Generating Query-specific Class API Summaries

Mingwei Liu*
Fudan University
China

Xin Peng*†
Fudan University
China

Andrian Marcus
The University of Texas at Dallas
USA

Zhenchang Xing
Australian National University
Australia

Wenkai Xie*
Fudan University
China

Shuangshuang Xing*
Fudan University
China

Yang Liu*
Fudan University
China

ABSTRACT

Source code summaries are concise representations, in form of text and/or code, of complex code elements and are meant to help developers gain a quick understanding that in turns help them perform specific tasks. Generation of summaries that are task-specific is still a challenge in the automatic code summarization field. We propose an approach for generating on-demand, extrinsic hybrid summaries for API classes, relevant to a programming task, formulated as a natural language query. The summaries include the most relevant sentences extracted from the API reference documentation and the most relevant methods.

External evaluators assessed the summaries generated for classes retrieved from JDK and Android libraries for several programming tasks. The majority found that the summaries are complete, concise, and readable. A comparison with summaries produce by three baseline approaches revealed that the information present only in our summaries is more relevant than the one present only in the baselines summaries. Finally, an extrinsic evaluation study showed that the summaries help the users evaluating the correctness of API retrieval results, faster and more accurately.

CCS CONCEPTS

• **Information systems** → **Summarization**; • **Software and its engineering** → **Documentation**.

KEYWORDS

Code summarization, API, Knowledge Graph, Code retrieval

*M. Liu, X. Peng, W. Xie, S. Xing, Y. Liu are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
†X. Peng is the corresponding author (pengxin@fudan.edu.cn).

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338971

ACM Reference Format:
Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating Query-specific Class API Summaries. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338971

1 INTRODUCTION

Automated software summarization is the process of generating a concise representation of one or more software artifacts, that conveys the information needed by a software stakeholder to perform a particular software engineering task [31]. Researchers have proposed techniques for the summarization of many software artifacts including: source code, posts on question and answer forums, bug reports and issue requests, user manuals, code and product reviews, code changes, etc. [36] One of the goals of these summaries is to facilitate quick understanding of the software artifacts. For example, during code retrieval tasks, the summaries of the retrieved code help developers decide quickly whether they are irrelevant, or whether they require in depth analysis to establish relevancy.

Software artifacts are usually comprised of natural language text and/or source code. The produced summaries may be textual, contain source code only, or hybrid (i.e., contain both). Approaches that generate textual summaries for textual software artifacts rely on standard techniques from the field of automated (natural language) text summarization [24, 25, 31, 36, 40, 41]. More challenging are the approaches that summarize code-base and hybrid artifacts, where standard techniques do not apply. Among these, hybrid documents and summaries are difficult as they usually require establishing relationships between the text and source code components.

A common trait of the automated code summarization techniques is that the generated summaries are independent from the user task (i.e., what the summary is needed for). Binkley et al. [13] studied human written summaries for the same code, but different tasks and concluded a person produces different summaries of the same code, for different task. It also highlighted that summarizing non-trivial unfamiliar code is extremely challenging and it is unclear what level of detail is required for a task-specific summary. In addition, McBurney and McMillan [28] showed that there are significant differences between source code summaries written by

the authors of the code or by the readers (i.e., users) of the same code. Also, human written summaries have different properties than automatically generated summaries. A major challenge left unaddressed by researchers is to determine what type of summary fits best a given user task. While this paper does not solve this problem entirely, it achieves one important step forward, by introducing and evaluating a technique (named *KG-APISumm*) that automatically generates source code summaries based on the user programming task, expressed as a natural language query.

Many developer tasks are related to using certain APIs (Application Programming Interfaces). To find the relevant APIs, developers often resort to API documentations [15, 26, 42, 44, 48]. In this paper, we focus on **automatically generating hybrid extractive summaries for classes, supporting developers in finding APIs relevant to their programming task**. The summaries are generated on-demand, in response to a user textual query, which describe the programming task at hand. The summaries are *hybrid* because they include *summary sentences* attached to the relevant methods of the class being summarized. *KG-APISumm* is an *extractive* summarization approach because the *summary sentences* are selected from the reference documentation. The summaries are *task-specific* because the extracted sentences and methods are the ones most related to the user query. In other words, a class may have two different summaries for two different queries.

Most existing automated code summarization techniques rely on the summarized code artifact only for generating the summary. Relationships between code elements and other information sources are rarely used, hence they cannot be used for creating programming task-specific summaries. Only a few summarization techniques rely on extracting code related information from sources external to the summarized code element [28, 34, 35, 37, 39, 48] (see Section 5). One problem is that information about the APIs, relevant for solving a programming task, is scattered across multiple information sources and they relate to each other in more than one way. Our solution to this problem is the construction of an *API knowledge graph* (API KG), which represents fine-grained information about a library, at method and sentence level. The ability to reason about individual sentences in documentation is essential for generating query specific and succinct summaries.

We constructed an API KG for JDK and Android and conducted several empirical studies for the intrinsic and extrinsic evaluation of the class API summaries, involving external evaluators. The evaluators found the summaries to be complete, concise, and readable. In addition, they contain relevant information, which is missing in summaries obtained with three baseline summarization approaches. Finally, the summaries helped users identify APIs relevant for a programing tasks, faster and more accurately.

2 KG-BASED CLASS API SUMMARIZATION

KG-APISumm takes as input a natural language user query Q, describing the developer task, a class C from an existing library L, and the API knowledge graph of the library L (*API KG(L)*).

The *API KG(L)* is used to extract up to S sentences describing C's functionality and up to M methods m_i in C, most relevant to Q. For each m_i *KG-APISumm* also includes in the summary up to S most relevant sentences extracted from the reference documentation. M and S are customizable by the user, which determine the maximum size of the summaries. An example, with *M=3* (i.e., up to three methods are included) and *S=2* (i.e., each method and the class are described by two sentences) is shown in Figure 1.

Query: Add all files recursively from root with Java 8 Stream
Summary for java.nio.file.Files
Description:
- This class consists exclusively of static methods that operate on files, directories, or other types of files.
- In most cases, the methods defined here will delegate to the associated file system provider to perform the file operations.

Relevant methods:
- *Files.walk(java.nio.file.Path,java.nio.file.FileVisitOption...)*
  - Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
  - The file tree is traversed depth-first, the elements in the stream are Path objects that are obtained as if by resolving the relative path against start.
- *Files.walk(java.nio.file.Path,int,java.nio.file.FileVisitOption...)*
  - Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
  - The Stream returned is guaranteed to have at least one element, the starting file itself.
- *Files.walk(java.nio.file.Path,int,java.nio.file.FileVisitOption)*
  - Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
  - The Stream returned is guaranteed to have at least one element, the starting file itself.

Figure 1: Summary example with M=3 methods and S=2 sentences per method/class

For generating the summaries, *KG-APISumm* needs to relate API elements to each other and to individual sentences from the reference documentation. The API KG captures such relationships.

2.1 API KG Construction

For Constructing the *APIKG*(L), we extract all the the API definitions and descriptions from L's reference documentation.

The first step in building the API KG is the extraction of structural information of the APIs. The API KG is later enriched with relationships between API elements and API descriptions. The high-level schema of the API KG is shown in Figure 2, which contains entities (circles) and their relationships (arrows). The API KG includes two parts, i.e., the structural knowledge (white circles) and the descriptive knowledge (rectangles).

