Recommending Analogical APIs via Knowledge Graph Embedding

Mingwei Liu[*]
Fudan University
China

Yanjun Yang[*]
Fudan University
China

Yiling Lou[*†]
Fudan University
China

Xin Peng[*]
Fudan University
China

Zhong Zhou[*]
Fudan University
China

Xueying Du[*]
Fudan University
China

Tianyong Yang[*]
Fudan University
China

ABSTRACT

Library migration, which replaces the current library with a different one to retain the same software behavior, is common in software evolution. An essential part of this is finding an analogous API for the desired functionality. However, due to the multitude of libraries/APIs, manually finding such an API is time-consuming and error-prone. Researchers created automated analogical API recommendation techniques, notably documentation-based methods. Despite potential, these methods have limitations, e.g., incomplete semantic understanding in documentation and scalability issues.

In this study, we present KGE4AR, a novel documentation-based approach using knowledge graph (KG) embedding for recommending analogical APIs during library migration. KGE4AR introduces a unified API KG to comprehensively represent documentation knowledge, capturing high-level semantics. It further embeds this unified API KG into vectors for efficient, scalable similarity calculation. We assess KGE4AR with 35,773 Java libraries in two scenarios, with and without target libraries. KGE4AR notably outperforms state-of-the-art techniques (e.g., 47.1%-143.0% and 11.7%-80.6% MRR improvements), showcasing scalability with growing library counts.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; **Software evolution**; **Software libraries and repositories**.

KEYWORDS

API Migration, Knowledge Graph, Knowledge Graph Embedding

[*]M. Liu, Y. Yang, Y. Lou, X. Peng, Z. Zhou, X. Du, and T. Yang are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.

[†]Y. Lou is the corresponding author (email: yilinglou@fudan.edu.cn).

1 INTRODUCTION

Third-party libraries are pivotal in modern software development, enhancing quality and productivity [37, 43, 46, 57]. However, as software and libraries evolve rapidly, current libraries might turn unsuitable due to factors like sustainability failures [35, 67], license restrictions [40, 68], lacking features [42], and security/performance issues [42]. This necessitates *library migration*, where developers replace current libraries with new ones to re-implement the same software behavior. Such migrations are common in software evolution [41]; for example, He *et al.* [41] found 8.98% ~ 28.72% of 17,426 open-source projects underwent at least one library migration.

However, library migration [27, 30, 31, 37, 43] is a very time-consuming, labor-intensive, and error-prone task for developers in practice. For example, the prior study shows that some developers even spend up to 42 days for library migration [30]. Given the currently-used library (called **source library**) and the API (called **source API**), one essential part in library migration is to find an analogical library (called **target library**) and an analogical API (called **target API**), which can provide the same functionality as current ones. However, manually finding analogical API is a heavy burden for developers, since they need to read length API documentation and code snippets of potentially-analogical APIs [27–30] while there are an extremely large number and fast changes of third-party libraries and APIs (e.g., as of January 2020 there are 35,773 common Java libraries with 15,441,057 APIs on Libraries.io [12]).

To reduce efforts in manually searching and reading API documentation and code snippets for determining the analogical relationship between libraries and APIs, many techniques have been proposed to recommend suitable target libraries or analogical target APIs. In this work, we focus on analogical API recommendation. Researchers have harnessed diverse resources to facilitate such recommendations [27–30], including evolution history [58, 65], online Q&A interactions [34], and API documentation [28, 30, 34, 54, 59,

60, 79]. Among these, documentation-based API recommendation has been intensively studied in the literature, since API documentation is prevalent and at low cost to collect while other information could be time-consuming to collect and are not always available. For a given source API, existing documentation-based techniques calculate the textual similarity between each candidate API and the source API (*e.g.,* the textual similarity between two API functionality descriptions in the documentation), and then recommend the candidate API with the highest similarity as the target API.

While promising, current documentation-based API recommendation techniques face two limitations. First, their way of calculating textual similarity falls short in capturing semantic-level connections in API documentation. These techniques mainly calculate the textual similarity based on the overlapping tokens [59] or measure the token similarity without contextual consideration [79]. This can lead to identifying analogical semantics in API descriptions that share similar noun phrases but differing action verbs (*e.g.,* "set S3 Object content" vs. "get S3 Object content length"). Additionally, these techniques seldom consider domain knowledge when calculating textual similarity. For example, JSON arrays, JSON objects, keys, and values are all JSON-related concepts that often occur in APIs related to JSON processing. Concepts, in the context of our work, refer to domain-specific entities or terms, often represented by noun phrases, that capture specific elements or ideas within the API domain. Without considering such conceptual relationships, the estimation of semantic similarity/relevance between two analogical APIs might be underestimated. Second, these techniques typically compute similarity pairwise, posing computational challenges with a vast number of candidate APIs. For example, envision a library like TestNG [23], encompassing over 4,000 candidate APIs. Existing techniques require performing over 4,000 pairwise comparisons to calculate the similarity between a single source API and all the candidate APIs. This exhaustive calculation demands substantial online costs and becomes prohibitively expensive when multiple target libraries are involved.

To address this, we propose **KGE4AR**, a novel documentation-based method leveraging **K**nowledge **G**raph **E**mbedding **for** analogical **A**PI **R**ecommendation *effectively and scalably.* KGE4AR constructs a unified API knowledge graph (KG) for third-party libraries from API documentation, leveraging graph embedding to represent nodes and edges as numeric vectors. It efficiently retrieves the most similar API for a given source API from the embedded KG. Compared to previous approaches, KGE4AR introduces two technical innovations. Firstly, it presents a novel *unified API KG* that comprehensively represents three types of documentation knowledge across diverse libraries, better capturing overall semantics in API documentation. Secondly, KGE4AR proposes *embedding the unified API KG*, enhancing efficiency and scalability by streamlining analogous API vector retrieval via vector indexing.

To implement KGE4AR, we build a unified API KG consisting of 59,155,631 API elements sourced from 35,773 Java libraries. This KG comprises a total of 72,242,099 entities and 289,122,265 relations connecting these entities. We evaluate KGE4AR in two API recommendation scenarios: with and without target libraries. When given the target libraries, KGE4AR achieves 47.1%-143.0% and 41.4%-95.4% improvements over the baselines in terms of MRR and Hit@10,

respectively; while without a given target library, KGE4AR substantially outperforms existing analogical API recommendation techniques by achieving 11.7%-80.6%, 26.2%-72.0%, and 33.2%-116.5% improvements in terms of MRR, precision, and recall, respectively. We also evaluate the scalability of KGE4AR and find that it scales well with an increasing number of libraries. Furthermore, we extensively investigate the impact of different design choices in KGE4AR.

