Identifying Concepts in Software Projects

Mathieu Nassif and Martin P. Robillard

Abstract—When working on a project, software developers must be familiar with computing concepts, standards, and technologies related to the project. We present a novel approach, called Scode, to automatically identify those concepts using the project's documentation. Scode combines entity linking and network analysis techniques specialized for the software development domain. In addition to concepts explicitly mentioned in the documentation, Scode can retrieve implicit concepts related to the project's domain. Concepts identified by Scode have a recognized meaning that is consistent across projects. We compared Scode to different baselines and found that it is more effective at mapping projects to a consistent concept space.

Index Terms—Concepts identification, Software documentation, Wikipedia mining, Semantic analysis.

1 Introduction

Given the breadth of programming concepts, software developers must constantly learn about various technical concepts related to a project [1], [2]. For example, developers working on an e-commerce application may have to learn concepts related to database management systems and secure transactions, even if the application relies on third-party libraries to implement these requirements. However, precisely identifying the concepts relevant to a software project, beyond the most prominent ones, is challenging. A precise mapping of concepts to the software project and its components can help developers perform activities such as identifying the source of a feature [3], [4], [5], the cause of a bug [6], [7], or a third-party library for a specific task [8].

We propose a novel solution, named *Scode*, to automatically identify concepts related to a software project. We sought to retrieve concepts whose meaning is *recognized* beyond a specific project. We refer to such concepts as *recognized concepts*. We argue that for a concept to be recognized beyond a project makes it easier to interpret by developers. We also sought to identify not only *explicit* concepts mentioned in a project, but also *implicit* concepts related to the project's domain. For example, a mention of the SHA-256 algorithm likely implies that the project is related to the more general concept Cryptography.

To realize these aims, Scode relies on Wikipedia articles as proxies for concepts. It first leverages a computational linguistic technique called *wikification* [9] to identify explicit concepts from the project's documentation. Then, it uses *community search* algorithms [10], from the network science domain, to identify additional, implicit concepts. Hence, Scode constitutes an alternative to prior work that synthesizes concepts from recurrent terms (e.g., [11], [12]) to build glossaries (e.g., [13], [14]) and ontologies (e.g., [15], [16]).

To better understand the merits and limitations of a concept identification strategy based on wikification and community search techniques, we performed an extensive evaluation of Scode. In addition to its end-to-end performance, we assessed its underlying mechanisms individually. We also assessed the impact of using documentation instead of code identifiers as input to identify concepts. We found that the precision of practical configurations of Scode ranges from 25% to 66%. Scode also identifies the same recurring concepts more consistently than comparable approaches. The contributions of this article are:

1) a novel approach to identify project-related concepts with a recognized meaning;
2) the results of an extensive empirical evaluation of our concept identification strategy;
3) a curated list of over 6700 computing-related Wikipedia articles, used to improve the performance of wikification tools for software documents.

In the remainder of this article, we motivate our work and provide a brief introduction to the wikification and community search problems (Section 2), describe the design of Scode (Section 3), then report the methodology and results of our evaluation (Section 4). We finally discuss related work in Section 5 and conclude in Section 6.

Data Artifact

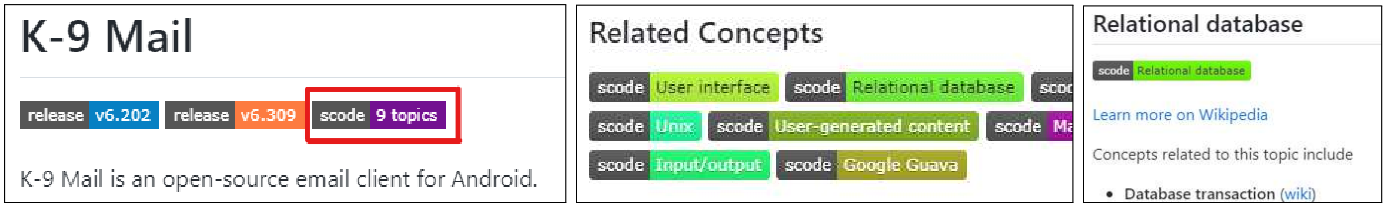
This article is complemented by an on-line appendix that contains replication data used in our evaluation and the list of 6746 computing-related Wikipedia articles. This appendix is available at https://doi.org/10.5281/zenodo.6459607.

2 Motivation and Background

We motivate the need for novel concept identification techniques by considering the design of a sample application. This application inserts badges in a README file to represent concepts relevant to a project, grouped by topic. Figure 1 shows an example of the badges generated by this application for the K-9 Mail Android app. A summary badge that indicates the number of topics identified for the project is placed alongside other usual badges at the top of the README file (Figure 1a). The application also inserts one badge per topic in a dedicated section of the README file (Figure 1b). In a separate file, the application generates a section for each topic, listing the project-related concepts that belong to this topic (Figure 1c).

• *M. Nassif and M. P. Robillard are with the School of Computer Science, McGill University, Montréal, Canada.*
E-mail: {mnassif,martin}@cs.mcgill.ca

Manuscript received XX; revised XX.



K-9 Mail is an open-source email client for Android.

Concepts related to this topic include

- Database transaction (wiki)

(a) README File with Summary Badge (b) Dedicated Section with All Topics (c) Topic Information

Fig. 1: Scode can be used to create badges that indicate the relevant topics for an open source project such as K-9 Mail.

The badges fulfill a purpose similar to GitHub Topics.[1] However, whereas GitHub topics typically focus on project-wide concerns (e.g., application type, development and building dependencies), *scode* badges also capture concerns related to smaller portions of a project, for example that it includes a user interface controlled by a layout manager. These badges can ease the introduction of new contributors to a project by showing the background they need. The topic information file contains relevant concepts from that topic, each linked to a Wikipedia article, so that developers can learn about or recall unfamiliar topics [1], [2].

This scenario surfaces the limitations of techniques that extract concepts by analyzing recurrent terms in source code and documentation. First, using concept names recognized by Wikipedia, rather than the project-specific terms, ensures that they have a consistent meaning across projects. Thus, developers will know whether they need to learn more about a topic before having to read the project-specific description. Second, associating each concept with a Wikipedia article provides a natural way to learn about the concept when necessary. Finally, grouping concepts by topic, rather than presenting them as a flat list, helps developers navigate efficiently through the required knowledge even if a large number of concepts are presented.

With over six million articles, the extensive coverage and popularity of Wikipedia makes it a valuable knowledge base of recognized concepts. Although there are software-specific knowledge bases and glossaries (e.g., ISO/IEC/IEEE's vocabulary of systems and software engineering [17]), the subset of Wikipedia relevant to computing offers a more extensive and up-to-date coverage, thanks to its vast community of contributors.

The development of efficient wikification techniques in the last decade enables the use of Wikipedia articles as proxies for concepts. *Wikification* refers to the task of linking mentions of concepts in free-form text to relevant Wikipedia articles [18], [19], [20]. It is a variant of the *named entity recognition and disambiguation* task, expanded to include unnamed entities, i.e., concepts [21]. Several tools are now available to perform this task on arbitrary texts. These tools are designed to retrieve the correct sense of a word in a text (e.g., associate "map" with the data structure or the visual representation depending on the context) and link synonyms to the same concept (e.g., associate "map", "dictionary", and "associative array" to the same concept).

Wikification tools, however, cannot identify concepts that are only implied by a document. For example, a text mentioning SHA-256 is also related to the implicit concept of CRYPTOGRAPHIC HASH FUNCTION. Implicit concepts are important to reduce the impact of variations in the way software documentation is written when identifying related concepts. Community search algorithms can help identify such implicit concepts. The *community search* task consists of identifying, within a graph, a densely connected subgraph that surrounds a query node [10], [22]. Thus, using Wikipedia articles as nodes and hyperlinks between them as edges, we can apply community search algorithms to find further articles related to those mentioned in the project's documentation. The *communities* found by these algorithms can also form the basis for aggregating concepts by topic.

1. https://github.com/topics/

3 Concept Identification

Our approach to identify relevant Wikipedia-based concepts works in two on-line phases, summarized in Figure 2. The first phase identifies concepts that are *explicitly* mentioned in the documentation using a *wikification* service (Section 3.2). The second phase uses explicit concepts as seeds to identify additional *implicit* concepts related to the same domain. For this phase, we implemented a set of *community search* algorithms (Section 3.3). The outcome of the community search algorithms is also used to group concepts into topics, each characterized by a single representative concept.

These two phases rely on data processed once during a third, off-line phase. This phase consists of preparing the Wikipedia data archive to extract a graph of computing-specific articles (Section 3.1). We also precompute properties of this graph to support the community search algorithms.

3.1 Off-Line Preparation

Both on-line phases of Scode rely on a graph of concepts derived from wikilinks, i.e., hyperlinks between Wikipedia articles. The edges between concepts in this graph provide the basis to identify related implicit concepts with the community search algorithms. The concepts included in the graph also act as a whitelist to filter the explicit concepts identified by the wikification service. Thus, the accuracy with which the graph models the computing domain plays an important role in improving the accuracy of Scode.