The **structural knowledge** describes the structure of the APIs entities (e.g., API packages, classes, methods/fields) and the signatures of the APIs (e.g., parameters and return values of API methods). It includes various API entities, as well as the containment and generalization/implementation relations between them. In addition, each API entity has properties such as, fully qualified name, added in version, API documentation URL, etc. The structural knowledge also describes the parameters, return values, thrown exceptions of methods and their types. It also includes the "*seeAlso*" relation between API elements, which indicates closely relevant API elements (e.g., API methods that provide similar functionalities).

The **descriptive knowledge** describes the functionalities and directives of the APIs. These descriptions are expressed as natural language sentences and will be further processed in the **knowledge fusion** part for identifying concepts and linking them with each other and with general software concepts.

2.1.1 Structural Knowledge Extraction. API entities and their relations in the structured knowledge are extracted from the class/interface declarations, class/interface member declarations, and method declarations in the API reference documentation. We developed a web crawler, which obtains the API reference documentation from the

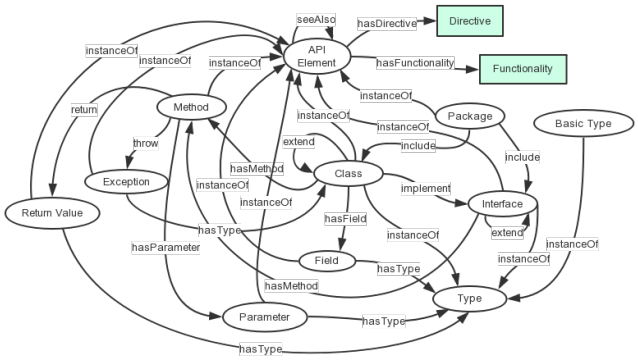

Figure 2: Schema of the Structural Knowledge Graph

web. We also developed a parser, based on the structure of specific API reference documentation (e.g., JDK and Android API documentations) to extract API entities and their properties and relations.

Note that parameters and return values are independent API elements that can be shared by multiple methods. For example, two methods may have the same return value. Methods that accept parameters or return values with the same meanings can be indirectly connected. We determine the similarity of the parameters or return values, of two methods by matching the names (for parameters only), types, and the descriptions.

2.1.2 Descriptive Knowledge Extraction. We use the natural language text descriptions of various API elements, extracted from API reference documentation, as input for the descriptive knowledge extraction. These descriptions include: package, class/interface, field, method, method return value, and method parameter descriptions. The extracted text descriptions are processed as follows:

- All HTML tags and separate code segments are removed. Code elements mentioned in sentences are preserved.
- Each text description is split into sentences.
- The sentences are classified into different types.

As reported by Maalej and Robillard [26], about a half of the documentation units attached to API class members in JDK and .NET contain information of little or no value. The purpose of sentence classification is to exclude meaningless sentences and identify sentences relevant for different types of descriptive knowledge. Based on the findings of previous studies [26] and our analysis of the JDK and Android API reference documentations, we define the following sentence types, relevant for descriptive knowledge extraction. **[Functionality]**—description of the functionality of API entities. Example: *"Object used to report key and button events"*. **[Directive]**—description about the usage of the API such as, correct or incorrect usage situations, constraints on method parameters, situations of exception throwing. Example: *"IllegalArgumentException: if the modifiers parameter contains invalid modifiers"*. **[Other]**—all other situations, usually implementation details. Example: *"There are two ways to programmatically determine the version number"*.

We use a back propagation (BP) neural network [21] to train a classifier for classifying the sentences into one of the three types. To this end, we transform the sentences into the inputs to the BP neural network in the following way: (1) convert each sentence into a bag of words using standard preprocessing procedure (i.e., tokenization, stop word removal, and lemmatization); (2) generate a vector for each sentence by averaging the vectors of words from the sentence; and (3) use Word2vec [30] to train word vectors based on the corpus of all the sentences from the API reference documentation.

To prepare the training data, we randomly select a subset of the sentences and manually label each of them into one of the three types. The details on the training and calibration of the classifier in our implementation is presented in Section 3.1.

Finally, for each sentence of an API element that is classified as **[Functionality]** or **[Directive]**, we create a "*hasFunctionality*" or "*hasDirective*" relation between the API element and the sentence.

2.1.3 Knowledge Fusion. We link concepts referenced in the descriptions of different API elements with each other and also with concepts from more general knowledge graphs. The these concept relationships help determining relationships between API elements and their relevance to user queries.

The descriptions of different API entities may refer to various API-related concepts such as, "*system service*", "*download*", "*system notification*". For simplicity we will refer to these concept, which are referred to in API descriptions, as *API concepts*. Identifying these concept references can reveal semantic relationships between API elements that share them. For example, different API classes or methods that provide similar or related functionalities are semantically related, even if they are not structurally related (e.g., through a direct reference). Some of the *API concepts* can be further linked with generic software-related concepts (which we call simply as *software concepts*) from existing general knowledge graphs such as, Wikidata. These general knowledge graphs reveal further relationships between concepts. For example, the *API concept* "download" is linked to the *software concept* "download", and then further connected with "service" and "upload" via the relations defined in Wikidata. The use of generic knowledge graphs allow us to determine relationships that are not explicit in the structure or the reference documentation of the library.

The knowledge fusion process includes three steps: (1) API concept reference extraction; (2) cross sentence concept fusion; and (3) software concept fusion. It requires a preliminary step for generic concept filtering, which is independent of the library.

Generic Concept Filtering. We extract the *software concepts* from general knowledge graphs, such as Wikidata.

Wikidata is a free and open knowledge graph for general knowledge and includes many software-related concepts (e.g., "*service*", "*upload*", "*download*", "*computer network*") and relations (e.g., <"*download*", "*part of*", "*service*">, <"*download*", "*opposite of*", "*upload*">). However, a large part of concepts and relations in Wikidata are irrelevant to software. Hence, we need to use a filter to select only *software concepts* from general knowledge graphs for the API KG.

We select *software concepts* from general knowledge graphs and add these concepts, together with their relations to the API KG.

Similar to the sentence classification in the descriptive knowledge extraction (see Section 2.1.2), we use a BP neural network to train a text classifier to distinguish between software-related concepts and irrelevant concepts. The classification is based on the text description of each concept in the general knowledge graph. For example, each concept in Wikidata has a corresponding Wikipedia[12] article describing it. We transform the concept descriptions into the

inputs to the BP neural network in the following way: (1) convert the description of each concept into a bag of words using standard preprocessing procedure (i.e., tokenization, stop word removal, and lemmatization); (2) generate a vector for each concept by averaging the vectors of words from its description; and (3) use Word2vec [30] to train word vectors based on the corpus of all the descriptions of generic concepts. As training data, we randomly selected a subset of the generic concepts and manually labeled each of them as software-related or not. Details on the training and calibration of the classifier used in our implementation are in Section 3.2.

API Concept Reference Extraction. We identify parts of descriptive sentences of API elements that correspond to *API concepts*.