In summary, this work makes the following contributions:

- **Novel Approach:** We introduce KGE4AR, a documentation-based analogical API recommendation method that builds a unified API KG for numerous libraries, offering scalable recommendations via KG embedding.

- **Thorough Evaluation:** We thoroughly evaluate KGE4AR through effectiveness comparisons in two API recommendation scenarios, scalability assessment across various library quantities, and analysis of design choice implications.

- **Public Benchmark**: We release a benchmark for extensive analogical API evaluations across numerous libraries.

2 BACKGROUND AND RELATED WORK

In this section, we discuss related work in analogical API recommendation and knowledge graphs in software engineering.

2.1 Analogical API Recommendation

Existing analogical API recommendation techniques leverage various sources like evolution history [29, 58], online posts [49], and API documentation [28, 30, 34, 54, 59, 60, 79] to find suitable target APIs. Evolution-history-based methods [65] use evolution history (*e.g.,* code changes) to mine frequently co-occurring API pairs, while documentation-based ones [28, 30, 34, 54, 59, 60] calculate textual similarity using API-related text (*e.g.,* descriptions). We concentrate on documentation-based recommendation due to its prevalence, low cost of data collection, and recent research emphasis.

Existing documentation-based API analogical techniques mainly fall into two categories, *e.g.,* supervised learning based [28] and unsupervised learning based ones [30, 33, 34, 54, 59, 60, 79]. For supervised learning-based techniques, Alrubaye *et al.* [28] propose to train a machine learning model (*i.e.,* boosted decision tree) for analogical API inference based on the features extracted from API documentation (*e.g.,* the similarity of their method descriptions, return type descriptions, method names, and class names) and leverage the trained model to predict the probability of an unseen API pair being analogical. Different from supervised techniques that require a large amount of labeled data, unsupervised learning-based techniques often vectorize APIs in an unsupervised way and then recommend analogical APIs based on vector similarity. For example, Zhang *et al.* [79] leverage the Word2Vec model to vectorize the API functionality description, API parameters, and API return values, and then calculate a joint similarity based on these vectors.

Although achieving promising effectiveness, existing documentation-based techniques suffer from two major drawbacks. First, they calculate the textual similarity based on the overlapping tokens [59] or measure the token similarity without considering the whole context [79], thus cannot well capture the semantic-level similarity in API documentation. Second, they calculate the pair-wise similarity between all APIs in an exhaustive way, thus suffering from the
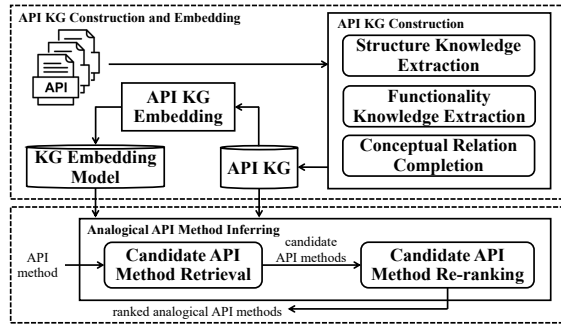
Figure 1: Overview of KGE4AR

scalability issue when the number of APIs is large. To address these issues, our work makes the first attempt to comprehensively and structurally represent the knowledge in API documentation with a novel *unified API KG*. In addition, we further leverage the KG embed to enable more effective and scalable similarity calculation. Our evaluation results also demonstrate our improvements over existing documentation-based techniques.

2.2 Knowledge Graph in Software Engineering

In the domain of software engineering, researchers have established knowledge graphs for diverse objectives, encompassing API concepts [50, 77], API caveats [45], API comparison [53], API documentation [51, 61], domain terminology [69–71], programming tasks [48], ML/DL models [52], and bugs [64, 73]. Our work applies the API knowledge graph in a task that is distinct from existing work, namely analogical API recommendation. In addition, since targeting different tasks, the design and focus of our API knowledge graph are also different from existing ones. For example, the existing API knowledge graph constructed for API misuse detection [62] mainly includes the call-order and condition-checking relations between APIs, while our API knowledge graph focuses on three types of knowledge (*i.e.,* API structures, API functionality descriptions, and API conceptual relationships) in API documentation which are helpful for analogical API recommendation. Moreover, we also propose a novel knowledge graph embedding to enable more effective and more scalable analogical API recommendation.

2.3 Knowledge Graph Embedding

Knowledge graph embedding (KGE) uses low-dimensional vectors to represent entities and relationships in a knowledge graph, capturing semantic relationships between entities [74]. KGE models map entities into a vector space, where similar ones are closer. They excel in applications like question answering, recommendations, and knowledge graph completion [38, 74]. Common KGE approaches are TransE, TransR, and DistMult [32, 47, 78]. These methods encode KG triples (head entity, relation, tail entity) into continuous vector representations. For instance, TransE treats entities and relations as vectors, defining relationships as translations from head to tail entities [32]. We employ KGE to embed a unified API KG for analogical API recommendation.

3 APPROACH

As shown in Figure 1, KGE4AR includes three phases, *i.e.,* API KG construction, API KG embedding, and analogical API method inferring. Given the API documentation from a large number of libraries

as inputs, KGE4AR first constructs a unified API KG (Section 3.1) and then trains an embedding model to embed the constructed KG (Section 3.2). Lastly, for a given source API, KGE4AR returns its analogical API based on the embedded KG (Section 3.3). Note that the first two phases only need to be run once. Once the unified KG is constructed and embedded, KGE4AR can recommend analogical APIs for the given API efficiently. In particular, KGE4AR mainly has two technical novelties.

Novelty 1: a unified API KG for a large number of libraries. We propose constructing a unified API KG for a substantial library count (*e.g.,* 35,773 Java libraries in this study). Our API KG comprises three knowledge types found in documentation, which often resemble analogical APIs: (1) *API structures* (e.g., package structures, class definitions, method declarations), (2) *API functionality descriptions* (e.g., *"get the number of elements in the JSONArray"*), and (3) *API conceptual relationships* (i.e., API concepts and their relationships like *"belong to"*). Unlike existing approaches that focus solely on API structures or functionality descriptions presented as token sequences, our unified API KG offers a broader, structural representation encompassing all three knowledge types. This includes a novel category—API conceptual relationships—previously unexplored. A graphical structure inherently suits the structured unification of multi-type data, thus effectively capturing the higher-level semantics within API documentation.

Novelty 2: a KG embedding-based similarity calculation. We propose embedding the unified API KG, representing each KG API as a vector. KG embedding offers two advantages. First, it effectively preserves structural and semantic data in the unified KG. Second, it expedites similarity calculations between APIs in the KG. Retrieving similar API vectors from a database via vector indexing is highly efficient. Unlike existing methods requiring exhaustive similarity calculations for all API pairs, our KG embedding enables a more efficient and effective approach to similarity calculation.