Although it would be possible to use the entire wikilink graph, this graph is extremely large and highly connected. We used a semi-automated approach to extract this software-specific subgraph from the April 2020 archive of the English-language Wikipedia. We only retained articles in the main namespace of Wikipedia, i.e., excluding pages such as categories and article discussions. We also retained only wikilinks that are inserted directly in the code of an article,

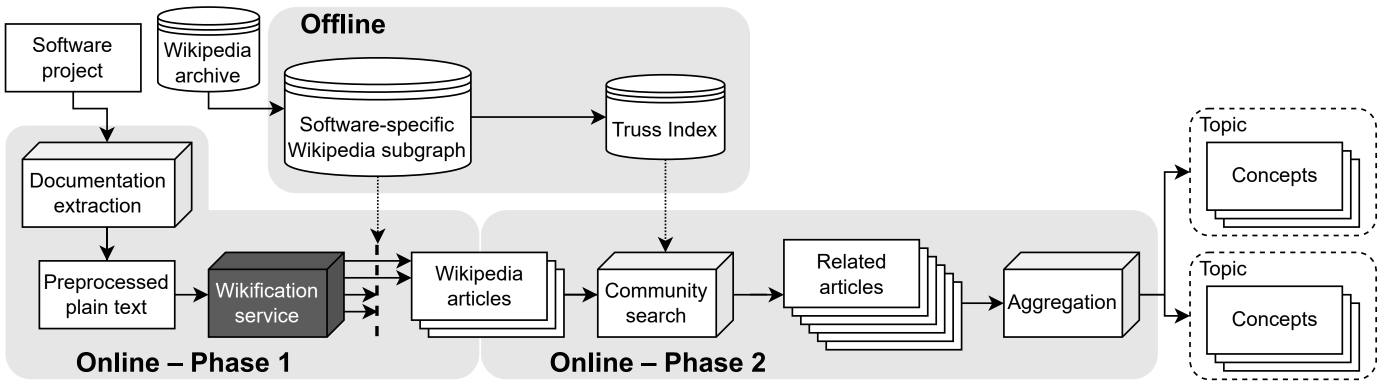


Fig. 2: Overview of our approach. It takes as input a software project and returns a set of associated concepts, grouped by topic. It builds on state-of-the-art wikification and community search techniques, and includes an off-line phase to process data from a Wikipedia archive.

i.e., not via a template. This filter discriminates wikilinks that are intentionally chosen by Wikipedia editors to relate two specific subjects from those added automatically by, e.g., *navboxes* and *sidebars*.[2]

Wikipedia contains *redirect pages*, which automatically redirect the reader to another target article when accessed. These pages are often used to add alternative titles to an article. We removed those pages from the set of nodes, but resolved and retained wikilinks passing through these redirect pages. That is, if an article A links to a redirect page R, whose target is B, we add an edge from A to B in the graph. Finally, we discarded *disambiguation pages*[3] and *set index articles*,[4] which are only used to list the different senses of polysemous terms.

After applying these filters, we obtained a pruned graph with 5.65 million nodes and 129 million edges (average node degree of 45.7). We considered all edges as undirected and unweighted, which is consistent with the interpretation that a wikilink indicates an unquantifiable and reciprocal relatedness relation between the source and target concepts.

We further filtered the set of articles using a systematic manual procedure to retain only those related to computer science and software technologies (Figure 3). The manual procedure requires as input the automatically pruned Wikipedia link graph W and a seed set C of candidate articles potentially related to computing. To produce the initial candidates, we gathered all articles listed in four Wikipedia navigation pages containing notable computing-related articles,[5] as well as the 1098 articles identified as computing-specific in a prior study [23]. The resulting *candidate* set C contained 2045 distinct articles covering different areas of computing.

For each candidate, an annotator scanned the content of the article to determine whether it was related to computing (line 5), in which case the candidate was added to a *related* set R (line 6). When adding an article to R, each of its neighbors not yet considered (line 7) and that have more neighbors inside than outside R (line 9) became a candidate (line 10).[6] Rejected articles were added to a *discard* set D to avoid considering them again as candidates (line 14). As the set of candidates is updated every time a candidate is accepted, we could not divide the annotation task into parallel subsets annotated independently by each author. Thus, the first author performed the entire task, accepting 6746 articles and rejecting 2811 others. To measure the subjectivity of the annotation task, the second author independently annotated a sample of 800 articles taken at random with equal probability from the final content of R and D. As this exercise only served to assess the subjectivity of the reliability of the first annotator's results, we did not resolve conflicts. We observed a substantial [24] inter-rater agreement (Cohen's $\kappa = 0.71$ [25]), suggesting that the selected articles are a reliable representation of the

```
Input: $W$: Wikipedia link graph
Input: $C$: Seed candidate set
Output: $R$: Subset of computing-related Wikipedia articles
1: $D \leftarrow \emptyset$    // unrelated articles to discard
2: while $C \neq \emptyset$ do
3:     $c \leftarrow$ arbitrary element from $C$
4:     $C \leftarrow C \setminus \{c\}$
5:     if $c$ is related to computing then
6:         $R \leftarrow \{c\} \cup R$
7:         for all $n \in$ neighbors$(c) \setminus (C \cup R \cup D)$ do
8:             $N \leftarrow$ neighbors$(n)$
9:             if $|N \cap R| \geq |N \setminus R|$ then
10:                $C \leftarrow \{n\} \cup C$
11:            end if
12:        end for
13:    else
14:        $D \leftarrow \{c\} \cup D$
15:    end if
16: end while
```

Fig. 3: Identification of Computing-related Articles

2. https://en.wikipedia.org/wiki/WP:NAVBOX
3. https://en.wikipedia.org/wiki/WP:DAB
4. https://en.wikipedia.org/wiki/WP:SIA
5. The articles are *Glossary of computer science*, *Index of object-oriented programming articles*, *Index of computing articles*, and *Index of software engineering articles*.

6. The condition on line 9 prevents the set of candidates from growing too quickly and rendering the manual validation impractical. We chose this criterion as it generates the same sets of related articles regardless on the order in which the set of candidates is processed.

computing domain on Wikipedia.

The resulting graph of 6746 computing articles contains 102 854 edges (average node degree of 30.5). This graph excludes general-domain concepts, such as LANGUAGE and MIND. It also excludes specific end-user applications (e.g., names of video games), learning resources on computing (e.g., textbooks), and low-level hardware technologies (e.g., models of microprocessors). It includes general computing concepts and domains (e.g., DATA PARALLELISM, ARTIFICIAL INTELLIGENCE), programming languages (e.g., C++), and development technologies (e.g., INTELLIJ IDEA, SPRING FRAMEWORK). We make this graph, as well as the annotation guide and annotations, publicly available in our on-line appendix.

In addition to extracting a relevant subgraph of concepts from the Wikipedia link graph, the off-line phase involves preparing a *truss index* to support one of the community search algorithms. We reimplemented the procedure to compute this index as described by Akbas and Zhao as part of their community search technique [26]. We define trusses and describe the use of the truss index in Section 3.3, together with the other algorithms we used to expand concepts into topics.

3.2 Explicit Concept Identification

Scode uses a wikification service to identify explicit concepts in documentation. To design this phase, we relied on the results of a prior study in which we compared six state-of-the-art wikification tools [23]. We found that the performance of all tools decreased when applied to software documents, typically due to technical terms being confused with their domain-general sense (e.g., TREE resolved as the type of plant). Using a whitelist of computing-specific concepts is an effective strategy to counter this limitation [23].

We chose the Babelfy tool [27] for Scode. Babelfy is an on-line wikification service that achieved good precision for reasonable recall levels, relatively to the other tools. To produce its natural language input, Scode aggregates all Javadoc comments in a source file, keeping them in their original order within the file. Each comment consists of an initial description of the type or method being documented, followed by a list of tag-fragment pairs. For example, the documentation of a method can start with its general purpose, followed by fragments describing each parameter, marked by the @param tag. All elements can use HTML and in-line Javadoc-specific syntax. Before aggregating these comments, Scode removes the HTML and Javadoc syntax, and excludes any comment description or fragment that is too short.[7] Excluding short descriptions and fragments prevents the input from containing incomplete phrases (e.g., *"Used for nontrivial settings upgrade"*) and irrelevant information (e.g., the authors of a class).

Like most wikification tools, Babelfy is more precise when processing large texts, because text fragments of only a few words lack sufficient context to disambiguate the sense of polysemous words. Thus, Scode aggregates all the comments of a project into a single input, to provide as much context as possible.[8] The order in which Scode aggregates the documentation from different files is arbitrary.[9]

7. We used an arbitrary minimum of ten words.

8. Due to a limitation of the API, Scode makes several requests if the aggregated comments contain more than 10 000 characters. Scode avoids splitting comments to the extent possible.

```
Input: Set $C$ of explicit concepts
Output: Set of topics $T$
 1: $M \leftarrow$ empty map
 2: $R \leftarrow \emptyset$
 3: for all concept $c \in C$ do
 4:     $T \leftarrow$ LARGETRUSS($c$)        // $T$ is itself a set of concepts
 5:     $T \leftarrow$ REDUCETRUSS($T$)
 6:     if $|T| \geqslant 100$ then
 7:         $T \leftarrow$ REDUCEECC($T$)
 8:     end if
 9:     $r \leftarrow$ REPRESENTATIVE($T$)
10:     if $r \in$ keys($M$) then
11:         $T \leftarrow T \cup M[r]$
12:         $R \leftarrow R \cup \{r\}$
13:     end if
14:     $M[r] \leftarrow T$
15: end for
16: for all $r \in R$ do
17:     $M[r] \leftarrow$ REDUCETRUSS($M[r]$)
18: end for
19: return values($M$)
```

Fig. 4: Implicit Concept Identification Procedure

Babelfy returns a list of concepts, identified as entries in the BabelNet knowledge base [28]. BabelNet entries can be mapped to corresponding Wikipedia articles. Scode removes from the output any concept that is not in our curated list of computing-specific Wikipedia articles. This filter can produce false negatives, for example due to non-computing concepts related to the problem domain of the project. However, the negative impact of these missing concepts is outweighed by the larger increase in precision observed in our prior work when using a whitelist [23]. The explicit concepts output by this phase seed the search for additional implicit concepts in the next phase.