Concept references are extracted from descriptive sentences of API elements by identifying noun phrases. More complex extraction techniques may be employed, but investigating their applicability is subject of future work. We use a parser to parse each descriptive sentence, following the standard process of tokenization, POS tagging, lemmatization, constituency parsing, and dependency parsing. We extract all the atomic noun phrases, which do not include other smaller noun phrases, from the parse tree. Then, we remove stop words from the noun extracted phrases and eliminate the noun phrases that only contain special characters (e.g., "#", "!"). All the remaining atomic noun phrases from the descriptive sentence are considered *candidate concept references*.

Cross Sentence Concept Fusion. We link different descriptive sentences of API elements that refer to the same *API concept*. The identified *API concepts* are added to the API KG with links from the descriptive sentences that reference the concepts.

For identifying candidate concept references (i.e., noun phrases) that refer to the same *API concept*, we cluster the candidate concept references based on their lexical similarity and context similarity. For two candidate concept references n_1 and n_2 (noun phrases in our case), their similarity $sim(n_1, n_2)$ is the linear combination of two similarities ($w_1 + w_2 = 1$).

$$sim(n_1, n_2) = w_1 \times sim_{lex}(n_1, n_2) + w_2 \times sim_{con}(n_1, n_2) \quad (1)$$

The lexical similarity (sim_{lex}) of n_1 and n_2 is the Jaccard similarity [47] between their token sets $Token(n_1)$ and $Token(n_2)$.

$$sim_{lex}(n_1, n_2) = \frac{|Token(n_1) \cap Token(n_2)|}{|Token(n_1) \cup Token(n_2)|} \quad (2)$$

The context similarity (sim_{con}) of n_1 and n_2 is the normalized cosine similarity between the text vectors of the two API documentation paragraphs where n_1 and n_2 reside. These paragraphs are extracted from the reference documentation webpages based on HTML tags, such as "<p>", "
". We choose paragraphs instead of larger context, such as documents, because we observed that relevant statements of a concept reference are often in close proximity to each other. In the following equation, $V_p(n)$ is the k-dimension text vector of the paragraph where n reside, Sim_{cos} is the cosine similarity between two vectors. Given a paragraph, its vector representation is generated in a way similar to the sentence vector generation used in sentence classification for descriptive knowledge extraction (see Section 2.1.2), i.e., averaging the vectors of the bag of words of the paragraph, after preprocessing.

$$sim_{con}(n_1, n_2) = \frac{Sim_{cos}(V_p(n_1), V_p(n_2)) + 1}{2} \quad (3)$$

Based on the similarity between each pair of candidate concept references, we calculate their distance as $1 - sim(n_1, n_2)$ and use a hierarchical clustering algorithm [43] to cluster all the candidate concept references. Initially, each candidate concept reference is considered a cluster. In each of the following iterations, the two clusters that have the smallest distance are merged together. The distance between two clusters is calculated by averaging the distance between the candidate concept references in the two clusters. The iterative process ends when the highest inter-cluster similarity is lower than a given threshold (explained later in Section 3.2).

For each cluster, the candidate concept reference with the highest average similarity with the other ones in the same cluster, is selected and used as the name of the cluster. The name of the cluster is added to the API knowledge graph as a new *API concept*. Finally we add a "*referTo*" relation from each of the descriptive sentences where the API concept references in the cluster reside, to the new *API concept*.

Software Concept Fusion. We link *API concepts* or API entities to generic software-related concepts from the general KG (a.k.a. *software concepts*). Note that software concept fusion have different meaning for an *API concept* versus an API entity: the former means that the *API concept* is relevant to the *software concept*; the latter means that the functionality of the API entity is relevant to a *software concept*.

For each *API concept*, we collect all the descriptive sentences that include references to it. For each *software concept*, we use its text description from the general knowledge graph (i.e., the corresponding Wikipedia article). Then for each *API concept* or *software concept*, we convert its description into a bag of words using standard preprocessing procedure (i.e., tokenization, stop word removal, and lemmatization), and generate a vector for the concept by averaging the vectors of words from its description. We use Word2vec [30] to train the word vectors based on the combined corpus of *API concepts* and *software concepts*. The corpus includes two parts: (1) all the text descriptions of the API reference documentation and (2) all the text descriptions of the *software concepts*.

Given an *API concept ac*, we compare it with all the *software concepts* and produce a set of *candidate software concepts* that have at least one common token in their names (including their aliases). Then, for each *candidate software concept sc*, we estimate the text similarity between *ac* and *sc*, as the cosine similarity between their vectors. If the similarity is higher than a given threshold, we regard *ac* as a reference to *sc* and add a "*relatedTo*" relation from *ac* to *sc*.

Software concept fusion for API entities is conducted in a similar way. To facilitate the identification of *candidate software concepts*, we automatically generate a special alias for each API entity by splitting its short name (i.e., the part after the last dot of the fully qualified name) by camel case and underscore. The vector for an API entity is generated based on all its descriptive sentences.

2.2 Summary Generation

For generating the summaries, we need the *classes*, *methods*, *sentences*, and their relationships from the API KG. We compute a relevance score between user query and these entities. Because each entity in the knowledge graph has a corresponding document, we could directly calculate the similarity between the query and the entity's document by using some document similarity model

approach (e.g., from text retrieval). However, such approaches tend to perform poorly for small documents, such as method entities that only have a few sentences as description. They do not take advantage of the relationships between entities (e.g., structural relations between APIs) and knowledge from the general KG.

So we design a KG-based similarity between the entity and the query, based on their textual and concept similarities with the user query. *Textual similarity* measures the similarity between the semantic representations of their text content, which can be learned from the text corpus. *Concept similarity* measures the similarity between the semantic representations of their corresponding concepts, which can be learned from the API knowledge graph.

Specifically, given a query q, its similarity with a candidate entity e is calculated with a linear combination of their textual and concept similarities ($w_1 + w_2 = 1$).

$$KGSim(q, e) = w_1 \times Sim_{text}(q, e) + w_2 \times Sim_{concept}(q, e) \quad (4)$$

The text semantics similarity between q and e (i.e., $Sim_{text}(q, e)$) is calculated based on their text vectors. Their text vectors are generated in the same way as the sentence vector, used for sentence classification (see Section 2.1.2), by averaging their word vectors trained with the corpus collected from all knowledge graph entity documents, which is the combined corpus of *API concepts* and *software concepts*. The word vectors of e are the vectors of the words from its document. The concept similarity between q and e (i.e., $Sim_{concept}(q, e)$) is calculated using their concept vectors, which are generated based on the API knowledge graph. For each entity in the API KG, we use node2vec [18], a scalable graph feature learning approach, to train a graph vector. The graph vector reflects the structural features of an entity in the KG. The concept vector of e can be represented by its graph vector.

We represent the query q in the same vector space as follows:

1) *Candidate entities selection.* We extract keywords from q, including concept references and verbs. The concept reference extraction is done in the same way as the API concept reference extraction (see Section 2.1.3). Verbs usually map to method names. In this way, we reduce the query to a set of keywords. For each keyword, we identify a set of candidate entities. The set of candidate entities related to the query is the union of the candidate entities sets found for the keywords. The identification is based on whether their names (including aliases), contain the keyword. Specifically, a sentence entity's name is the sentence itself. To increase the likelihood of matching entities, we automatically generate a special alias for each API entity by splitting its short name (i.e., the part after the last dot of the fully qualified name) by camel case and underscore (e.g., for class `java.math.BigDecimal` we generate an alias "big decimal").