3.1 API Knowledge Graph Construction

In this phase, KGE4AR constructs a unified API KG for a large number of libraries based on their API documentation. The API KG construction mainly consists of three steps. (1) Structure knowledge extraction: KGE4AR first extracts all API elements (*e.g.,* packages, classes/interfaces, methods, fields, parameters) and their relationships from the documentation to form a basic skeleton of the API KG; (2) Functionality knowledge extraction: KGE4AR then extracts the functionality knowledge of the API libraries, *i.e.,* the standardized functionality expressions of the methods (including functionality verbs, functionality categories, and phrase patterns) and the involved concepts, from the names and text descriptions of methods; (3) Conceptual relation completion: KGE4AR completes conceptual relations between API elements and concepts by analyzing names and text descriptions of API elements and concepts. In this way, API elements from different libraries can be related to each other based on shared type references (*e.g.,* types of method parameters and return values), functionality expressions, and concepts.

3.1.1 Schema of the Unified API Knowledge Graph. Our API KG captures the structural and high-level information present in API documentation. It consists of entities (nodes) and relations (edges)
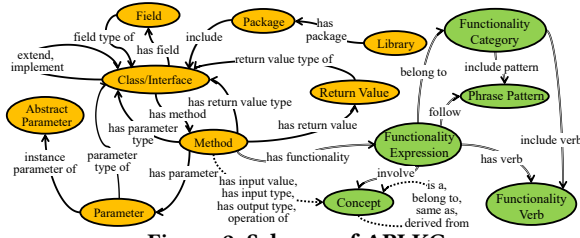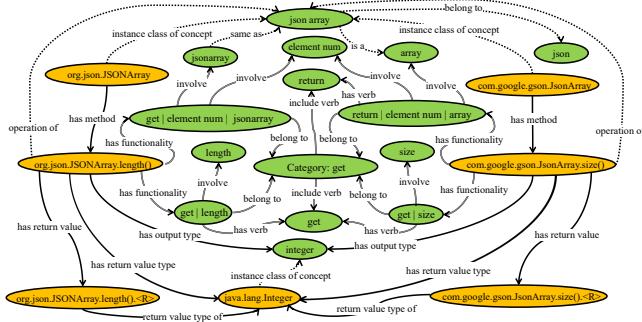
Figure 2: Schema of API KG

Figure 3: An Example of API KG

that represent various aspects of APIs. Here, we offer definitions for key entities and relations:

- **API Element**. *API elements* encompass components like libraries, packages, classes/interfaces, fields, methods, return values, parameters, and abstract parameters, forming the fundamental API building blocks.

- **Structural Relation**. *Structural relations* describe the relationships between API elements, including "extend" (inheritance), "implement" (interface implementation), "has field" (fields within classes/interfaces), "has method" (methods within classes/interfaces), and "has parameter" (methods with required parameters), forming the API KG's foundation.

- **Functionality Expression Element**. *Functionality expression elements* pertain to the structural representation of API functionality descriptions. This includes functionality expressions, functionality verbs, functionality categories, phrase patterns. They facilitate the standardized representation of API functionalities, as defined by Xie et al. [76].

- **Functionality Expression**. A *functionality expression* provides a structural representation for the functionality descriptions of methods following the standardized form defined by Xie *et al.* [76]. It is extracted from the description sentence of a method.

- **Functionality Verb**. A *functionality verb* represents the verb that express the main action of the functionality, *e.g.,* "return", "get", and "obtain".

- **Functionality Category**. A *functionality category* categorizes the functionality expressions based on their semantic meanings, which is abstracted from a set of functionality verbs that have similar meanings, *e.g.,* "return", "get", and "obtain" can be classified into the same category.

- **Phrase Pattern**. *Phrase patterns* capture specific syntactic patterns or templates used in functionality expressions, *e.g.,* "V {patient}" and "V {patient} in {location}". In the phrase pattern "V {patient} in {location}" the placeholders "patient" and "location"

represent noun phrases that fulfill semantic roles. "{Patient}" corresponds to the direct object of the functionality verb, signifying the entity or object directly affected by the action. "{location}" denotes the spatial or temporal context associated with the verb.

- **Concept**. Concepts in the API KG are specific semantic units that capture domain-specific knowledge or common themes in API documentation. These concepts are typically represented by noun phrases. For instance, in APIs related to JSON processing, concepts like JSON arrays, JSON objects, keys, and values frequently appear. Concepts may be involved in functionality expressions by playing some semantic roles (*e.g.,* patient, location).

Figure 2 showcases the schema of our API KG, illustrating the types of entities and relations involved. Furthermore, Figure 3 provides a partial API KG example, highlighting the interconnectedness of these entities and relations. The complete schema, including definitions for all the entity and relation types, is available in our replication package [20]. The orange ellipses and solid lines denote API elements and the structural relations between them, respectively. Among them, abstract parameters represent the abstraction of the parameters of different methods that share the same names and types. For example, all the method parameters with the name *path* and type *java.lang.String* are treated as the instances of the same abstract parameter. The green ellipses denote functionality expression elements (*i.e.,* functionality expressions, functionality verbs, functionality categories, and phrase patterns) and related concepts and double lines denote relations between these elements and concepts. Note that multiple methods may share the same functionality expression if their functionality descriptions include the same functionality verb, phrase pattern, functionality category, and concepts. The dashed lines denote various relations (*e.g.,* "is a" and "belong to") between concepts and the involvement relations between API elements and concepts (*e.g.,* "has input type" between methods and concepts). Some relations are omitted in Figure 2 for brevity, *e.g.,* the "instance class of concept" relation between classes and concepts (see Section 3.1.4).

In this way, API elements from different libraries can be indirectly related through structural relations (*e.g.,* shared parameter or return types), functionality expressions (*e.g.,* shared functionality categories and involved concepts), and concepts (*e.g.,* associated with related concepts).

Figure 3 shows some entities and relations related to the API methods *org.json.JSONArray.length()* and *com.google.gson.JsonArray-.size()*. The two methods are analogical API methods from two libraries *org.json* [15] and *gson* [4]. Although they have different names (*i.e., length()* and *size()*) and functionality descriptions (*i.e.,* "Get the number of elements in the JSONArray, included nulls" and "Returns the number of elements in the array"), they are indirectly related in the API KG through different kinds of relations. They share similarities such as return value type, functionality category, and associations with concepts like "json array" and "element num".