3.3 Implicit Concept and Topic Identification

Edges in the computing-specific Wikipedia graph indicate relatedness between articles, and thus can be used to find implicit concepts. However, despite the heuristics described in Section 3.1 to reduce the number of edges, the graph remains densely connected. Trivial heuristics, such as taking all concepts within an arbitrary distance of the seed concepts, produce too many concepts.

To circumvent this issue, we used community search techniques to identify a practical number of concepts. Informed by Fang et al.'s survey of community search algorithms for large graphs [10], we reimplemented three algorithms to identify implicit concepts, as shown in Figure 4.

The procedure starts by identifying a separate community around each explicit concept c. An initial community (or topic) T is generated using Akbas and Zhao's algorithm [26] to find the *largest k-truss* around c, for the highest

9. Wikification techniques are noisy and sensitive to the order of their input. In an ancillary experiment, we confirmed this sensitivity, but found no obvious way for the input order to systematically improve the quality of the result. Babelfy introduced a similar amount of noise for the arbitrary order of Scode as for other random permutations.

possible value of k (function LARGETRUSS of Figure 4). Within a graph, a k-truss is a subgraph in which any two neighbors (i.e., nodes connected by an edge) share at least $k-2$ other common neighbors. Akbas and Zhao's algorithm is particularly efficient for repeated queries over a large graph, as it precomputes the key information about the potential trusses and stores it in a *truss index*.

As we favor smaller communities, Scode applies Huang et al.'s algorithm [29] to reduce the size of the k-truss while maintaining the query concept c and the value of k (function REDUCETRUSS of Figure 4). Their algorithm approximates the subset of the initial k-truss for which the maximum distance from the query concept c to any other concept is as small as possible.

We observed a bimodal distribution in the number of concepts per topic. In many cases, the reduced topic has fewer than 25 concepts. However, when the initial query concept is a popular computing concept, such as JSON or SQL, this minimal community still contains several hundred concepts. Scode further reduces the size of any community with over 100 concepts by relying on an alternative definition of connectivity for communities: *k-edge-connected-components*, or k-ECC. A k-ECC is a subgraph that requires at least k edges to be removed for the graph to become disconnected (i.e., it is no longer possible to find a path between any two nodes). For example, a set of k nodes that are all pairwise connected is a (k-1)-ECC. This connectivity requirement is looser than k-trusses, which allows to reduce the size of a (k-1)-ECC derived from a k-truss while maintaining the value of k. We used Hu et al.'s algorithms [30] to perform this reduction for large k-trusses (function REDUCEECC of Figure 4).

The algorithms described so far generate a separate community for each explicit concept. The next step is to merge communities that describe similar topics. As the size of communities varies considerably, many intuitive heuristics (e.g., merging communities with a large overlap) have a detrimental effect on the output. For example, they can void the information captured by a small specialized topic by merging it with a large generic community. Trivial merging heuristics can also lead to arbitrary topics that depend on the order in which the communities are merged.

We implemented the community aggregation step by seeking a *representative* concept for each community. Following the insights from Lizorkin et al. [31], we chose the representative as the concept with the highest *pagerank* value [32] among the community (function REPRESENTATIVE of Figure 4).[10] Scode merges all communities that have the same representative into a single community, and applies the k-truss reduction algorithm once more on the new communities to further reduce their size.

This final set of communities forms the outcome of Scode. Each community represents a single topic of related concepts, identified by one representative concept.

3.4 Implementing the Sample Application

We implemented the application described in Section 2 to demonstrate the usefulness of Scode in a practical scenario.

10. The representative concept is typically more general than the explicit concept. In our evaluation with Java classes, we observed that 15% of topics have an explicit concept as its representative.

After collecting a set of concepts grouped by topic, the application separates small topics (i.e., containing at most 25 concepts) from larger ones. The summary badge inserted at the top of the README file indicates the number of small topics. For each small topic, the application generates one badge using the name of the topic's representative concept as the value of the badge. The color of the badge is based on the representative concept: we automatically assigned to each concept in the computing-related subset of Wikipedia a unique color, giving similar colors to similar concepts based on a hierarchical clustering of the Wikipedia graph. The application uses the Shields.io service to generate the badge image with the desired name (*scode*), value, and color.

The application also creates a new file to describe the topics in more detail. In this file, we add one section for each small topic, and each badge in the README file links to its corresponding section. The section body contains the explicit concepts from this topic first, as they are more likely to be of importance to the developer. Further implicit concepts are placed in a collapsible list under the explicit concepts. As each concept is a Wikipedia article, the application creates a hyperlink from each concept to the associated article.

Large topics are not included. We observed that they tend to contain less cohesive groups of general programming concepts. Hence, developers are less likely to find the extent of such topics useful, especially if they must scan hundreds of concepts. Hence, we only retain *explicit* concepts included in large topics, and list them all under a *"General Programming Concepts"* section in the new file.

4 EVALUATION

The objective of our evaluation was to understand the strengths and limitations of an approach based on wikification and community search, as represented by Scode, to identify concepts relevant to a software project. We centered this evaluation around four research questions:

RQ1 How accurate are concepts identified by Scode?
RQ2 How consistent are concepts identified by Scode?
RQ3 How effective are the internal mechanisms of Scode?
RQ4 How do concepts extracted from documentation differ from those extracted from code identifiers?

We first evaluated the end-to-end performance of Scode and compared it to two baseline techniques. To measure the performance of concept identification techniques, we used two metrics: The *precision* of a technique corresponds to the proportion of the concepts it identifies that are actually related to the project (RQ1, Section 4.2). Conservative techniques can achieve a high precision by returning few or very specific concepts. To balance this dimension, we used the *consistency* of the identified concepts as the second metric (RQ2, Section 4.3). A technique should consistently identify the same concept for any project related to this concept.

To gain a better understanding of the promising and limiting components of our approach, we evaluated the internal mechanisms of Scode separately (RQ3): the whitelist of computing concepts (Section 4.4), the wikification service (Section 4.5), and the community search algorithms (Section 4.6). Finally, as it is a major design decision for Scode, we studied the impact of extracting concepts from documentation rather than source code (RQ4, Section 4.7).

4.1 Study Design

We generated a list of concepts related to a sample of Java classes and manually validated the concepts' relatedness to their associated class. We compared Scode to two state-of-the-art techniques as baselines, following the same procedure. We computed the *precision* and *consistency* of all techniques to answer our first two research questions.

We selected as baselines tools that are publicly available and that extract concepts from the documentation of a project. The first baseline is a technique to identify concepts related to Java classes in order to build a comprehensive API knowledge graph [33], [34]. The technique relies on various natural language processing heuristics specifically tailored for API concept extraction. In contrast to Scode and the other baseline, the concepts are not intended to have a recognized meaning in an external knowledge base. For our evaluation, we extracted the concepts from the API knowledge graph, which we refer to as *PengKG*, instead of re-implementing the concept extraction technique.

We used explicit semantic analysis (ESA) [35] as the other baseline technique. ESA associates arbitrary text inputs with entries of a knowledge base, such as Wikipedia. We used the EasyESA implementation of ESA in this evaluation [36]. To produce the input for EasyESA, we used the same processed documentation as for Scode, described in Section 3.2.

We limited the size of the subjects in our evaluation sample, so that the evaluators could understand each subject in a reasonable amount of time. Thus, instead of using whole projects, we considered individual Java files as independent subjects. Because PengKG did not include concepts for interfaces, we only sampled files for which the top level type is a class. Hence, we refer to each subject as a Java class, with the understanding that it may include nested classes.

We randomly sampled 100 Java classes, each with equal probability, from the standard Java Class Library, version 15. We used the standard library due to its coverage of many areas and the high quality of its documentation. These classes were also part of PengKG.

Scode identified a total of 84 to 774 concepts per class (average of 459), except for five classes that were associated with no concept due to their insufficient documentation. To avoid overloading the annotators, for classes associated with more than 100 concepts, we randomly sampled 100 concepts among Scode's output for the annotation task, with each concept having an equal probability of being selected.

For each class, PengKG includes one to 42 related concepts (average of 10). We included all of them in our evaluation. For EasyESA, as it takes a number of concepts to generate as part of its input, we generated five more concepts than the number of concepts identified by Scode. We did not use a constant number of concepts to account for classes that are inherently related to very few or many concepts. Because EasyESA's results are ranked, we selected the 100 highest ranked concepts for the annotation task.

In total, we generated 20 005 class–concept pairs to evaluate. For each pair, the two authors judged whether the concept may be relevant to a developer working with the class. We did not rely on external annotators for this procedure as it requires a significant effort to read and understand the code of each class and peruse the content of Wikipedia articles before judging whether the concepts are related. To avoid a negative bias against baseline techniques, the technique that generated each pair was not identifiable during the annotation task. We also put all concepts in lower case, and removed parentheses from Wikipedia titles to further reduce the distinctions between Wikipedia-based concepts and the free-form concepts from PengKG.