2) *Text similarity calculation and filtering.* Estimate the textual similarity between each candidate entity and q by calculating the cosine similarity between their text vectors. If the text similarity of an entity to the query is less than a certain threshold t, the entity is removed from the set of candidate entities.

3) *Calculate the concept vector of the query.* We use the concept vectors of the set of candidate entities to estimate the vector representation of the query q. The easiest way is to average the graph vectors of all candidate entities. Then the query q can be mapped to the central point of the set of mapped related candidate entities in the API KG. Each candidate entity e is assigned a weight considering its relevance and keyword importance to the query (see Equation 5 and 6):

$$Importance(e, q) = \sum_{t \in query} \frac{TFIDF(t)}{|C_t|} \quad (5)$$

$$weight_{e,q} = Importance(e, q) \times Sim_{text}(q, e) \quad (6)$$

In Equation 5, $TFIDF(t)$ the is the TFIDF value of one keyword t in query q. $|C_t|$ is the number of candidate entities extracted by the keyword t. An entity A can be extracted by two keywords and a keyword can extract multiple entities. For example, if the number of entities extracted by these two keywords are 10, 20, and their TFIDF scores are 0.2, 0.3, respectively. The importance score for this entity A is $\frac{0.2}{10} + \frac{0.3}{20} = 0.035$.

Based on the text vectors (V_{text}) and concept vectors ($V_{concept}$) of q and e, their textual similarity and concept similarity are calculated with the following equations, where Sim_{cos} means the cosine similarity between two vectors.

$$Sim_{text}(q, e) = \frac{Sim_{cos}(V_{text}(q), V_{text}(e)) + 1}{2} \quad (7)$$

$$Sim_{concept}(q, e) = \frac{Sim_{cos}(V_{concept}(q), V_{concept}(e)) + 1}{2} \quad (8)$$

Finally, with $KGSim(q, e)$ from Equation 4, *KG-APISumm* ranks the API methods and descriptive sentences for the class and each method that are relevant to a user query. Then, it selects the top M methods (there could be fewer than M) for each method and the class it selects the top S sentences (there could be fewer than S).

3 IMPLEMENTATION

We constructed an API knowledge graph for JDK 1.8[6] and Android API 27[2].

3.1 Knowledge Extraction Implementation

We use Beautiful Soup[3], a Python library for parsing HTML and XML documents, to parse the HTML pages of API reference documentations for knowledge extraction. We also use NLTK[7], a Python library for text processing, to implement text preprocessing (i.e., tokenization, stop word removal, and lemmatization) in descriptive knowledge extraction.

To train the sentence classifier, we developed a web-based tool for sentence labeling and invited 24 master and PhD students to label sentences that were randomly selected from the API documentations. These students have between 2 and 5 years (average 3) of Java and Android development experience and 10 of them have industrial internship experience of at least 6 months. Each sentence is independently labeled as one of the three types (i.e., **[functionality]**, **[directive]**, or **[other]**) by two students. The students have access to the complete context of the sentence (i.e., the reference documentation). For the sentences that were labeled differently, a third student was assigned to give an additional label, to resolve the conflict. The label with two of three votes was selected as final. Finally, we obtained 8,345 labeled sentences: 4,167 labeled as **Functionality**, 3,557 as **Directive**, and 621 as **Other**.

Based on the sentence corpus, we trained an 128-dimensional word vector for each word using the word2vec algorithm provided by gensim[30]. We set the following hyperparameters for the training, based on the default settings: min count=3, windows size=10, sample=0.001, algorithm=skip-gram.

The BP network for sentence classification was implemented with Tensorflow 1.10.0[11], a Python framework for deep learning framework. We set the following hyperparameters for the training, based on the default settings: one hidden layer with 256 neurons, learning rate=0.01, softmax function for output layer, batch size=512. We obtained 0.9 accuracy with these parameters on a test set with 10% of the labeled data.

3.2 Knowledge Fusion Implementation

The descriptions of the generic concepts used in generic concept filtering were extracted from the downloaded English Wikipedia dump[1]. We invited four students to label the data for generic concept filtering. Wikidata has a very large number of concepts (about 50 million) and most of them are irrelevant to software. So we selected a set of concepts that are likely related to software in the following way. We selected all the concepts that have "*subclass of*", "*instance of*", or "*part of*" relations (directly or indirectly) with "*programming language*", " *software*", or "*computer science*". Then we selected a subset of concepts whose names or Wikipedia descriptions include keywords such as, "*software*", "*library*", "*computer*". In this way, we selected 22,306 concepts and each of them was independently labeled as software related or not by two students. The consensus rate of the two people reached 99.8%. For the concepts that were labeled differently, a third student was assigned to give an additional label, for resolving the conflict. The label with two of three votes was selected as final. Then we obtained 21,809 software-related concepts and 497 irrelevant concepts. As a large part of the Wikidata concepts are irrelevant to software, we randomly selected another 32,216 concepts (which did not satisfy our previous criteria) and automatically labeled them as irrelevant, making the number of irrelevant concepts 1.5 times that of software-related concepts. Finally we obtained 54,522 labeled generic concepts (21,809 software-related and 32,713 irrelevant). The BP network for generic concept filtering was also implemented with Tensorflow 1.10.0. We used the same hyperparameters as for sentence classification.

We use NLTK to parse descriptive sentences for API concept reference extraction. The hierarchical clustering in cross concept reference extraction is implemented with SciPy 1.0.0[10]. The two weights in the similarity calculation (Equation 1) are equally set to 0.5 and the distance threshold of hierarchical clustering is set to 1.6. These parameters were chosen based on tuning with a small testing data set, selected randomly from the API concept reference extraction result, and labeled by the two of the authors.

The word vectors used for software concept fusion are trained in the same way as for the sentence classification. The only difference is that the corpus used here includes the descriptions of all the software concepts from Wikipedia. The threshold of similarity is set to 0.8 and the maximum number of software concepts for the fusion of an API concept is set to 5. These parameters were chosen based on tuning with a small testing data set, consisting of randomly selected API concept or API entities, and labeled software concepts that should be fused from them (labeled by two of the authors).

3.3 Resulting API Knowledge Graph

The resulting API KG includes 562,578 entities and 4,243,842 relations. Among them, there are 137,113 API entities and 305,826 relations between API entities. The KG includes 292,684 descriptive sentences: 130,641 for **Functionality** and 162,043 for **Directive**. Within these sentences, 54,596 API concepts are identified, which are linked to 278,994 sentences (5.11 on average). The knowledge graph also includes 78,182 software concepts, with 20,640 software concepts linked to 49,256 API concepts and 90,209 API entities.

3.4 Summarization Generation

We use node2vec[8] to train the graph vectors. Because we consider the relationships to be similar, we set the weights of all relationships to 1.0. The hyperparameters used for the training followed the default settings that node2vec implementation provides.

The two weights w_1 and w_2 used in the calculation of the combined similarity (see Equation 4) were set to 0.6 and 0.4, respectively, based on tuning with a testing data set, using queries from randomly selected SO questions with high quality answers and queries proposed by the authors, based on their experience.