3.1.2 Structure Knowledge Extraction. This step extracts structure knowledge from the document so as to construct the basic skeleton of the API KG. In this work, we focus on Java libraries due to its popularity, but our approach is not specific to the programming language. We use the Javadoc API documentation in the JAR files

of each library given its neat format and prevalence, and KGE4AR could also use API documentation from other sources (*e.g.,* online official documentation). In particular, KGE4AR extracts all API elements and their structural relations from the API definition according to the schema shown in Figure 2. Meanwhile, KGE4AR further extracts the textual descriptions of API elements from their Javadoc comment (*i.e.,* the comment before the method declaration [5]). The extracted text descriptions can be used for the subsequent functionality knowledge extraction and conceptual knowledge extraction. In our implementation, we utilize JavaParser [10] to analyze the Java source files contained within JAR files. Through static analysis based on abstract syntax tree (AST), we extract all the API elements, as well as their structural relations and textual descriptions.

3.1.3 Functionality Knowledge Extraction. We extract functionality knowledge of API methods by analyzing their names and text descriptions. Xie *et al.* [76] provide a dataset for standardized functionality description which is available online [21]. It includes 10,016 functionality verbs, 89 functionality categories, and 523 phrase patterns. We add all of them into the API KG as the basis of functionality knowledge extraction. Xie *et al.* [76] also provide a tool FuncVerbNet [9], which can parse a functionality description into a standardized functionality expression. FuncVerbNet uses a text classifier to classify a functionality description into a functionality category and then identifies the corresponding phrase pattern, functionality verb, and concepts based on dependency tree parsing. For example, it extracts the following functionality expression from the description *"returns the number of elements in the array"*:

> **Functionality Category**: get;
> **Functionality Verb**: return;
> **Phrase Pattern**: V {patient} in {location};
> **Concepts**: [element number, array];
> **Functionality Expression**: return | element number | array

For each API method, we take the first sentence of its text description as its functionality description (if exists), following previous work [53, 76]. Next, we utilize FuncVebNet to extract the associated functionality expressions. The concepts present in the functionality expressions, which correspond to noun phrases that fulfill semantic roles in the phrase pattern, are extracted and refined through the removal of stop words and lemmatization techniques [76]. If the extracted functionality expressions and associated concepts do not already exist in the API KG, we add them as entities and establish *"involve"* relations between them. We also establish relations between the extracted functionality expressions and other existing elements like functionality verbs, phrase patterns, and functionality categories defined by the schema (see Figure 2).

If a method has no text description, we extract functionality expression from its name. We split the name into a sequence of tokens according to camel case and underscore and then use the token sequence as the functionality description of the method. For example, *e.g., "get Int"* can be extracted from the name of the method *getInt()* as its functionality description. If a verb is missing at the beginning of the method name, we add a default functionality verb according to the following rules. We utilize WordNet [56], a lexical database that provides word meanings and classifications, to determine the part of speech (*e.g.,* adjective, noun) of words.

- Add *"get"* if the method name is a noun phrase, *e.g., "get length"* for *JSONArray.length()*;
- Add *"convert"* if the method name starts with "to", *e.g., "convert to String"* for *JSONArray.toString()*;
- Add *"check"* if the method name is an adjective, *e.g., "check empty"* for *ArrayList.empty().*

3.1.4 Conceptual Relation Completion. Conceptual relation completion establishes conceptual relations between analogical APIs by analyzing the names/descriptions of API elements and concepts and then completing conceptual relations for methods. API element name/description analysis creates relations between API elements and concepts and adds new concepts if necessary. Concept name analysis creates relations between concepts. Method conceptual relation completion completes the relations between API methods and concepts based on existing relations.

API Element Name Analysis. Each API element (except method) can be regarded as an instance of a corresponding concept, for example *java.io.File* represents an instance of the concept *file*. We extract the corresponding concepts in different ways according to the type of API elements:

- Package, Class, and Interface: the lowercase phrase obtained by splitting the short name (*i.e.,* the part after the last dot of the fully qualified name) of the API element by camel case and underscore, *e.g., "json array"* is the concept for *org.json.JSONArray*;
- Return Value: the lowercase phrase obtained by splitting the return value type's short name by camel case and underscore;
- Parameter and Field: the lowercase phrase obtained by splitting the short name of the parameter/field by camel case and underscore *e.g.,* "src file" is the concept for *File srcFile.*

For each concept obtained in this way we create an "instance of" relation between the API element and the concept, *e.g., <org.json.JSONArray, instance class of concept, json array>.*

API Element Description Analysis. We extract concepts from the descriptions of API elements with the following steps:

- Extract all the noun phases with Spacy [22], for example *"A JSONObject"* and *"the value"* are extracted from the description of a return value *"A JSONObject which is the value"*;
- Lowercase and lemmatize extracted noun phrases, for example *"files"* and *"A JSONObject"* are converted into *"file"* and *"a jsonobject"*, respectively;
- Remove stop words at the beginning of a phrase, for example "a" is removed from *"a jsonobject"*.

All the remaining noun phrases are treated as concepts mentioned in the description of API elements and the corresponding concept mention relations are created between them, *e.g., <jsonobject, mentioned in return value description, org.json.JSONObject.optJSONObject(java.lang.String).<R>>.*

Concept Name Analysis. The name of a concept may imply some conceptual relations between concepts, *e.g., <json array, is, array>.* Such conceptual relations are useful for establishing possible associations between API elements with subtle differences in concept expression. Following the previous work [53], we use the following rules to identify possible conceptual relations between two concepts $C1$ and $C2$ in the API KG:

Table 1: Method Conceptual Relation Completion Rules (M: Method; C: Class, T: Type; Con: Concept)

Existing Multi-hop Relations	Completed Relation
<C, has method, M> <C, instance class of concept, Con>	<M, operation of, Con>
<M, has parameter, P> <P, instance parameter of concept, Con>	<M, has input value, Con>
<M, has parameter type, T> <T, instance class of concept, Con>	<M, has input type, Con>
<M, has return value type, T> <T, instance class of concept, Con>	<M, has output type, Con>

- If *C*1's name is derived from *C*2's name, add a relation *<C1, derived from, C2>*, e.g., *<builder, derived from, build>*.

- If *C*1's name is shorter than and the prefix of *C*2's name and there are no other longer concepts that satisfy this rule for *C*1, add a relation *<C2, facet of, C1>*, e.g., *<character sequence length, facet of, character sequence>*;

- If *C*1's name is shorter than the suffix of *C*2's name and there are no other longer concepts that satisfy this rule for *C*1, add a relation *<C2, is, C1>*, e.g., *<json array, is, array>*.