TABLE 1: Concept Identification Precision for 100 Java Files

Technique	Concepts	Sample	Gen.	Relevant	Prec.
PengKG	1006	1006	187	584	71.3%
EasyESA	44 069	9455	674	340	3.9%
top 1	97	97	9	20	22.7%
top 3	291	291	36	44	17.3%
with whitelist	1160	1160	439	79	11.0%
top 1 whitelist	93	93	26	22	32.8%
top 3 whitelist	265	265	96	38	22.5%
Scode	43 584	9663	6432	146	4.5%
small topics	1453	288	159	32	24.8%
representative	548	115	93	9	40.9%
explicit	1071	306	208	65	66.3%
implicit	42 513	9357	6224	81	2.6%

TABLE 2: Examples of Concepts Identified by PengKG, EasyESA, and Scode

Technique	Sample Concepts (original capitalization)
PengKG	PREFERREDSIZE, Minimum Size, Property Change Listener, utc, detail message, CompositeData value
EasyESA	Byte stream, GUI widget, Stream (computing), Endianness, Home directory, Aspect ratio
Scode	Concurrent computing, Character encoding, Serialization, Layout manager, Dialog box, Parsing

The authors first annotated all concepts associated with two classes and discussed the results to refine the annotation procedure. We found that it was unreliable to grade relatedness on a scale. Thus, the annotators annotated each pair as a binary variable, i.e., whether the concept is related to the implementation of the class, its usage, or the abstraction it represents. During the preliminary phase, we also found that some general concepts could be relevant to virtually all developers for any class. These concepts include, for example, fundamental computer science concepts such as BYTE and CONDITIONAL BRANCHING. Annotators assigned the special annotation general to these concepts and did not further assess their relatedness to specific classes.

After refining the annotation guidelines, each annotator independently annotated 54 of the remaining 98 classes, so that the concepts associated with ten unmarked classes would be annotated by both annotators to assess the inter-rater reliability. The two annotators achieved a substantial agreement [24] (Cohen's $\kappa = 0.74$ [25]). Conflicts within the ten common classes were resolved by a discussion between both annotators.

4.2 Precision of the Identified Concepts

Table 1 summarizes the results of the annotation task to answer our first research question. The number of concepts identified by each technique and those that we annotated are shown in the "Concepts" and "Sample" columns, respectively. The sample size varies based on the number of concepts identified by each technique (capped at 100 per class) for the 100 classes. The "Gen." column shows how many concepts were marked as general. Finally, the last two columns report the number of class–concept pairs marked as related and the precision of each technique, computed as the ratio of relevant to non-general concepts. We excluded general concepts from the precision as they are related to virtually any class, but capture little information about them. This conservative decision impacted Scode the most. For each technique, the top row in bold indicates the global values for the technique, and subsequent rows show values for different variations as described below.

PengKG

PengKG is the most precise technique with a precision of 71.3%. It returned a number of true positives comparable to but smaller than the other techniques.

However, while inspecting the concepts provided by PengKG, we observed recurrent patterns of concepts that matched the class name and its members. For example, the class ServerSocket, which contains the methods getLocalPort() and getInetAddress(), was associated with the concepts LOCAL PORT and INET ADDRESS. Although relevant, these concepts do not reveal additional information beyond the interface of a class. In particular, PengKG associated 44 of the 100 classes with a concept that exactly matches the class name. PengKG also contained many overly specific concepts such as COMPOSITEDATA VALUE (see Table 2). Contrary to concepts with an externally recognized definition, these concepts are meaningless without prior knowledge of the class implementation.

EasyESA

In contrast to Scode and PengKG, EasyESA ranks concepts from most to least relevant to the input text. Considering only the top concepts increases the precision of EasyESA. However, we found that even when considering only its highest ranked result, the precision was only 22.7%. As more concepts are considered, the precision decreased rapidly, already to 17.3% when considering the top three concepts (see rows *top 1* and *top 3* of Table 1).

Scode

Scode produced a large number of concepts for each class, with a considerable amount of noise. The high proportion of false positives can be explained by explicit general concepts, which tend to gather large, ill-defined topics during the community search phase of Scode. Concepts from such large topics were also pervasive in the annotation sets due to the uniform sampling across topics. For example, if Scode produces for a class three small, cohesive topics of 20 concepts each, and one large, low-quality topic of 500 concepts, sampling 100 concepts uniformly from the 560 identified by Scode will select mostly concepts from the large topic, despite most topics being more specific to the class. Specifically, only 1453 of the 43 584 concepts (3.3%) identified by Scode were in topics with at most 25 concepts.

When considering only concepts included in topics of 25 or fewer concepts, we observed that Scode's precision was significantly higher, up to 24.8% (see row *small topics* of Table 1). We used a threshold of 25 concepts for the size of a small topics based on our preliminary observation of a gap in the distribution of topic sizes, between 25 and 75 concepts. Although the precision remains modest, within a topic of 16 concepts, approximately four would be directly relevant to the project, thus warranting a look by developers unfamiliar with the domain.

We also assessed the quality of the topics by considering their single representative concept, as defined in Section 3.3. The precision of these concepts reached 41% (see the *representative* row of Table 1), although a disproportionately large number of concepts were marked as general and thus excluded from this computation.

Without constraints on the number of topics, we found that most of the time, Scode produced a manageable number of topics. For our sample of 100 Java classes, Scode produced at most 15 topics per class. For 90 classes, it even produced ten or fewer topics (including the five classes for which no concept were identified). The number of topics for an entire project is also small: When using 227 Android projects as input to Scode (see Section 4.6), it produced on average 16 topics per project, with 90% of the projects having fewer than 26 topics. Thus, it is possible for humans to get an idea of the conceptual context of an entire project by reviewing only a small number of topics.

Findings: Scode achieves a low precision when considering all implicit concepts (4%). However, when filtering out large general topics, the precision rises to 25%. Scode is even more precise for topic representatives (41%). In any configuration, Scode outperforms EasyESA for a comparable output, but EasyESA generates fewer general concepts. PengKG is the most precise concept identification technique (71%), but returns many trivial concepts.

4.3 Consistency of the Identified Concepts

Our second research question focuses on Scode's ability to generate recognized concepts consistently across projects. We looked at the concepts marked as related to get a sense of each technique's performance, and report illustrative examples in Table 2. Although PengKG is more precise than Scode and EasyESA, the concepts it identified were specific to each class. For example, PengKG identified that many classes of the Java Swing library define a PREFERREDSIZE, but such a concept has no meaning outside this library. Other concepts, such as COMPOSITEDATA VALUE, are obscure outside the specific context of the class. In contrast, Scode and EasyESA's concepts have by design a meaning recognized in a popular independent knowledge base.

To empirically assess the consistency of the three techniques' output, we measured how often each technique identified the same concept for different classes. Although some concepts are genuinely relevant to few classes, a technique should not only identify concepts so specific that they are only related to a single class. Otherwise, these concepts will be of limited value when comparing the conceptual context of different classes.

Figure 5 shows the recurrence of concepts in our sample of 100 Java classes. We used all concepts returned by the three techniques to produce this graph, without limiting them to 100 concepts per class to avoid the bias of a random

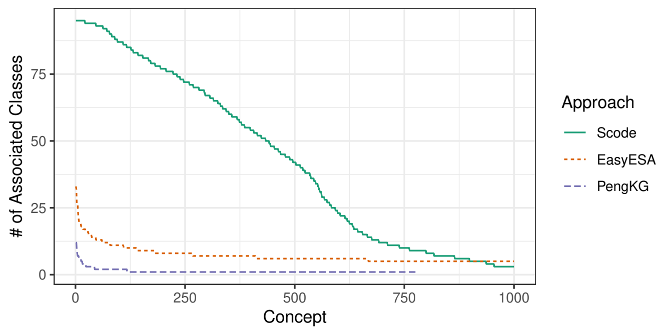


Fig. 5: Number of classes associated with each concept identified by Scode, EasyESA, and PengKG. The x axis shows each concept, in decreasing order of their recurrence across classes. For readability, the x axis is limited to the 1000 most recurrent concepts.

filter on the output. Therefore, the figure includes false positives, as not all results were annotated.

As the figure shows, Scode identified the same concept for multiple classes more often than EasyESA and PengKG. Some general concepts were associated with almost all classes, whereas other concepts were more specific to a few classes. This distribution of classes per concept effectively supports a comparison of the domain of multiple classes.

In comparison, PengKG's concepts were rarely reused across classes. Specifically, the 1006 class–concept pairs from PengKG contained 776 distinct concepts, 84.9% of which were associated with a single class. The most common non general concept, PREFERREDSIZE, was associated with only eight classes. In comparison, Scode's 43 584 class-concept pairs involved only 1281 distinct concepts. Only 10.9% of them were associated with a single class, and the median number of classes associated with each concept was 18.

Despite also using a finite set of concepts, i.e., Wikipedia articles, EasyESA also rarely identified the same concept for multiple classes. Of the 27 943 distinct concepts identified by EasyESA, 71.9% were associated with a single class. The most common concept, METHOD, was only associated with 33 classes, even though such a general concept is related to virtually any Java class. This result suggests that using an external knowledge base is not sufficient to identify consistent concepts, and shows the importance of finding a strategy to identify implicit concepts.

Findings: Concepts identified by Scode are more consistent than those identified by EasyESA and PengKG. They were linked to a varying number of the 100 sampled classes, from over 90 classes to a single one, with a median of 18. In contrast, the median EasyESA concept was linked to two classes, with the most common concept, METHOD, only found for 33 classes. The majority of PengKG's concept (85%) were linked to a single class.

4.4 Whitelist of Computing Concepts

One of our contributions is the manually curated list of 6746 Wikipedia articles about computing-related concepts. Scode uses this list to remove false negatives from the output of the wikification service. We investigated whether this whitelist could also improve the precision of other concept identification techniques by applying it to EasyESA.[11]

When using the whitelist to filter the top 100 results from EasyESA, we observed that it removed 87.7% of the original concepts, increasing the precision up to 11.0% (see row *with whitelist* of Table 1). When considering only the highest ranked result that is also on the whitelist, EasyESA achieved its highest precision of 33%, but this precision dropped rapidly, reaching 22.5% for the first three concepts (see rows *top 1 whitelist* and *top 3 whitelist* of Table 1).