We use M=3 and S=2 summary generation, i.e., one class summary contains at most three methods and at most two sentences for the class itself and for each each methods.

4 EVALUATION

We conducted several empirical studies to evaluate the intrinsic quality and usefulness of the API summaries. We are interested in answering the following research questions about the summaries generated by *KG-APISumm* .

> **RQ$_1$** *What is the intrinsic quality of the summaries generated by KG-APISumm ?*
>
> **RQ$_2$** *How are the summaries generated by KG-APISumm different than those generated by other approaches?*
>
> **RQ$_3$** *How useful are the summaries generated by KG-APISumm in helping developers during API retrieval?*
>
> **RQ$_4$** *Is KG-APISumm generating different summaries for different queries, for the same class?*

The answer to RQ$_1$ will inform us whether the generated summaries include relevant and understandable information. RQ$_2$ will reveal whether the *KG-APISumm* generated summaries contain relevant information, which is not included in other type of class summaries, or vice versa. RQ$_3$ provides extrinsic evaluation and will indicate whether the summaries are useful in addressing specific developer tasks. Finally, RQ$_4$ will tell us whether *KG-APISumm* actually produces task-specific summaries, that is, the same class will have different summary for different queries.

In order to answers the research questions, we collected a set of queries, corresponding to programming tasks. We asked external evaluators to assess the quality of the summaries generated by *KG-APISumm* for these tasks (RQ$_1$) and compared them with summaries generated by three baseline approaches (RQ$_2$). Another group of users analyzed the classes and methods retrieved for another set of queries, with and without the help of the generated summaries (RQ$_3$). Finally, we answer RQ$_4$ by providing examples of classes that have distinct summaries for different queries. Details of the empirical studies can be found in our replication package [9].

Table 1: Queries for the intrinsic evaluation

Post ID	Title	C	M
1109022	Close/hide the Android Soft Keyboard	3	9
153724	How to round a number to n decimal places in Java	3	10
2115758	How do I display an alert dialog on Android?	2	7
2885173	How do I create a file and write to it in Java?	6	22
3028306	Download a file with Android, and showing the progress in a ProgressDialog	3	14
3035692	How to convert a Drawable to a Bitmap?	1	5
3481828	How to split a string in Java	2	7
415953	How can I generate an MD5 hash?	1	6
432037	How do I center text horizontally and vertically in a TextView	1	2
46898	How do I efficiently iterate over each entry in a Java Map	2	4
5369682	Get current time and date on Android	3	6
858980	File to byte[] in Java	4	11
Sum		31	103

4.1 Intrinsic Quality of the Summaries

We perform an empirical intrinsic evaluation for assessing the API summaries generated by *KG-APISumm*, following the evaluation proposed in previous research [32, 45].

4.1.1 Tasks and Queries. We selected programming tasks, which appear in StackOverflow (SO) questions, based on the following criteria: (1) the question has a tag "Java" or "Android"; (2) the programming task involves using APIs that are included in JDK 1.8 or Android API 27; (3) the question has an accepted answer. We ranked the SO questions that match the above criteria by their scores and selected the 20 top-ranked questions. Form these, we randomly selected six questions for Java and six for Android. The SO questions are included in Table 1. Column C shows the number of relevant classes for this task the column M the number of relevant methods.

We determined the correct answers (classes and methods) for each query, by consulting the SO answers. Two of the co-authors and two external persons are involved, with one of the co-authors acting as expert. Each question was analyzed by two persons (at least one external). Each person independently annotated the class(es) and method(s) that he thinks are relevant. If a class/method is marked as relevant by both persons, then the class/method is considered to be the correct. Otherwise a the expert will make the final judgment, selecting one or both options.

4.1.2 Participants. We invited 12 master students who are experienced in Android and Java development to evaluate the quality of the API summaries. Their programming expertise was assessed through a survey administered to 50 graduate students. The 12 with the most experienced were selected. It is important for the summary evaluators to know and understand how to solve each task. Six of the 12 students evaluated the summaries for RQ_1, while the other six (decided randomly) evaluated the summaries for RQ_2.

4.1.3 Protocol. The students completed the evaluation in one session, in the lab, under the supervision of one of the authors, who also conducted the post interviews. For each task, they read the SO post solution for the task, to ensure they know how the task can be solved. Then they evaluated each summary for completeness, conciseness, and understandability, as in previous research on software summarization [32, 45]. For each class summary, the students answered three questions on a 4-points Likert scale (1-Disagree; 2-Somewhat disagree; 3-Somewhat agree; 4-Agree):

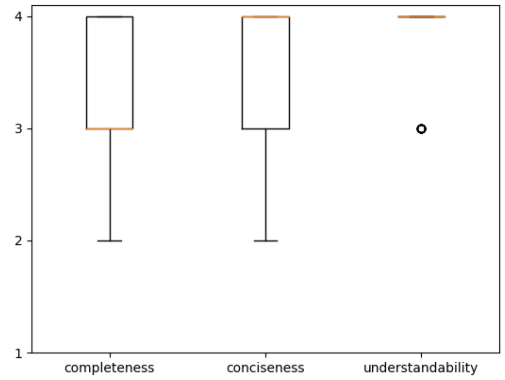

Figure 3: Answers to the three RQ_1 questions. 1-disagree, 2-somewhat disagree, 3-somewhat agree, 4-agree

(1) *Completeness*–does the summary contain all the necessary information?
(2) *Conciseness*–does the summary contain no (or very little) unnecessary or redundant information?
(3) *Understandability*–is the summary understandable?

We phrase the second question negatively to maintain the interpretation of the answers similar to all three questions. After a participant finished the evaluation, we asked for explanations, in case of low ratings (1 or 2).

4.1.4 Results and Analysis. Each question was answered by three students, for each summary. We cumulate the answers for each question, across all summaries, and we focus on the percentage of "positive" (3 or 4) and negative" answers (1 or 2).

Figure 3 shows the distribution of the answers for each question. For *completeness* **45.2**% of the answers are 4 (agree), **45.2**% are 3 (somewhat agree), **9.6**% are 2 (somewhat disagree), and there are no 1 (disagree) answers. For *conciseness* **53.8**% of the answers are 4, **36.6**% are 3, **9.6**% are 2, and there are no 1 answers. For *understandability* **80.6**% of the answers are 4 , **19.4**% are 3, and there are no 2 and 1 answers.

We used the one sample T-test [14] for verifying the statistical significance of the difference between the participants' ratings and random ratings. The null hypothesis is that the ratings for *completeness*, *conciseness*, and *understandability* are random and the mean of the ratings for each property is 2.5. The results show that for each property the statistical difference is significant ($p << 0.01$), so we reject the null hypothesis.

Seven summaries received at least one 2 rating (somewhat disagree) for conciseness. The participants explained in the post interview that they think some methods in summary are irrelevant and considered that as unnecessary information. However, for all of the seven summaries, there was at least one 3 (somewhat agree) or 4 (agree) rating for conciseness. For example, the summary for class `java.math.BigDecimal` was rated for conciseness with 2, 4, 4. The class is relevant for the "*How to round a number to n decimal places in Java*" question and it includes the following methods in the summary:

`java.math.BigDecimal.movePointRight(int)`,
`java.math.BigDecimal.movePointLeft(int)`, and
`java.math.BigDecimal.scale()`.