- If *C*1's name is the same as *C*2's name after removing spaces, add bidirectional relations *<C2, same as, C1>* and *<C1, same as, C2>*, e.g., *<json array, same as, jsonarray>* and *<jsonarray, same as, json array>*;

API Method Conceptual Relation Completion. To better reflect the conceptual associations between methods in the subsequent API KG embedding, we further create direct relations between methods and concepts that are indirectly connected through multi-hop relations. We follow the rules shown in Table 1 to complete the relations. In this way, we establish direct relations between methods and concepts based on different parts of the methods, *i.e.*, object, input value, input type, and output type.

3.2 API Knowledge Graph Embedding

In this phase, KGE4AR trains a KG embedding model based on all the relation triples of the API KG. The model maps all the entities in the API KG (*e.g.*, API elements, functionality expression elements, concepts) to a high-dimensional vector space, where API elements with similar structural, functionality, and conceptual relationships are close. The benefits of KG embedding include: (i) graph embedding could well reserve both structural and semantic information in the graph, and (ii) mapping APIs into vector spaces could accelerate similar API retrieval since all API vectors are restored in a vector database and the vector index is very efficient.

In particular, we use the ComplEx model [66], a tensor decomposition based KG embedding method, to train the API KG embedding model. A tensor decomposition models the KG as a three-way tensor (*i.e.*, a three-dimensional adjacency matrix), which can be decomposed into a combination of low-dimensional vectors (*i.e.*, the embeddings of entities and relations [66]). ComplEx calculates a score for each relation triple <*h*, *r*, *t*> using the equation: $\phi(h, r, t) = E_h \times E_r \times E_t$, where *h*, *r* and *t* are the head entity, relation type and tail entity respectively, and E_h, E_r and E_t are their embeddings. The score indicates the probability that the corresponding relation holds. The model training takes all the relation triples in a KG as input and produces the embeddings of all the entities and relations in the KG as output. The goal of the optimization during training is to assign a higher score to the true triplet

Figure 4: Examples of API KG Embeddings Using ComplEx

(E_h, E_r, E_t) compared to the corrupted false triplets (E'_h, E_r, E_t) and (E_h, E_r, E'_t). To support antisymmetric relations, the model represents E_h, E_r and E_t in complex-valued space instance of real-valued space, *e.g.*, *h* has a real part $Re(h)$ and an imaginary part $Im(h)$, *i.e.*, $h = Re(h) + iIm(h)$.

Given the large size of the API KG (*i.e.*, including more than 72 million entities and more than 289 million relations), we use PyTorch-BigGraph (PBG) [44] and its implementation shared on GitHub [18] to train the ComplEx model. PyTorch-BigGraph is a distributed system implemented by Facebook with the purpose of supporting the training of knowledge graph embedding models on large graphs. We also investigate using other KG embedding models (*e.g.*, TransE [32] and DistMult [78]) in Section 4.3.

To facilitate more efficient similarity calculation based on the KG embeddings, we store all the KG embeddings in a vector database, *i.e.*, Milvus [?]. Milvus is an open-source vector database that supports high-efficient vector index and similarity search. Based on Milvus, we can efficiently obtain the KG embeddings for a given entity in the KG or find the top-*k* similar entity embeddings for a given embedding.

Figure 4 shows the distribution of KG embeddings of some API methods in the vector space, which is generated after dimension reduction through PCA (Principal Component Analysis) [26]. Each point in Figure 4 represents an API method from our benchmark (in Section 4.1.1). Points with the same color and shape (*i.e.*, triangle or circle) represent API methods from the same library. The API methods of the two analogical libraries have the same color but different shapes. We could observe that API methods in the same library (*e.g.*, *org.json*) or analogical libraries (*e.g.*, *org.json* and *gson*) are relatively close in the vector space, while the API methods of libraries with different topics are far apart. For example, the API methods of the libraries related to logging (*slf4j* [16] and *commons-logging* [2]) are far apart from the ones of the libraries related to testing (*e.g.*, *junit* [11] and *org.testing* [17]).

3.3 Analogical API Method Inferring

In this phase, KGE4AR returns a list of ranked analogical API methods for a given source API method based on the API KG and the embedding model. First, KGE4AR selects candidate API methods based on their similarities with the given API method (candidate API method retrieval in Section 3.3.1); then, KGE4AR re-ranks the candidate API methods by considering the similarities between the given API method and the neighbors of the candidate API methods (candidate API method re-ranking in Section 3.3.2). The purpose of the candidate API method retrieval in the first step is to narrow

down the scope of candidate APIs, so that the second re-ranking step only needs to calculate the similarity between the given API and a small number of candidate APIs.

3.3.1 Candidate API Method Retrieval. For a given source API method s, we first obtain its KG embedding E_s by querying Milvus. Then we calculate the KG similarity Sim_{kg} between s and methods from other libraries (called method similarity Sim_m) according to Eq. 1, which is normalized cosine similarities between their KG embeddings. We select the top-k (*e.g.,* 100) API methods as candidates, utilizing the efficient vector indexing in our database, which achieves low latency in milliseconds on trillion vector datasets.

$$Sim_{kg}(E_1, E_2) = (Cos(E_1, E_2) + 1)/2 \tag{1}$$

3.3.2 Candidate API Method Re-ranking. Two API methods with high KG embedding similarity are not necessarily analogical API methods. For example, *org.json.JSONArray.getJSONObject(int)* and *com.google.gson.JsonArray.remove(int)* have a high KG embedding similarity since they belong to analogical classes. To address this issue, we further compute the similarity between the same type of neighbor concepts of the given API method s and each candidate API method e, which reflects the conceptual similarity of API methods in different aspects (*e.g.,* functionalities, inputs, and outputs). The neighbor-based similarities we compute include functionality similarity Sim_{func}, object similarity Sim_{obj}, input type similarity Sim_{it}, input value similarity Sim_{iv}, output type similarity Sim_{ot}, and average neighbor similarity Sim_{neig}. To get the final analogical score $Score(s, e)$, we then perform a weighted sum of these neighbor-based similarities and the method similarity Sim_m according to Eq. 2.

$$Score(s, e) = \sum_{t \in m, func, obj, iv, it, ot, neig} W_t \times Sim_t(s, e) \tag{2}$$

All candidates are ranked by the analogical scores. We then explain each similarity as follows.

Method Similarity (Sim_m). Sim_m is the KG similarity between two methods, which is already computed in the retrieval step.