A whitelist of concepts also reduces the number of unique concepts that a technique can return, limiting the number of concepts associated with a single class. In the case of EasyESA, among the 1070 unique concepts both in the top 100 results for a class and in the whitelist, only 38.5% were associated with a single class, and the median concept was associated with three classes.

Findings: Applying the computing-specific whitelist of Scode to EasyESA improves its precision from 23% to 33% (considering only the highest ranked concept). Although better, it remains below the precision of Scode (41% for topic representatives or 66% for explicit concepts).

4.5 Wikification and Community Search Algorithms

We measured the performance of the wikification service by considering only the subset of explicit concepts in our annotated data set (see *explicit* row of Table 1). We found that Babelfy identified on average 10.7 concepts per class, with a precision of 66.3% for non general concepts.

The precision is consistent with what we had observed in a prior comparative study of six wikification tools [23]. In this prior study, Babelfy achieved a precision of 75% with the same configuration as used in this current work. For this precision level, Babelfy achieved a recall of approximately 35%. This value is close to the maximum performance of other wikification techniques, which hardly surpass 40% recall. Nevertheless, it motivated the need for a second phase to recover missed concepts.

Excluding the explicit concepts, we observed that the precision of the community search algorithms was low, 2.6%, for identifying implicit concepts (see *implicit* row of Table 1). However, as discussed above, most false positives are due to large, generic topics. When considering only small topics, which mostly contain implicit concepts, the precision of Scode is well above the 2.6% value. Thus, the drop in precision is a necessary cost to achieve additional benefits, such as identifying further implicit concepts, grouping the results by topic, and discriminating specialized concepts from generic ones (based on the size of the community).

Findings: Babelfy's precision (66%) is almost on par with PengKG (71%), the most precise technique in our evaluation. In contrast, the community search algorithms introduce a relatively large amount of noise when identifying implicit concepts (precision of 2.6%), especially due to concepts that generate large communities.

11. As PengKG does not use Wikipedia articles as concepts, we cannot apply our whitelist to this technique.

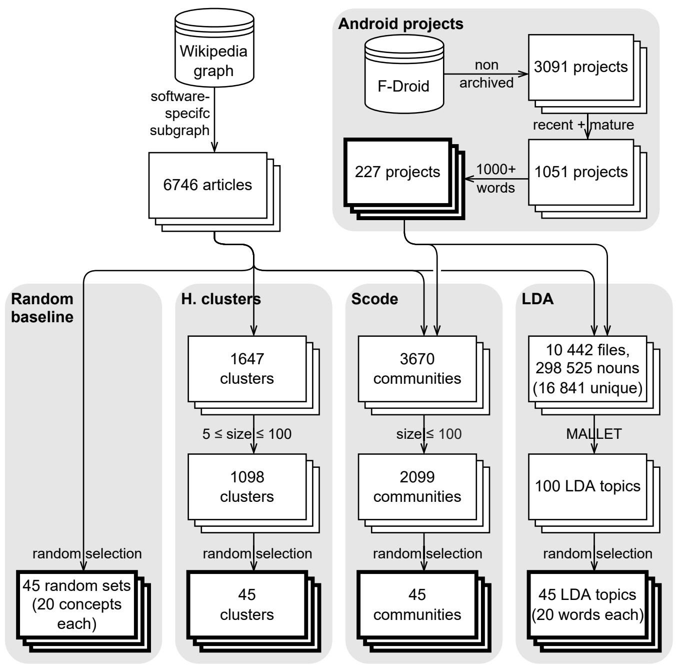


Fig. 6: Sample for the Topic Cohesiveness Evaluation

4.6 Topic Cohesiveness

We evaluated the ability of the community search algorithms, developed for general network science applications, to generate cohesive topics in the context of software-specific knowledge graphs. We compared the community search algorithms to three baselines: a hierarchical clustering algorithm [37], latent Dirichlet allocation (LDA) [38], and random groups of concepts as a soundness check. Because LDA requires a large training set, we sampled entire Java projects for this evaluation.

Project Sample

Figure 6 summarizes our project and topic sampling procedure. We sampled Android applications from F-Droid [39]. We considered all 3091 applications that were not archived at the time of the evaluation. We excluded from them applications for which the last version was released five or more years ago (*recency filter*) or for which the time difference between their first and last released versions was less than one year (*maturity filter*). The recency filter is relevant to ensure that recent concepts have a chance to be included in our sample. The maturity filter excludes trivial applications.

After applying these two filters, we obtained a set of 1051 projects. We removed projects for which the entire documentation contained less than 1000 words, computed as described in Section 3.2 (*documentation filter*). We only considered classes and interfaces in the package matching the application ID of the project, or one of its subpackages. After applying this filter, we obtained our final sample of 227 Android applications.

Topic Sample

We sampled 45 topics from each approach. This sample size is sufficient for a statistical analysis when comparing ordinal score distributions, but small enough to limit the threat of annotation fatigue (see the *Evaluation Metrics*).

We applied Scode to the 227 Android projects, generating a total of 3670 topics. We removed those that contained more than 100 concepts. We randomly sampled 45 of the remaining 2099 topics as the sample set from Scode.

Generating hierarchical clusters requires a distance metric between any two concepts, which we defined as the length of the shortest path between them in the undirected and unweighted Wikipedia subgraph. We then applied R's implementation of the unweighted pair group method with arithmetic mean (UPGMA) algorithm [37] to aggregate concepts into hierarchical clusters. As the outcome of this algorithm is a binary tree by design, it is sensitive to noise in the distance function. To mitigate this sensitivity, we rounded the distance function during the merging phase of the UPGMA algorithm to produce a tree that is not binary and less deep. After trying different rounding functions, we chose to round distances to the nearest non-exceeding quarter (e.g., 3.1 and 3.2 were rounded to 3 and 3.3 was rounded to 3.25). We excluded clusters if they either contained fewer than five concepts or were contained in larger cluster of ten or fewer concepts. We also removed clusters of more than 100 concepts. This procedure generated 1098 topics, from which we sampled 45 at random for the evaluation.

To generate LDA topics, we used the MALLET implementation of the algorithm for Java [40]. LDA is an unsupervised learning technique that takes as input a set of documents and generates latent topics represented as lists of words [38]. As LDA requires a training set with many documents to generate a good model, we considered each of the 10 442 files across all projects as a distinct document. For each document, we only retained nouns, for consistency with Wikipedia titles which are predominantly noun phrases. We further removed stop words and converted all nouns to lower case. This preprocessing produced a total of 298 525 terms across the 10 442 documents, forming a vocabulary of 16 841 unique nouns. We generated 100 topics, with the initial value of parameters α and β set to 0.01, and using 2000 iterations. We left other parameters to their default values. The resulting topics comprise between 62 and 670 terms (average of 295).

Finally, to support a soundness check, we generated 45 sets of 20 random concepts selected using a uniform probability distribution over the 6746 computing-related Wikipedia articles, with replacement between sets.

Evaluation Metrics

We used two complementary metrics to evaluate the cohesiveness of the topics. The first metric is a direct, subjective assessment on a five-point ordinal scale from 1 (least cohesive) to 5 (most cohesive). We asked three graduate students that were not involved in the project to provide this assessment. For each topic, we showed a maximum of 20 concepts to avoid overwhelming annotators with very large topics. We split the topics so that each annotator evaluated 15 topics from each approach, with no overlap.

We used a second metric to mitigate the subjectivity of the direct assessment. This metric is the success rate of the *word intrusion* task, described by Chang et al. [41]. In this task, an annotator is shown a set of $N + 1$ concepts in random order. N of these concepts belong to the same topic and the last one, the *outlier*, is chosen randomly among

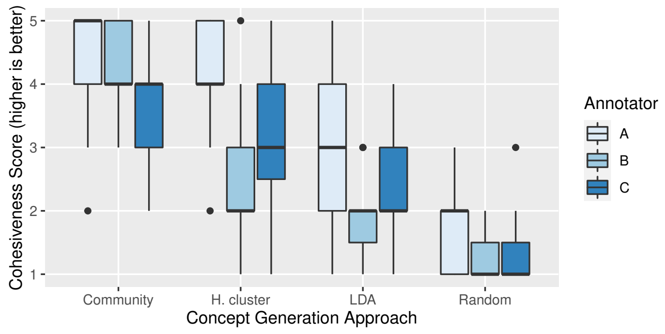


Fig. 7: Distributions of the cohesiveness scores on the five-point ordinal scale (higher is better) for each combination of approach and annotator. Each boxplot aggregates the 15 topics evaluated by one annotator.

TABLE 3: Success rate of the word intrusion task (percentage), median cohesiveness score (ordinal scale from 1 to 5, higher is better), and average cohesiveness score (in parentheses) for concepts generated by the four techniques and as evaluated by three annotators.