One rater considered the `scale()` method as the sole correct method for the task and the other two are unnecessary, hences gave a 2 rating. The other two raters considered `movePointRight(int)` are also useful for this task.

Six summaries received 2 (somewhat disagree) ratings for completeness. Rater reported that these ratings were given by those who considered other methods, not included in the summary, to be relevant to the task. Once again, these classes also received at least one 3 (somewhat agree) or 4 (agree) rating, as the other raters considered that the three listed methods are the most relevant, even when they found additional relevant methods. For example, the summary for the `java.util.Map.Entry` class was rated with 2, 4, 4 for completeness. The class is relevant for the query"*How do I efficiently iterate over each entry in a Java Map?*". One rater considered that the summary misses useful methods for the task. The other two raters considered the two class sentences useful enough to explain that this class can be used for iterate over the Map, hence their 4 rating. The iteration is done by methods from `java.util.Map`, while the methods from `java.util.Map.Entry` are just use to wrap the data. In this case, the lower rating seems more appropriate.

We also look at the agreement rate for each question. For conciseness, **29**% of summaries received the same rating by all three raters and **97**% the same rating by at least two raters. For completeness, **29**% of summaries received the same rating by all three raters and **100**% the same rating by at least two raters. For understandability, **55**% of summaries received the same rating by all three raters and **100**% the same rating by at least two raters.

4.1.5 Threats to Validity. The main threat to the *construct validity* is the subjectivity introduced in determining the correct answers for the 12 queries. To minimize bias, we selected tasks related to SO posts with valid answers, which we used.

KG-APISumm 's calibration impacts the *internal validity* of our conclusions. We used different data from the one used in the evaluation to find the best parameters of our approach. Another threat is subjectivity and error-proneness of the human-based evaluation. To mitigate this threat, we relied on three evaluators per summary and reported agreement data.

The number of tasks used in the evaluation impacts the *external validity* of our conclusions. A larger evaluation would be desirable.

4.2 Comparison with Baseline Approaches

For answering RQ_2, we use the same tasks and summaries used for answering RQ_1 (see Section 4.1.1). We asked six of the 12 participants selected as described in Section 4.1.2 to compare the summaries produced by *KG-APISumm* with the summaries produced by three baseline approaches.

4.2.1 Baseline Approaches. We used three baseline approaches, one implemented by the authors, and two using existing tools.

The first baseline is based on TextRank [29], a graph-based ranking model for automatic text summarization. We use TextRank to generate the summary for classes, using all sentences from the class. TextRank extracts up to five sentences as the class summary. For implementation we use gensim[4].

Table 2: Comparison with Baseline Approaches

Approach	APIKG-Summ	Biker	TextRank	Google	Average
APIKG-Summary	X	2.75	3.29	3.34	3.13
Biker	2.67	X	2.73	2.91	2.77
TextRank	2.30	1.96	X	2.39	2.22
Google	1.84	1.82	2.21	X	1.96

The second baseline approach is based on Google. In order to generate a class summary, we search in Google with the SO title, limiting the results to jdk reference documentation website or android sdk documentation website. We take the digests of the relevant search results (class and methods) as the API summaries.

The third baseline approach is Biker [22], an API recommendation approach that provides summaries for the recommended APIs by combining information from SO and from the API documentation. We use the SO question title as query in Biker, limit the search to class-level, and get the summary from the search results.

4.2.2 Protocol. We adopt a protocol similar to the one used in previous research, when comparing release notes produced in two different ways [34, 35]. We assigned each query/task to two participants, each participant rated the summaries for four tasks. For each query/task, participants were shown the same relevant classes, but with different summaries, each generated by a different approach. The raters only compared two summaries at a time. Given summary S_1 and S_2, for class C and query Q, each produced by a different approach, the raters were asked to consider each elements of the summary (i.e., method/class sentence and method signature, SO question, or code fragment). For each summary element they were asked to mark whether the element is: (1) present only in S_1; (2) present only in S_2 only in B; or (3) present in both in both S_1 and S_2. Then for those that are only in S_1 or only in S_2, and rate on a 4-Likert scale whether the unique elements are useful. The rating correspond to: 1-Strongly disagree; 2-Disagree; 3-Agree; and 4-Strongly agree. We asked the raters to provide an overall evaluation of these unique items, rather than assessing each item individually, which would have taxed the participants too much, in terms of time and effort.

Because Biker only works for JDK, we only compare *KG-APISumm* TextRank and Google using the Android queries, whereas all three for the JDK queries. If an approach cannot generate a summary for a class, then we ignore this approach when comparing different summaries of this class. For example, Google did not retrieve some relevant classes with the specified query).

4.2.3 Results and Analysis. Table 2 show the average ratings for each pair. The average rating in cell (i,j) indicates the rating of the information present in the summaries produced by the approach from row i and not in those produced by the approach from column j. For example, the information present the *KG-APISumm* summaries (row 2) and not in TextRank summaries (column 4) were rated in average with 3.29. Conversely, the information present the TextRank summaries (row 4) and not in *KG-APISumm* summaries (column 2) were rated in average with 2.30.

In **72**% of the cases the two ratings for a pair of summaries the same. In **48**% of the cases the two ratings for a pair of summaries are 3 and 4. In **51**% of the cases the two ratings for a pair of summaries are 1 and 2. In **15**% of the cases the two ratings disagree significantly, one is 1 or 2 and the other is 3 or 4, indicating low disagreement.

4.2.4 Threats to Validity. We share the same *construct validity* threats with the previous study, as we used the same data.

The *internal validity* of our conclusions is impacted by the fact that the baseline approaches generate different type of summaries and we had to compose the summaries using the data produced by those tools, to be comparable with ours. Another threat is subjectivity and error-proneness of the human-based evaluation. To mitigate this threat, we relied on two evaluators per summary pair and reported agreement data.

The *conclusions validity* is impacted by the fact that we used different data sets for different pairs of summaries.

The number of tasks used in the evaluation impacts the *external validity* of our conclusions. A larger evaluation would be desirable.

4.3 Usefulness of the Summaries

We perform an extrinsic evaluation for addressing RQ_3. A group of students were asked to find the relevant classes to a query, returned by a tool named Biker [22]. Students performed these tasks with and without having *KG-APISumm* generated summaries. The goal is to assess whether the use of the *KG-APISumm* summaries help in performing the given task.

4.3.1 Tasks and Queries. We selected 20 queries (for API retrieval) from the data set used to evaluate Biker [22] and use its ground truth information. We divided the queries into 10 types and we choose randomly two queries for each type query. The query classification combines the number of ground truth classes corresponding to the query (one or multiple) and the ground truth classes ranking in the Biker search results. For a query that has one ground truth class, there are five types of rankings: ground truth class in top 1, top 2-3, top 4-5, top 6-10, out of top 10. For a query that has multiple ground truth classes, there are five types of rankings: all ground truth classes are in top 3, all in top 5, all in top 10, at least one in top 10 and at least one out of top 10, all out of top 10. The reason we created these categories is that we wanted to maximize the variety of retrieval tasks and results (i.e., one or more relevant classes, with various ranks). We divide the queries into two groups, QA and QB. Each group contains 10 queries, one form each query category.