Functionality Similarity (Sim_{func}). The functionality similarity measure (Sim_{func}) captures the similarity in the functionalities provided by two API methods. It relies on the assumption that comparable APIs should have similar functionality expressions. We calculate the maximum similarity between the functionality expressions corresponding to the two methods according to Eq. 3 as their functionality similarity $Sim_{func}(s, e)$. In Eq. 3, $Func(s)$ denotes the functionality expression of the method s (*i.e., <s, has functionality, Func(s)>*), which is extracted from the method name or the functionality description (see Section 3.1.3). This measure allows us to capture the similarity of API methods based on their intended functionality and purpose.

$$Sim_{func}(s, e) = Max(Sim_{kg}(E_{Func(s)}, E_{Func(e)})) \tag{3}$$

Object Similarity (Sim_{obj}). Sim_{obj} captures the conceptual-level similarity between the classes of two API methods. It is based on the intuition that methods belonging to analogous classes are likely to exhibit similar behavior and usage patterns. Sim_{obj} is calculated according to Eq. 4, where $Obj(s)$ represents the concept corresponding to the class of the method s (*i.e., <Obj(s), has operation, s>*).

$$Sim_{obj}(s, e) = Sim_{kg}(E_{Obj(s)}, E_{Obj(e)}) \tag{4}$$

Table 2: Statistics of Resulting API KG

Type	Number	Type	Number
Library	35,773	Return Value	15,451,223
Package	229,061	Abstract Parameter	1,892,120
Class	3,090,537	Functionality Expression	5,200,297
Interface	281,854	Functionality Category	89
Field	6,232,643	Functionality Verb	10,016
Method	15,441,057	Phrase Pattern	523
Parameter	16,501,363	Concept	5,660,553

Input Type Similarity (Sim_{it}). Sim_{it} of two methods reflects the conceptual-level similarity of their parameter types. Analogical APIs are expected to operate on similar types of input data. Sim_{it} is calculated according to Eq. 5, where $InType(s)$ represents a concept corresponding to one of the parameter types of the method s (*i.e., <InType(s), has input type, s>*) and $\bar{E}_{InType(s)}$ represents the average of KG embeddings of all $InType(s)$.

$$Sim_{it}(s, e) = Sim_{kg}(\bar{E}_{InType(s)}, \bar{E}_{InType(e)}) \tag{5}$$

Input Value Similarity (Sim_{iv}). The purpose of Sim_{iv} is to capture the conceptual-level similarity of parameters between two methods, which contributes to identifying analogical APIs. Analogical APIs often exhibit similarities in the values they accept as input, irrespective of the specific parameter types. Sim_{iv} is calculated according to Eq. 6, where $InVal(s)$ represents a concept corresponding to one of the parameter of the method s (*i.e., <InVal(s), has input value, s>*) and $\bar{E}_{InVal(s)}$ represents the average of KG embeddings of all $InVal(s)$.

$$Sim_{iv}(s, e) = Sim_{kg}(\bar{E}_{InVal(s)}, \bar{E}_{InVal(e)}) \tag{6}$$

Output Type Similarity (Sim_{ot}). Sim_{ot} of two methods reflects the conceptual-level similarity of their return value types. Analogical APIs often exhibit similarities in the types of values they return. Sim_{ot} is calculated according to Eq. 4, where $OutType(s)$ represents a concept corresponding to the return value type of the method s s (*i.e., <s, has output type, OutType(s)>*).

$$Sim_{ot}(s, e) = Sim_{kg}(E_{OutType(s)}, E_{OutType(e)}) \tag{7}$$

Average Neighbor Similarity (Sim_{neig}). Analogical APIs often exhibit similarities not only in their individual aspects but also in their overall context or behavior. By calculating Sim_{neig} using Eq.8 and Eq.9, where $E_{Neig(s)}$ represents the average of KG embeddings of the method and its neighboring concepts, we can capture the similarity of overall neighbors between two methods. This similarity measure provides a holistic view of the methods' surrounding context, allowing us to identify analogical APIs based on the similarity of their overall behavior.

$$E_{Neig(s)} = Avg(E_s + E_{Obj(s)} + E_{Func(s)}$$
$$+ \bar{E}_{InVal(s)} + \bar{E}_{InType(s)} + E_{OutType(s)}) \tag{8}$$
$$Sim_{neig}(s, e) = Sim_{kg}(E_{Neig(s)}, E_{Neig(e)}) \tag{9}$$

Note that instead of directly using the similarity of neighboring API elements of two methods (*e.g.,* their return values), we use concepts related to neighboring API elements, as API elements are library-specific while concepts are more likely to be shared between libraries. In addition, to ensure the diversity of the return APIs, we further limit the number (*i.e.,* 3) of recommended API methods that come from the same library.

4 EVALUATION

To implement KGE4AR, we construct a unified API KG from 35,773 Java libraries. Table 2 presents the entity type statistics of the resulting API KG. To collect the Javadoc documentation for those libaries, we first get the metadata (*e.g.,* groupId and artifactId) of a

list of Java libraries according to the Libraries.io dataset [12] (last updated in January 2020); then, we download the latest version of JAR files (as of August 11, 2022) from the Maven Central Repository, resulting in 35,773 JAR files; lastly, we leverage zipfile [25] and Java-Parser [10] to extract the API-relevant documentation from JAR files, including the API definition and API functionality descriptions. In this way, we construct an API KG with 72,242,099 entities, including 59,155,631 API elements, 5,210,925 functionality elements, and 5,660,553 concepts. Further, we train the KG embedding model using ComplEx with a logistic loss.

The weight of each similarity in Eq. 2 is determined as $W_m = 0.05$, $W_{func} = 0.95$, $W_{obj} = 0.8$, $W_{it} = 0.25$, $W_{iv} = 0.05$, $W_{ot} = 0.05$, and $W_{neig} = 0.95$ based on our experiments in a separate validation setting (to avoid overfitting) and we also investigate the impact of different weights in Section 4.3.

We evaluate KGE4AR by answering the following research questions. RQ1 and RQ2 investigate the effectiveness of KGE4AR in two analogical API recommendation scenarios, *i.e.,* one with the given target library and the other without the given target library. To better understand the capabilities and characteristics of KGE4AR, RQ3 analyzes the impact of different components in KGE4AR, and RQ4 further studies the scalability of KGE4AR when the number of libraries is increasing.

- **RQ1 (Effectiveness with target libraries)**: How does KGE4AR compare to existing documentation-based techniques when recommending analogical API methods *with given target libraries*?

- **RQ2 (Effectiveness without target libraries)**: How does KGE4AR compare to existing documentation-based techniques when recommending analogical API methods *without given target libraries*?

- **RQ3 (Impact Analysis)**: How do different components in KGE4AR (i.e., the KG embedding models, knowledge types, and similarity types and weights) impact the effectiveness of KGE4AR?

- **RQ4 (Scalability)**: How scalable is KGE4AR with the increasing number of libraries?