Ann.	Community	H. cluster	LDA	Random
A	60%; 5 (4.3)	40%; 4 (4.2)**	60%; 3 (3.0)**	20%; 2 (1.7)
B	67%; 4 (4.3)***	60%; 2 (2.5)	60%; 2 (1.9)**	13%; 1 (1.3)
C	73%; 4 (3.8)*	60%; 3 (3.1)*	73%; 2 (2.5)***	13%; 1 (1.3)
All	67%; 4 (4.2)***	53%; 3 (3.3)***	64%; 2 (2.5)***	16%; 1 (1.4)

Stars indicate the p-value of a one-tailed Mann–Whitney U test comparing the distributions of cohesiveness scores in a column to those to its right: less than 0.05 (*), 0.01 (**), or 0.001 (***).

concepts not in the topic. The annotator must then identify the outlier among the $N+1$ concepts. If a topic is cohesive, identifying the outlier should be easy, resulting in a high success rate. However, concepts from a vague topic will be indistinguishable from the outlier. The success rate for such topics should decrease towards $1/(N+1)$. For this evaluation, we chose the value $N = 5$. We generated a word intrusion task for each topic, and asked the same three annotators to perform the task, dividing topics evenly among them and ensuring that the same annotator did not provide a subjective score on the same topic for which they performed the word intrusion task.

For both measures, annotators were unaware of the technique that generated each topic. They did not know how many techniques were being compared, or that some concepts were Wikipedia titles (which were shown in lower case) whereas others were simply terms from the vocabulary of the LDA model. When selecting the evaluation metrics, we also considered synthetic measures such as the Silhouette index [42]. Although these metrics are arguably more objective than those we selected, their abstraction of human judgment poses a threat to construct validity.

Results

The community search algorithms generated topics that were more cohesive than those from the three baselines. Figure 7 shows the distribution of cohesiveness scores given by each annotator to topics generated by all four approaches. Each boxplot aggregates the scores of exactly 15 topics. The results show a consistent trend across annotators: communities are generally more cohesive than hierarchical clusters, followed by LDA topics, and finally the random baseline. As expected, random topics were the least cohesive, and validate that annotators can discriminate spurious topics by giving them very low scores, most often one or two.

Table 3 confirms this observation. It shows the median cohesiveness score of each group of topics, as well as the average scores, in parentheses, as this measure is more commonly used to approximate the center of a distribution. However, because cohesiveness scores are not on an interval scale, the averages cannot be used for further statistical analysis or interpretation. We used one-tailed Mann–Whitney U tests [43] to test the hypothesis that the cohesiveness differences are not due to chance. This test compares two groups of ordinal values and evaluates the probability that a value randomly selected from the first group is larger than a value randomly selected from the second group. The stars beside each value indicates the significance of the tests between consecutive approaches in the sequence {communities, hierarchical clusters, LDA, random}.[12]

All but two comparisons are statistically significant at the 0.05 level. The difference between hierarchical clusters and LDA topics annotated by B has a p-value of 0.080, and the difference between communities and hierarchical clusters annotated by A has a p-value of 0.30. These inconclusive results can be explained by the small group sizes (15 topics), as well as the high cohesiveness scores (capped at 5) given by A, which limits the discrimination of the most cohesive topics. When comparing all 45 topics from each approach, all tests are significant, with p-values all lower than 0.00092, confirming our initial observation (communities > hierarchical clusters > LDA topics > random concepts).

Table 3 also includes the proportion of outliers correctly identified in the word intrusion task. Interestingly, although communities remain ahead of the other approaches, LDA topics appear to be more cohesive than hierarchical clusters. One possible explanation for the higher success rate with LDA topics is that outliers are selected from a different vocabulary (i.e., Wikipedia articles) than the terms that form the topics (i.e., nouns).[13] Due to the binary result for each topic (success or failure), the only statistically significant differences are between the random baseline and the other approaches. Detecting a significant difference between the other approaches would have required an impractical sample size, i.e., over eight times larger to reduce the confidence interval radius to 5% at the 95% confidence level.

Findings: Three external annotators found topics identified by community search algorithms significantly more cohesive in a direct evaluation than topics generated by alternative techniques (hierarchical clustering and LDA). This subjective preference was confirmed by a corresponding performance on a word intrusion task.

12. P-values for non-consecutive approaches are smaller than the minimum of the p-values shown in the table. For example, the difference of *community* and *LDA* score distributions for annotator A is significant with a p-value lower than 0.01.

13. We chose this conservative evaluation approach because it favors the baseline. The alternative, i.e., selecting an outlier among the much larger, non computing-specific set of nouns used to train the LDA model, would have had a negative impact on the LDA baseline.

4.7 Documentation or Source Code as Input

In this work, we considered the documentation of a software project as input for identifying concepts. This decision contrasts with prior approaches that extract concepts from source code identifiers (e.g., [12], [16]).

Although using identifiers is the only option when documentation is lacking, using documentation as input can leverage more diverse forms of information. We hypothesized that the two strategies should generate different concepts, because information in source code is at a lower level of abstraction than the information found in documentation. To validate this hypothesis, we compared the output of Scode to concepts generated from source code identifiers.

We used LDA to generate topics from the identifiers in all the source files of the Java 15 standard library.[14] We considered each file as a separate document. We split each identifier based on camel case conventions and underscores and removed terms with less than three characters. Then, for each group of terms with the same stem according to the Porter stemming algorithm [44], we replaced all terms with the most common form. We set the number of topics to 100, the number of training iterations to 2000, and left the other hyperparameters to their default values in MALLET.

We compared the top LDA topics with Scode's concepts that are included in small topics (see Section 4.2). As it is difficult to compare LDA topics to concepts represented by Wikipedia articles, we focused on the *vocabulary* used by both strategies. We used the union of the top 20 terms from the top 10 LDA topics and compared it to the set of all terms from all Wikipedia titles, excluding terms in parentheses. We used their overlap coefficient, i.e., the ratio of the sets' intersection size to the size of the smaller set, to quantify this comparison. As MALLET's implementation of LDA is not deterministic, we repeated this procedure 100 times.

We found that the overlap between Scode's concepts and LDA topics varied depending on the class. Excluding classes for which Scode did not find any small topic,[15] the average overlap coefficient for each class, across the 100 trials, ranged from 0 to 0.19 (standard deviation ranges from 0 to 0.11). Inspecting the LDA topics confirmed that they capture low-level concepts. In addition to implementation-specific topics with terms such as *length*, *attribute*, and *string*, other topics representing domain abstraction are expressed with low-level concepts. For example, a topic about time zones includes terms such as *central*, *america*, and *gmt*.

Findings: Concepts extracted from recurrent terms in code identifiers differ from those extracted using Scode: their vocabulary overlaps on average by less than 20%. This is explained by the identifier-based concepts typically being at a lower level of abstraction than Scode's concepts.

4.8 Discussion

Beyond evaluating the performance of a single tool, the goal of our evaluation was to explore the theoretical potential and limits of a new concept identification strategy. The findings demonstrate the difficulty of this problem. Both phases of Scode achieved good performance in isolation, but together, the precision remains low. Although there are more precise concept identification approaches, they typically represent concepts using a project-specific terminology, limiting the comparison and understandability of the concepts outside the project's context. Even for Scode, configuration details matter: Using only concepts from small topics, topic representatives, or explicit concepts has a considerable impact on the precision and coverage. Hence, there is currently no single technique to solve all concept identification tasks.

Scode improves the state of the art with regards to identifying recognized concepts from documentation. The evaluation showed the potential of an approach like Scode to identify concepts more consistently. It also elicited challenges to solve to continue improving our approach. In particular, the density of links in the Wikipedia graph, despite our pruning steps, introduced noise in the community search algorithms and generated very large topics that are impractical for developers. Although itself a challenging knowledge engineering task, constructing a graph of Wikipedia articles with fewer, more meaningful links should improve the precision of Scode while reducing the size of its output.

Another aspect to consider when improving concept identification strategies is the different relations between concepts and a project. Some concepts that are not related to the current implementation of a project may still be relevant to know for developers. For example, developers who work with the MD5 cryptographic algorithm should likely be aware of alternative algorithms, such as SHA-256. There may also be a distinction between core concepts implemented by a project, and concepts implemented in third-party libraries. Precisely identifying such categories of relatedness would be a promising avenue for future work.

Finally, there is no inherent reason for Scode to use only header comments as input. Future work could include investigating more types of documentation sources, such as requirements files and issue trackers, and more preprocessing strategies to improve the precision of Scode.

Solutions to these challenges can be integrated within Scode without having to modify the other components. Hence, Scode provides a framework to study different aspects of the concept identification problem.

4.9 Threats to Validity

The intangible nature of concepts and of the relatedness relation introduced threats to the *internal validity* of our results. Judging whether a concept is related to a class is a subjective decision. To control this subjectivity, we prepared a strict coding guide, which in particular avoided speculations about concepts that may be relevant to be aware of, even though they are not directly used by a class. The authors also performed the annotation task entirely. Although this may introduce a bias in the annotations, judging the relevance of a concept to a class requires a considerable effort to understand the class and the concept, as well as their context. Thus, we favored this threat over the one that would have been introduced by less motivated external annotators. We further mitigated the threat of investigator bias by removing

14. An alternative would be to use Scode to identify concepts directly from source code. However, this comparison would be unfair as the wikification service expects natural language inputs. Most prior work uses language models such as LDA to identify concepts.

15. These classes were mostly custom exception types and poorly documented classes.

any indication of the technique that identified each concept during the annotation task.

Performing the end-to-end evaluation using classes as a substitute for projects also poses a threat to validity. This was necessary to design a practical evaluation, as it would be infeasible for annotators to reliably reject unrelated concepts without understanding the entire project. The impact of this threat was mitigated by the fact that, in the context of this work, a concept that is relevant to a single class in a project is relevant to the project. Hence, despite the smaller scope of the evaluation units, the precision of Scode on classes should generalize to projects. This property shows a beneficial aspect of Scode: it can be used to analyze projects at different granularities, from a project to a class level.