4.3.2 Participants. We invited 12 master students with Java programming experience, yet with different experience levels (beginners, intermediate, advanced). Their programming expertise was assessed through a survey distributed to 50 graduate students. None of these students participated in the previous studies. We divided the participants in two groups of six (PA and PB), each group with two beginners, two intermediates, and two advanced.

4.3.3 Protocol. The studied factor is the use of summaries (i.e., the independent variable). We adopted a balanced treatment distribution for the groups. Participants in group PA were asked to complete the tasks in group QA, using the *KG-APISumm* summaries and the tasks in group QB, without the use of summaries. Conversely, participants in group PB were asked to complete the tasks in group QB, using the *KG-APISumm* summaries and the tasks in group QA, without the use of summaries. Overall, each participant was asked to complete all 20 tasks, 10 with the use of *KG-APISumm* summaries and 10 without. The tasks were interleaved, for each participant, one performed with *KG-APISumm* summaries and one without.

For a given task, the participant is given the corresponding query and the top-10 classes retrieved by Biker (with the full javadoc). In the first treatment, the list of results also includes the *KG-APISumm* summaries for each class. Note that in some cases the relevant classes are not in top-10. The participants were asked to select the class(es) they think can solve the problem described in the query. The participants were allowed to use the internet, if they deemed the provided information as insufficient, yet we asked them to ignore the SO post where the task query is from.

We developed a website each person used to complete the task, one by one, and to submit their answer. The system automatically records the completion time. We recorded the time each task is completed and the correctness of each answer. If the classes submitted by a participant contained at least one ground truth class, we consider the answer correct, and the correctness score is 1. Otherwise the correctness score is 0 for this query.

4.3.4 Result and Analysis. Table 4 reports the average completion time and correctness for each treatment, which indicate that when using the *KG-APISumm* summaries, participants (in both groups) complete the tasks 17.9% faster (37 seconds) and their answers are more correct (in average).

We verified the distribution of the data (i.e., normal) using the Kolmogorov-Smirnov test [27] ($p < 0.05$) and used the Welch's T-test [49] for verifying the statistical significance of the differences between treatments. The differences in time and correctness are both statistically significant ($p < 0.05$).

For most of queries, the use of summary leads to faster responses, but not in all cases. In some cases the participants insisted on additional web searches, leading to longer response time. In some cases, participants submitted an empty answer after a short time, essentially indicating that they cannot solve the task. Participants using the summaries tended to spend more time reading the summaries and than searching for additional results.

We analyzed the effect of the summaries on beginners, intermediate, and advanced programmers from the subject groups (see Table 5). The beginners spent more time in average per task (in both treatments), than the intermediates and advanced participants, who spent similar times. In addition, the average time improvement seen by the beginners is larger than that experienced by intermediates and advanced participants, suggesting that the summaries are more useful for inexperienced programmers. We also observed that when task is more complex (i.e., with more than one class in the oracle), the summary saves more time than when the task is easier. For the single class tasks, the summaries are most helpful (221s->172s) for the single-top-10 cases. For the multi class tasks, the summaries are most helpful (232s->160s) for the multi-top-10 cases.

4.3.5 Threats to Validity. We share the simlar *construct validity* threats with the previous studies, but we mitigated this issue by selecting tasks that have an external oracle, define in other research.

Another threat is subjectivity and error-proneness of the human-performed tasks. To mitigate this threat we designed a balanced treatment, to account fo variability between subjects and tasks.

The *conclusions validity* is impacted by the comparisons of averages, which we mitigate by reporting statistical significance. The number of tasks and users used in the evaluation impacts the *external validity* of our conclusions. A larger evaluation is needed.

Table 3: Same Class Summmary For different Query

ID	Query	Method in Summary	Biker Ground Truth Method
Q1	Elegant way to read file into byte[] array in Java	readAllBytes, write, copy	readAllBytes
Q2	Getting A File's Mime Type In Java	probeContentType, createTempFile, createTempFile	probeContentType
Q3	Getting file creator/owner attributes in Java	getOwner, setOwner, getOwner	getOwner
Q4	How to append text to file in Java 8 using specified Charset	newBufferedWriter, write, lines	write
Q5	How can I write to a specific line number in a txt file in Java	write, readAllLines, write	readAllLines, write
Q6	Add custom attribute or metadata to file java	getFileAttributeView, readAttributes, getOwner	setAttribute
Q7	Add all files recursively from root with Java 8 Stream	walk, walk, walk	walk

Table 4: Avg. Correctness and Time per Treatment

	Task Group QA	Task Group QB	Average
With Summary	0.87/159s	0.95/185s	0.91/172s
Without Summary	0.82/212s	0.78/207s	0.80/209s

Table 5: Average Correctness and Completion Time Analysis

	Beginner	Interm.	Advanced	Single Cl.	Multi Cl.
With Summary	0.85/179s	0.92/169s	0.95/167s	0.95/169s	0.87/175s
Without Summary	0.87/227s	0.67/202s	0.85/199s	0.85/190s	0.75/228s
Improvement	-0.02/48s	0.25/33s	0.1/32s	0.10/21s	0.12/53s

4.4 Variability in the API Summaries

We want to verify that *KG-APISumm* generates distinct summaries for different queries. We look examples where that is the case.

We select seven queries from Biker data set that contain the same class in the ground truth, namely `java.nio.file.Files`[5]. The class provides methods that operate on files and directories.

The seven queries are shown in Table 3. Column 3 indicates the methods included in the summaries. Same method name in the column indicates overriding. Column 4 shows ground truth method from the Biker data set.

The class sentences are the same for each query (not displayed here), but the methods vary from query to query, implicitly the describing sentences too (not included here). However, six of the seven queries the summaries include the Biker's ground truth method and one (Q6) does not. Nontheless, the methods included in the summary are somewhat related. Overall, *KG-APISumm* generates different query-related summaries for the same class.

5 RELATED WORK

We survey related research that focuses on source code summarization at class level. Recent surveys on automatic software summarization [31, 36] cover approaches for summarizing other type of software artifacts, or smaller code elements than classes (e.g., code blocks or methods). Most related is the work Petrosyan et al. [39], who augmented API type documentation with usage explanations extracted from development tutorials, while Treude and Robillard [48] did it with insights from SO. In both cases, the injected information was detected with a supervised text classifier.

Other related work focused on discovering relevant tutorial fragments [23] and linking source code examples to API documentation [33, 46]. In common with our approach, these approaches link APIs with relevant text or code fragments in various sources. However, unlike our work they do not construct a graph of API-related knowledge and do not generate query-dependent summaries.

In other work, Moreno et al. [32] investigated the generation of natural language comments for Java classes, by leveraging code stereotypes (i.e., abstractions that represent the role or responsibility of code elements), identified through static analysis. A similar approach was used by Panichella et al. [38] to document test cases. Haiduc et al. [19, 20] proposed the use of text retrieval techniques (in particular, vector space model and latent semantic indexing) to select descriptive terms for methods and classes. Similar approaches were followed by De Lucia et al. [16, 17] to generate class keywords. These type of summaries rely strictly on information extracted from the code to be summarized and are task-agnostic.