4.1 RQ1: Effectiveness with Target Libraries

In this RQ, we evaluate the effectiveness of KGE4AR and state-of-the-art documentation-based analogical API recommendation techniques with given target libraries.

4.1.1 Protocol. In this section, we introduce the benchmark, baselines, and metrics utilized for this research question.
Benchmark. There are two exiting benchmarks [30, 65] of manually validated analogical API pairs; and we directly obtain both datasets from their replication packages [1, 24] and merge them into one benchmark. In this way, we construct a large benchmark, which contains 245 pairs of analogical API methods from 16 pairs of analogical libraries, covering different topics such as JSON processing, testing, logging, and network requests. For each analogical API pair, either API can be used as the source API, resulting in 490 source APIs (245 pairs × 2). In each query, the source API and all candidate APIs from the target library are provided as inputs, and the output is the ranked list of candidate APIs.
Baselines. We include two state-of-the-art documentation-based analogical API recommendation techniques (*i.e.,* RAPIM [28] and

Table 3: Effectiveness with Given Target Libraries

Approach	MRR	Hit@1	Hit@3	Hit@5	Hit@10
RAPIM	0.158	0.082	0.180	0.229	0.304
D2APIMap	0.261	0.180	0.278	0.343	0.420
KGE4AR	**0.384**	**0.267**	**0.449**	**0.527**	**0.594**

D2APIMap [79]) for comparison. We select these two techniques since they are the latest and the most effective ones in the unsupervised learning-based category and supervised learning-based category, respectively.

- RAPIM [28] is a supervised learning-based approach, which trains a machine learning model (*i.e.,* boosted decision tree) and leverages the trained model to predict the probability of an unseen API pair being analogical. In particular, for a given API pair, RAPIM calculates a set of features that are based on the lexical similarity of the method descriptions, return type descriptions, method names, and class names between two APIs. We collect their features according to the paper and then directly use RAPIM via its network requests [19].

- D2APIMap [79] is an unsupervised learning-based approach that utilizes the Word2Vec model to compute similarities between functionality descriptions, return values, and parameters of API pairs. It recommends the API with the highest total similarity. Due to the unavailability of the source code, we re-implement D2APIMap following the original paper.

Metrics. Following prior work [34], we adopt common evaluation metrics: MRR (Mean Reciprocal Rank) and Hit@k ($k = 1, 3, 5, 10$). MRR calculates the average rank of the correct analogical API in the generated list, while Hit@k measures the proportion of queries in which the correct analogical API appears within the top-k positions. Considering the vast number of APIs in each library, we limit our analysis to the top 100 candidates in the ranked list for each query.

4.1.2 Results. Table 3 presents the evaluation results, and the best value of each metric is in boldface. KGE4AR substantially outperforms both baselines on all metrics. In particular, KGE4AR achieves 47.1%-143.0%, 48.3%-225.6%, 61.5%-149.4%, 53.6%-130.1%, and 41.4%-95.4% improvements over the baselines in terms of MRR, Hit@1, Hit@3, Hit@5, and Hit@10, respectively.

We further investigate the results and find the potential reason why KGE4AR outperforms baselines might be that KGE4AR analyzes API functionality descriptions in a better way. For example, when two APIs share the same noun phrases and different verbs (*e.g., StorageObject.getContentLength()* and *S3ObjectWrapper.setObjectContent(S3ObjectInputStream)*), it is often difficult for RAPIM and D2APIMap to distinguish them. RAPIM incorporates a TF-IDF model to calculate similarity-related features, which often assigns functionality verbs with low weights due to their high frequency in the names and descriptions; D2APIMap incorporates a Word2Vec model to calculate similarities, which often represents functionality verbs with similar vectors due to their similar contexts. However, KGE4AR extracts the functionality knowledge of methods (*e.g.,* functionality category, functionality verb), and considers functionality similarity of methods in the re-ranking step (see Section 3.3), which can effectively distinguish the difference between methods even if they share same noun phrases. Therefore, in this example, KGE4AR successfully identifies these two APIs as not analogical while baselines consider them as analogical.

In summary, KGE4AR substantially outperforms state-of-the-art documentation-base techniques when inferring analogical API methods with given target libraries.

4.2 RQ2: Effectiveness without Target Libraries

RQ1 evaluates analogical API recommendation techniques when the target library is known. However, in practice, selecting the correct target library is challenging, and existing automated target library recommendation approaches have limited effectiveness (Top1-recall < 20% [42]). Therefore, in this RQ, we assess the effectiveness of KGE4AR in the scenario where no target library is available.

4.2.1 Protocol. We then introduce the benchmark, metrics, and baselines used in this RQ.

Benchmark. The benchmark in RQ1 only contains analogical API pairs whose candidate APIs are from one given target library, and is not suitable for the analogical API recommendation scenario without target libraries. Therefore, in this RQ, we manually construct a new benchmark of analogical API pairs whose candidate APIs are from a wide range of libraries instead of from one given target library. In particular, based on previous work [34], online resources such as Awesome-Java [3], and our expert knowledge, we first manually select 9 pairs of analogical libraries (*i.e.,* 18 libraries); then for each of these 18 libraries, we randomly select 15 API methods in the library as the source APIs for evaluation, leading to 270 source APIs in total. The selected libraries include both popular ones (usage number > 500 in Maven Central [13]), such as *gson* [4], and less popular libraries, such as *dsl-json* [7] and *dom4j* [6]. The selected libraries represent diverse domains such as data processing and code analysis, ensuring the evaluation of our approach's effectiveness and generalizability in real-world scenarios.

Ground-truth labeling. We manually label whether the API pair in our newly-constructed benchmark is analogical or not. Due to the large number of potential API pairs, we only label the Top-10 APIs returned by each technique in each query, resulting in a total of 6,986 labeled API pairs. In particular, six participants each with more than three years of Java development experience manually assess whether the returned APIs are analogical to the source API. In each query, two participants are asked to read the API documentation of the source API and the returned APIs to make the judgment whether they are analogical or not. The returned APIs for each source API are shuffled before assessment, and annotators are unaware of the technique that produced the results. In cases where the assessment by two annotators is inconsistent, a third annotator is involved to make a judgment, and the final annotation is based on majority agreement. The inter-annotator agreement is substantial, with a Cohen's Kappa coefficient [55] of 0.666.

Metrics. In addition to the four metrics used in RQ1 (i.e., MRR, Hit@1, Hit@3, Hit@5, and Hit@10), we further include precision and recall in this RQ, since in this scenario there could be multiple correct answers corresponding to a source API. In particular, precision is the fraction of analogical API methods among the returned results, while recall is the fraction of analogical API methods that are retrieved. In total, we compare KGE4AR with baselines on all these metrics based on manually labeled ground truths.

