3341, 2009.

[39] J. C. Platt. Autoalbum: Clustering digital photographs using probabilistic model merging. In *Content-based Access of Image and Video Libraries, 2000. Proceedings. IEEE Workshop on*, pages 96–100. IEEE, 2000.

[40] L. R. Rabiner and B.-H. Juang. An introduction to hidden markov models. *ASSP Magazine, IEEE*, 3(1):4–16, 1986.

[41] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi. Graph kernels for chemical informatics. *Neural Networks*, 18(8):1093–1110, 2005.

[42] K. D. Rosa, R. Shah, B. Lin, A. Gershman, and R. Frederking. Topical clustering of tweets. *Proceedings of the ACM SIGIR: SWSM*, 2011.

[43] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[44] J. Shawe-Taylor and N. Cristianini. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[45] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *The Journal of Machine Learning Research*, 11:1201–1242, 2010.

[46] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

[47] L. Wu, L. Du, B. Liu, G. Xu, Y. Ge, Y. Fu, J. Li, Y. Zhou, and H. Xiong. Heterogeneous metric learning with content-based regularization for software artifact retrieval. In *Data Mining (ICDM), 2014 IEEE International Conference on*, pages 610–619. IEEE, 2014.

[48] T. Xie and J. Pei. Mapo: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[49] D. Zhang, Y. Liu, L. Si, J. Zhang, and R. D. Lawrence. Multiple instance learning on structured data. In *Advances in Neural Information Processing Systems (NIPS)*, pages 145–153, 2011.

[50] D.-Q. Zhang, C.-Y. Lin, S.-F. Chang, and J. R. Smith. Semantic video clustering across sources using bipartite spectral clustering. In *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*, volume 1, pages 117–120. IEEE, 2004.

[51] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.