Some work on summarizing code at other granularity is also relevant, in as much as it extracts information from other sources than the code to be summarized, based on relationships with it (structural or textual). For example, McBurney and McMillan [28] generate summaries for Java methods, including information about the code's context (e.g., called methods). Moreno et al. [34, 35] extracts information from issue trackers for summarizing code changes, as release notes. Panichella et al. [37] extract method descriptions from related developer communications. More recently Huang et al. [22], proposed Biker, a tool which recommends relevant APIs to a query. At the same time it generates template-based summaries, including information from API descriptions and relevant code examples in SO posts. The Biker summaries are used in this paper for comparison purposes. Unlike our work presented here, the summaries produced by all these approaches are not specific to a programming task or a user query.

6 CONCLUSIONS

We proposed an approach (*KG-APISumm*) for generating query-based class API summaries, using an API knowledge graph. The queries are natural language formulations of programming tasks. In many cases, different queries result in different summaries for the same classes. A set of external subjects deemed the summaries to be complete, concise, understandable, and including unique information that is more useful than that included in summaries produced by other three summarization tools. More importantly, subjects solving 20 different API retrieval tasks, with the help of the *KG-APISumm* summaries, achieved better and faster answers to the retrieval tasks, in average, than in the absence of summaries. More than that, novice programmers benefit most from the use of such summaries in the retrieval tasks.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004803, and in part by the US NSF grants CCF-1526118 and CCF-1848608.

REFERENCES

[1] 2018. *Wikipedia Dump.* Retrieved July 20, 2018 from https://dumps.wikimedia.org/enwiki

[2] 2019. *Android 27.* Retrieved June 18, 2019 from https://developer.android.com/reference/packages

[3] 2019. *Beautiful Soup.* Retrieved June 18, 2019 from https://www.crummy.com/software/BeautifulSoup/

[4] 2019. *gensim.* Retrieved June 18, 2019 from https://radimrehurek.com/gensim/

[5] 2019. *java.nio.file.Files Document.* Retrieved June 18, 2019 from https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

[6] 2019. *JDK 1.8.* Retrieved June 18, 2019 from https://docs.oracle.com/javase/8/docs/api/

[7] 2019. *nltk.* Retrieved June 18, 2019 from https://www.nltk.org/

[8] 2019. *node2vec.* Retrieved June 18, 2019 from https://github.com/aditya-grover/node2vec

[9] 2019. Replication Package. Retrieved June 18, 2019 from https://fudanselab.github.io/Research-ESEC-FSE2019-APIKGSummary/

[10] 2019. *scipy.* Retrieved June 18, 2019 from https://www.scipy.org

[11] 2019. *tensorflow.* Retrieved June 18, 2019 from https://www.tensorflow.org

[12] 2019. *wikipedia.* Retrieved June 18, 2019 from https://wikipedia.org

[13] David Binkley, Dawn Lawrie, Emily Hill, Janet Burge, Ian Harris, Regina Hebig, Keszocze, Karl Reed, and John Slankas. 2013. Task-Driven Software Summarization. In *29th IEEE International Conference on Sofware Maintenance (ICSM'13), ERA track*. Eindhoven, The Netherlands, 432–435.

[14] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146.

[15] Barthélémy Dagenais and Martin P. Robillard. 2012. Recovering traceability links between an API and its learning resources. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 47–57.

[16] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2012. Using IR Methods for Labeling Source Code Artifacts: Is It Worthwhile?. In *IEEE 30th International Conference on Program Comprehension (ICPC'12)*. Passau, Germany, 193–202.

[17] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2014. Labeling Source Code with Information Retrieval Methods - An Empirical Study. *Empirical Software Engineering* (2014).

[18] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[19] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting Program Comprehension with Source Code Summarization. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), NIER track*. Cape Town, South Africa, 223–226.

[20] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *17th IEEE Working Conference on Reverse Engineering (WCRE'10)*. Beverly, MA, 35–44.

[21] Robert Hecht-Nielsen. 1988. Theory of the backpropagation neural network. *Neural Networks* 1, Supplement-1 (1988), 445–448.

[22] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 293–304.

[23] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 38–48.

[24] Rafael Lotufo, Zeeshan Malik, and Krzystof Czarnecki. 2012. Modelling the 'Hurried' Bug Report Reading Process to Summarize Bug Reports. In *28th IEEE International Conference on Software Maintenance (ICSM'12)*. Trento, Italy, 430–439.

[25] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. 2014. Modelling the 'hurried' bug report reading process to summarize bug reports. *Empirical Software Engineering* 20, 2 (2014), 516–548.

[26] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282.

[27] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[28] Paul W. McBurney and Collin McMillan. 2014. Automatic Documentation Generation via Source Code Summarization of Method Context. In *Proceedings of the 22Nd International Conference on Program Comprehension*. ACM, New York, NY, USA, 279–290.

[29] Rada Mihalcea and Paul Tarau. 2004. TextRank: Bringing Order into Text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing , EMNLP 2004, A meeting of SIGDAT, a Special Interest Group of the ACL, held in conjunction with ACL 2004, 25-26 July 2004, Barcelona, Spain*. 404–411.

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.

[31] Laura Moreno. 2016. *Software documentation through automatic summarization of source code artifacts.* The University of Texas at Dallas.

[32] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and Vijay Shanker. 2013. Automatic Generation of Natural Language Summaries for Java Classes. In *21st IEEE International Conference on Program Comprehension (ICPC'13)*. IEEE, San Francisco, USA, 23–32.

[33] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE, 880–890.

[34] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic Generation of Release Notes. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, USA, 484–495.

[35] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2017. ARENA - An Approach for the Automated Generation of Release Notes. *IEEE Trans. Software Eng.* 43, 2 (2017), 106–127.

[36] N. Nazar, Y. Hu, and H. Jiang. 2016. Summarizing Software Artifacts: A Literature Review. *Journal of Computer Science and Technology* (2016).

[37] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, and Andrian Marcus. 2012. Mining Source Code Descriptions from Developer Communications. In *20th IEEE International Conference on Program Comprehension (ICPC'12)*. Passau, Germany, 63–72.

[38] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In *Proceedings of the 38th International Conference on Software Engineering*. 547–558.

[39] Gayane Petrosyan, Martin P. Robillard, and Renato De Mori. 2015. Discovering Information Explaining API Types Using Text Classification. In *Proceedings of the International Conference on Software Engineering*. 869–879.

[40] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2010. Summarizing Software Artifacts: A Case Study of Bug Reports. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. Cape Town, South Africa, 505–514.

[41] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2014. Automatic Summarization of Bug Reports. *Software Engineering, IEEE Transactions on* 40, 4 (2014), 366–380.

[42] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[43] Lior Rokach and Oded Maimon. 2005. Clustering Methods. In *The Data Mining and Knowledge Discovery Handbook*. 321–352.

[44] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Software Eng.* 34, 4 (2008), 434–451.

[45] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. Antwerp, Belgium, 43–52.

[46] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 643–652.

[47] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining.* Addison-Wesley.

[48] Christoph Treude and Martin P. Robillard. 2016. Augmenting API documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 392–403.

[49] Bernard L Welch. 1947. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.