Similarly, most of our results focus on concepts rather than topics. We made this decision as prior work have already rigorously evaluated the properties of the topic modeling techniques used by Scode, including the communities' cohesiveness [10], [26], [29], [30] and the use of *pagerank* to select a topic's representative [31]. To ensure these previous results would apply in a software development context, we performed additional evaluations to measure the cohesiveness of topics and the precision of topic representatives.

The evaluation of the topic cohesiveness relied on the judgment of external annotators. In this case, we found that external annotators were a better option as the annotation tasks were better encapsulated and required less contextual knowledge. Nevertheless, the background of each annotator can affect the internal validity of the results. To mitigate this threat, we triangulated results from two complementary metrics—a direct but subjective assessment of the cohesiveness and the success rate of the word intrusion task—and from three annotators working independently. To assess the consistency of the external annotators, we tested the hypothesis that topics for which an annotator succeeded at the word intrusion task would receive higher cohesiveness scores by the other annotators. We observed that all tests were significant at the 0.05 level using Mann–Whitney U tests, except the one that compared cohesiveness scores attributed by annotators B and C to topics for which the word intrusion task was performed by annotator A (p-value of 0.086). This suggests that the results of the evaluation are reliable, despite the subjectivity of cohesiveness.

The different vocabulary used by LDA topics and PengKG's concepts can also introduce a bias in our evaluation. We mitigated this bias by normalizing the concepts presentation during the annotation tasks, i.e., by putting them in lower case. We also removed the occasional clarifying terms in parentheses that are characteristic of Wikipedia titles for the end-to-end evaluation. By doing so, different articles can result in the same concept. During the annotation task and when reporting results, we considered these articles as a single, polysemous concept. We erred on the side of inclusion when judging the relatedness of such polysemous concepts, as trying to find an unrelated sense for each concept would be counter-productive. We did not remove parentheses from Wikipedia titles for the cohesiveness evaluation as the bias favored the LDA baseline.

Finally, our sampling strategies limit the generalizability of our results. For the topic cohesiveness evaluation, we sampled open source Android projects. This sampling frame biases the output of the compared techniques towards Java- and Android-related topics. For the end-to-end evaluation, we used classes instead of larger project components as input units to ensure that it was possible for the annotators to reliably understand the context of each unit when judging the concepts. We also only sampled classes from a single source, i.e., the standard Java Class Library. However, given that this library is intended to provide fundamental classes supporting a wide variety of applications, it covered concepts from a large variety of computing domains.

5 Related Work

Using wikification and community search techniques to identify concepts in software projects is a novel approach, but there exists a notable amount of research in software engineering that requires explicit forms of knowledge or generates a representation of knowledge.

Knowledge Graphs in Software Engineering

Knowledge graphs have enabled the development of techniques to automate many software development activities that previously relied on human judgment. Wang et al. [45] and Zhou [46] proposed approaches to fix software defects by capturing the latent knowledge in various software artifacts. Liu et al. [33] used, among other features, concepts mentioned in documentation to produce a knowledge-aware graph of API elements and generate class summaries tailored to user queries. This knowledge graph also supports the semantic comparison of similar API elements, such as StringBuilder and StringBuffer [34]. Ferrari and Esuli [47] and Ezzini et al. [48] extract subsets of Wikipedia articles related to a target domain, based on Wikipedia categories, to support the identification and resolution of language ambiguities in requirements. Chen et al. [8] created a knowledge base for identifying libraries that offer similar functionalities, but for different languages. For example, it can identify that Gson, Json.NET, and Simplejson are three analogue libraries for handling JSON objects in Java, C#, and Python, respectively. Petrenko et al. also observed in two case studies that self-constructed concept graphs can help efficiently navigate an unfamiliar code base to fix a bug [49]. Our technique can contribute to the improvement of knowledge graphs by supporting a new type of relation: associations between software projects and a project-independent graph of concepts, i.e., Wikipedia articles.

Knowledge Extraction and Representation

To address the need for knowledge-aware software engineering techniques, several concept extraction techniques have been proposed in prior work. A recurrent strategy for these techniques is to extract relevant terms from source code identifiers and generate a semantic structure based on further information in source code. For example, Ratiu et al. designed an approach to build lightweight ontologies by parsing the identifiers to generate concepts and using the syntactic relations between code elements to generate relations between concepts [15]. Falleri et al. instead used linguistic cues within each identifier to infer relations between them, as well as identify low-level implicit concepts [11]. A common limitation of these early techniques is that the number of concepts grows quickly as more identifiers are considered. A core issue to solve for extracting useful concept is to filter out irrelevant terms and cluster similar concepts in the output.

To partly address this issue, Yang and Tan proposed a technique to mine pairs of semantically related terms used in a software project [50]. Kelly et al. used LDA to group concepts extracted from identifiers into a practical number of topics, and further group topics generated for different projects using the K-means clustering algorithm [12]. Abebe and Tonella designed and compared several concept extraction and filtering techniques [16], [51], [52]. In particular, they studied how to distinguish high-level domain concepts from low-level implementation concepts [52], and they compared a topic modeling technique similar to LDA and two keyword-based techniques, one interactive and the other fully automated, to filter the ontologies they built [16]. They found that the interactive keyword-based technique outperforms the other automated techniques, even if the required interaction is minimal. Our work complements this prior work by exploring techniques to extract concepts from documentation rather than source code and by focusing on concepts that are easily interpretable by developers and consistent across projects.

Abstract Knowledge Representations

Prior work has used abstract knowledge representations such as LDA and vector space embeddings to operationalize semantic exchanges. LDA has been used, for example, on Stack Overflow posts to elicit topics relevant to developers in general [53] or related to non-functional requirements [54] and mobile development [55]. Vector space embeddings have been used to translate various elements, such as words or API elements, into Euclidean vectors. Ye et al. use this technique to compute a similarity score between natural language text and source code, and propose a bug localization approach based on this score [56]. Nguyen et al. proposed an approach to embed API elements in a vector space and studied the correlation between usage patterns of the API elements and properties of their respective embeddings [57]. These approaches, however, do not target a specific objective in terms of how the captured knowledge can be represented and interpreted. LDA produces large overlapping sets of terms that are specific to the training corpus, and with considerable noise. Vector embeddings produce numerical vectors whose dimensions are, by design, meaningless. In contrast, Scode uses a mature community-generated knowledge base, Wikipedia, to identify recognizable concepts.

Natural Language Understanding

The extraction of structured information from natural language documents is an active area of research in the natural language understanding community. Prior work has developed techniques to automatically extract concept dependencies from on-line and university courses, a problem similar to the one addressed by our approach [58], [59]. Linking documents to entries of a knowledge base, a task known as *concept linking* or *entity linking*, is particularly relevant to our work, as we leveraged a state-of-the-art concept linking technique to identify Wikipedia articles from the documentation of software systems [9], [21], [27]. The concept linking task is itself a part of the knowledge base population problem that focuses on generating comprehensive knowledge bases [60]. However, applying such techniques designed for a general context on software documents requires specific adaptations, as we found in a prior study [23]. Furthermore, a notable component of our approach is a post-processing step that aggregates cohesive concepts into topics.

Our work is also related to the generation of knowledge graphs from large document corpora. Notable large knowledge graphs include Wikidata [61], DBpedia [62], and BabelNet [28]. However, because these knowledge graphs are not specific to software development, they often cannot be used directly for software engineering applications, despite the vast knowledge they capture. Researchers have proposed techniques to generate software-specific knowledge graphs using association rule mining [63] or heuristics based on grammatical dependencies identified by natural language parsers [64]. Although the latter techniques are specific to the software domain, they lack the inclusion of high-level concepts such as algorithms or design patterns. Our work can bridge this gap by providing a way for software-specific knowledge graph generation techniques to incorporate links to abstract concepts. Related to knowledge graph construction is the problem of automated glossary construction, which attempts to identify relevant terms specific to a project [13], [14]. Glossaries provide a basis for developers to precisely express and discuss requirements and solutions of the software project. However, they are typically only relevant within the context of a single project.

Traceability

Finally, this work relates to the traceability recovery problem, as Scode depends on the identification of links between software artifacts, i.e., documentation, and entries of the Wikipedia knowledge base. In the software engineering domain, many researchers have studied the idea of traceability between software artifacts [65], and proposed techniques to identify mentions of API elements in natural language texts [66], [67], [68], [69]. Our work complements these advances in the traceability problem by proposing a new data source to link software artifacts to.

6 Conclusion

Understanding the conceptual context of a software project is useful to the project's contributors and users of its API, as well as researchers and tool-makers who design semantic-aware techniques, e.g., to locate bugs or choose third-party libraries. However, concepts that are relevant to a software project are often only implicitly known, and knowledge transfer is still an *ad hoc* process between developers. Precisely understanding and representing this exchange of knowledge is challenging. To address this issue, we investigated the use of wikification and community search techniques to identify concepts from a project's documentation. Our approach, named Scode, generates concepts whose meaning is generally recognized and externalized in a popular independent knowledge base, Wikipedia.

Our comprehensive evaluation revealed that Scode is more consistent when retrieving recurring concepts across projects. However, although Scode is more precise than other Wikipedia-based concept identification techniques, future work is needed to reach the precision level of techniques that are not tied to a knowledge base. One promising direction for improvement is to carefully engineer a graph of computing concepts that is less densely connected than Wikipedia's wikilinks, especially around general computing concepts. Such a graph would reduce the noise produced by community search algorithms.

ACKNOWLEDGMENTS

We are grateful to the anonymous annotators for helping with the evaluation of the community search algorithms, and to the researchers who made the tools used in this work available. We also thank Deeksha Arya and Jin L. C. Guo for their feedback on a preliminary version of this manuscript. This work is funded by the Natural Sciences and Engineering Research Council of Canada and the Fonds de recherche du Québec – Nature et technologies.

REFERENCES

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[2] M. P. Robillard and C. Treude, "Understanding Wikipedia as a resource for opportunistic learning of computing concepts," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 2020, pp. 72–78.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the Working Conference on Reverse Engineering*, 1993, pp. 482–498.

[4] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[5] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 4, pp. 23:1–23:34, 2012.

[6] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.

[7] M. M. Rahman and C. K. Roy, "Improving IR-based bug localization with context-aware query reformulation," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 621–632.

[8] C. Chen, Z. Xing, and Y. Liu, "What's Spain's Paris? mining analogical libraries from Q&A discussions," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1155–1194, 2019.

[9] W. Shen, J. Wang, and J. Han, "Entity linking with a knowledge base: Issues, techniques, and solutions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 2, pp. 443–460, 2014.

[10] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *The VLDB Journal*, vol. 29, no. 1, pp. 353–392, 2020.

[11] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a WordNet-like identifier network from software," in *Proceedings of the 18th IEEE International Conference on Program Comprehension*, 2010, pp. 4–13.

[12] M. B. Kelly, J. S. Alexander, B. Adams, and A. E. Hassan, "Recovering a balanced overview of topics in a software domain," in *Proceedings of the IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 135–144.

[13] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Automated extraction and clustering of requirements glossary terms," *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 918–945, 2017.

[14] C. Wang, X. Peng, M. Liu, Z. Xing, X. Bai, B. Xie, and T. Wang, "A learning-based approach for automatic construction of domain glossary from source code and documentation," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 97–108.

[15] D. Ratiu, M. Feilkas, and J. Jurjens, "Extracting domain ontologies from domain specific APIs," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 203–212.

[16] S. L. Abebe and P. Tonella, "Extraction of domain concepts from the source code," *Science of Computer Programming*, vol. 98, pp. 680–706, 2015.

[17] "ISO/IEC/IEEE international standard - Systems and software engineering–Vocabulary," Standard ISO/IEC/IEEE 24765:2017(E), 2017.

[18] E. Meij, W. Weerkamp, and M. de Rijke, "Adding semantics to microblog posts," in *Proceedings of the 5th ACM International Conference on Web Search and Data Mining*, 2012, pp. 563–572.

[19] M. Cornolti, P. Ferragina, and M. Ciaramita, "A framework for benchmarking entity-annotation systems," in *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 249–260.

[20] J. Szymański and M. Naruszewicz, "Review on wikification methods," *AI Communications*, vol. 32, no. 3, pp. 235–251, 2019.

[21] R. Usbeck, M. Röder, A.-C. Ngonga Ngomo, C. Baron, A. Both, M. Brümmer, D. Ceccarelli, M. Cornolti, D. Cherix, B. Eickmann, P. Ferragina, C. Lemke, A. Moro, R. Navigli, F. Piccinno, G. Rizzo, H. Sack, R. Speck, R. Troncy, J. Waitelonis, and L. Wesemann, "GERBIL: General entity annotator benchmarking framework," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1133–1143.

[22] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, 2010, pp. 939–948.

[23] M. Nassif and M. P. Robillard, "Wikifying software artifacts," *Empirical Software Engineering*, vol. 26, no. 2, pp. 31:1–31:31, 2021.

[24] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

[25] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[26] E. Akbas and P. Zhao, "Truss-based community search: A truss-equivalence based indexing approach," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, 2017.

[27] A. Moro, A. Raganato, and R. Navigli, "Entity linking meets word sense disambiguation: a unified approach," *Transactions of the Association for Computational Linguistics*, vol. 2, pp. 231–244, 2014.

[28] R. Navigli and S. P. Ponzetto, "BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network," *Artificial Intelligence*, vol. 193, pp. 217–250, 2012.

[29] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 276–287, 2015.

[30] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang, "On minimal steiner maximum-connected subgraph queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2455–2469, 2017.

[31] D. Lizorkin, O. Medelyan, and M. Grineva, "Analysis of community structure in Wikipedia," in *Proceedings of the 18th International Conference on World Wide Web*, 2009, pp. 1221–1222.

[32] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, pp. 107–117, 1998.

[33] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu, "Generating query-specific class API summaries," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 120–130.

[34] Y. Liu, M. Liu, X. Peng, C. Treude, Z. Xing, and X. Zhang, "Generating concept based API element comparison using a knowledge graph," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 834–845.

[35] E. Gabrilovich and S. Markovitch, "Computing semantic relatedness using Wikipedia-based explicit semantic analysis," in *Proceed-*

ings of the 20th International Joint Conference on Artificial Intelligence, 2007, pp. 1606–1611.

[36] D. Carvalho, Ç. Çallı, A. Freitas, and E. Curry, "EasyESA: A low-effort infrastructure for explicit semantic analysis." in *Proceedings of the 13th International Semantic Web Conference*, 2014, pp. 177–180.

[37] R. R. Sokal and C. D. Michener, "A statistical method for evaluating systematic relationships," *The University of Kansas Science Bulletin*, vol. 38, pp. 1409–1438, 1958.

[38] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[39] F-Droid Limited and Contributors, "F-Droid," 2022, accessed 2022-03-13. [Online]. Available: https://www.f-droid.org/

[40] A. K. McCallum, "MALLET: A machine learning for language toolkit," 2002, http://mallet.cs.umass.edu.

[41] J. Chang, S. Gerrish, C. Wang, J. L. Boyd-Graber, and D. M. Blei, "Reading tea leaves: How humans interpret topic models," in *Advances in neural information processing systems*, 2009, pp. 288–296.

[42] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[43] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[44] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[45] L. Wang, X. Sun, J. Wang, Y. Duan, and L. Bin, "Construct bug knowledge graph for bug resolution," in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion*, 2017, pp. 189–191.

[46] C. Zhou, "Intelligent bug fixing with software bug knowledge graph," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 944–947.

[47] A. Ferrari and A. Esuli, "An NLP approach for cross-domain ambiguity detection in requirements engineering," *Automated Software Engineering*, vol. 26, no. 3, pp. 559–598, 2019.

[48] S. Ezzini, S. Abualhaija, C. Arora, M. Sabetzadeh, and L. C. Briand, "Using domain-specific corpora for improved handling of ambiguity in requirements," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, 2021, pp. 1485–1497.

[49] M. Petrenko, V. Rajlich, and R. Vanciu, "Partial domain comprehension in software evolution and maintenance," in *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008, pp. 13–22.

[50] J. Yang and L. Tan, "SWordNet: Inferring semantically related words from software context," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1856–1886, 2014.

[51] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Proceedings of the IEEE 18th International Conference on Program Comprehension*, 2010, pp. 156–159.

[52] ——, "Towards the extraction of domain concepts from the identifiers," in *Proceedings of the 18th Working Conference on Reverse Engineering*, 2011, pp. 77–86.

[53] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in Stack Overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.

[54] J. Zou, L. Xu, W. Guo, M. Yan, D. Yang, and X. Zhang, "Which non-functional requirements do developers focus on? an empirical study on Stack Overflow using topic analysis," in *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 446–449.

[55] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using Stack Overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.

[56] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 404–415.

[57] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 438–449.

[58] C. Liang, J. Ye, Z. Wu, B. Pursel, and C. L. Giles, "Recovering concept prerequisite relations from university course dependencies," in *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, 2017, pp. 4786–4791.

[59] L. Pan, C. Li, J. Li, and J. Tang, "Prerequisite relation learning for concepts in MOOCs," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1447–1456.

[60] H. Ji and R. Grishman, "Knowledge base population: Successful approaches and challenges," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, 2011, pp. 1148–1158.

[61] D. Vradečić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Communications of the ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[62] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia – a large-scale, multilingual knowledge base extracted from Wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[63] C. Chen and Z. Xing, "Mining technology landscape from Stack Overflow," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.

[64] X. Zhao, Z. Xing, M. A. Kabir, N. Sawada, J. Li, and S.-W. Lin, "HDSKG: Harvesting domain specific knowledge graph from content of webpages," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evluation and Reengineering*, 2017, pp. 56–67.

[65] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Proceedings of the on Future of Software Engineering*, 2014, pp. 55–69.

[66] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering*, 2013, pp. 832–841.

[67] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract API mentions from informal natural language discussions," in *IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 389–399.

[68] D. Ye, L. Bao, Z. Xing, and S.-W. Lin, "APIReal: an API recognition and linking approach for online developer forums," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3129–3160, 2018.

[69] S. Ma, Z. Xing, C. Chen, C. Chen, L. Qu, and G. Li, "Easy-to-deploy API extraction by multi-level feature embedding and transfer learning," *IEEE Transactions on Software Engineering*, p. 15 pages, 2019, to appear.

Mathieu Nassif is a Ph.D. student at McGill University. His research focuses on the extraction, representation, and manipulation of knowledge in software systems to optimize the contribution of developers. Mathieu received a M.Sc. in Computer Science from McGill University in 2018. His thesis explored a flexible approach to embed documentation directly in source code to reduce the redundancy of information while improving documentation quality.

Martin P. Robillard is a Professor in the School of Computer Science at McGill University. He received his Ph.D. in Computer Science from the University of British Columbia. His research is in the area of software engineering, with an emphasis on the human-centric aspects of software development.