Table 4: Effectiveness without Given Target Libraries

Approach	MRR	Hit@1	Hit@3	Hit@5	Hit@10	Precision	Recall
RAPIM∗	0.381	0.311	0.404	0.485	0.585	0.271	0.237
D2APIMap∗	0.616	0.570	0.644	0.685	0.715	0.369	0.385
KGE4AR	**0.688**	**0.648**	**0.719**	**0.737**	**0.774**	**0.513**	**0.480**

Baselines. Existing baselines (*i.e.,* RAPIM and D2APIMap) exhaustively calculate the similarity between the source API and all candidate APIs, and thus it is unaffordably expensive to directly apply these techniques when there is no given target library and the number of candidate APIs is extremely large (*e.g.,* there could be over 15 million candidate APIs for each source API when there is no specified target library). Therefore, in this RQ, we enhance baselines by first narrowing the scope of their candidate APIs. In particular, we first leverage the lightweight information retrieval technique BM25 [63] to select Top-100 candidate APIs whose documentations share high relevance to the source API; we then apply baselines on these candidate APIs. We adopt BM25 for its effectiveness and efficiency [63]. Additionally, we clean the documentation (*e.g.,* removing stop words, splitting camel case, and performing lemmatization) following previous work [79] to further enhance the effectiveness of BM25. For distinction, we denote two baselines (*i.e.,* RAPIM and D2APIMap) enhanced with BM25 as RAPIM∗ and D2APIMap∗, respectively. We implement the BM25-based candidate selection with Elasticsearch [8].

4.2.2 Results. Table 4 presents the evaluation results. Overall, KGE4AR outperforms both baselines on all metrics by achieving 11.7%-80.6%, 13.7%-108.3%, 11.6%-77.9%, 7.6%-52.0%, 8.3%-32.3%, 26.2%-72.0%, and 33.2%-116.5% improvements in terms of MRR, Hit@1, Hit@3, Hit@5, Hit@10, precision, and recall, respectively.

We further investigate how KGE4AR performs on different libraries. Figure 5 shows how KGE4AR and baselines perform on popular libraries and less popular libraries. We could find that KGE4AR consistently outperforms baselines in both popular and less popular libraries. Interestingly, the improvement of KGE4AR over baselines is even larger on those less popular libraries. For example, MRR, precision, and recall of KGE4AR on *dsl-json* [7] (with only 18 usages on Maven Central) are 0.542, 0.327, 0.562, respectively; while these metrics of D2APIMap∗ on the same library are only 0.206, 0.080, and 0.171, respectively. One potential reason might be that the APIs of less popular libraries may target relatively uncommon functionality, whose descriptions may have a large semantic gap with analogical APIs. Existing baselines rely on simplistic text matching to recommend analogical APIs, which cannot handle less popular APIs well; while KGE4AR could better combine the structural information and functionality descriptions of APIs together through knowledge graph embedding to infer analogical APIs from a large number of candidate APIs.

In summary, KGE4AR outperforms existing techniques for inferring analogical APIs without given target libraries.

4.3 RQ3: Factor Impact

In this RQ, we further analyze the impact of components in KGE4AR, including the re-ranking component, KG embedding models, knowledge types, similarity types, and weights. Given the large number of comparison experiments in this RQ (*i.e.,* 15 runs), we perform experiments on a small-scale API KG based on RQ1 benchmark.
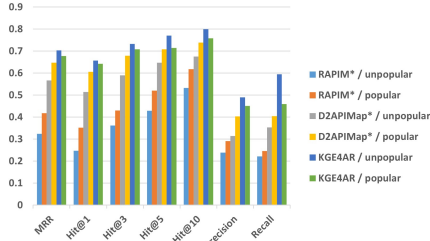
Figure 5: Effectiveness on Popular and Less Popular Libraries

Table 5: Impact of Different KG Embedding Models

Method	MRR	Hit@1	Hit@3	Hit@5	Hit@10
TransE	0.284	0.174	0.312	0.396	0.518
DistMult	0.288	0.180	**0.331**	0.400	0.494
ComplEx	**0.293**	**0.183**	0.324	**0.422**	**0.524**

Table 6: Impact of Different Knowledge Types

Knowledge Type	MRR	Hit@1	Hit@3	Hit@5	Hit@10
Structure	0.154	0.070	0.169	0.233	0.331
Functionality*	0.282	**0.185**	0.309	0.385	0.489
Concept*	0.237	0.150	0.259	0.335	0.422
All	**0.293**	0.183	**0.324**	**0.422**	**0.524**

4.3.1 Impact of Re-ranking Component. To investigate the contribution of the re-ranking step in KGE4AR, we include a variant (denoted as KGE4AR-Ret) of KGE4AR by removing the re-ranking step in inferring the analogical API. The results of KGE4AR-Ret in MRR, Hit@1, Hit@3, Hit@5, and Hit@10 are 0.233, 0.133, 0.253, 0.327, and 0.447 respectively, which are much lower than the default KGE4AR (*e.g.,* 50.2% lower in Hit@1). Such results indicate the re-rank step indeed contributes to the effectiveness of KGE4AR.

4.3.2 Impact of KG Embedding Models. We train various KG embedding models on the small-scale API KG to explore their impact. We compare ComplEx with TransE [32] and DistMult [78]. We evaluate KG embedding models using KGE4AR-Ret baseline on inferring analogical API methods with given target libraries (Section 4.1). KGE4AR-Ret retrieves analogical API methods using KG embedding similarity, reflecting how well models learn method semantics. Comparison is on top 100 results (Table 5). As shown in the table, ComplEx, the default in KGE4AR, achieves the best performance on all metrics, implying its suitability.

4.3.3 Impact of Knowledge Types in the API Knowledge Graph. To evaluate the impact of different types of knowledge in the API KG, we train different KG embedding models based on a subset of relation triples in the small-scale API KG. We try three situations: only structural relation triples (denoted as *Structure*), all relation triples except functionality-related relations (denoted as *Functionality**), and all relation triples except conceptual relations (denoted as *Concept**). Then we evaluate different KG embedding models based on KGE4AR-Ret and the benchmark as well. The results are shown in Table 6. Both functionality and conceptual contribute positively to analogical API method inferring, while conceptual knowledge has a greater impact than functionality knowledge.

4.3.4 Impact of Similarity Types and Similarity Weights. As mentioned in Section 3.3.2, we tune the weights of similarities (*i.e.,* W_m, W_{func}, W_{obj}, W_{it}, W_{iv}, W_{ot}, and W_{neig}) in the re-ranking step on a small-scale API KG instead of on the large-scale API KG to avoid overfitting. In particular, we randomly divide the benchmark into
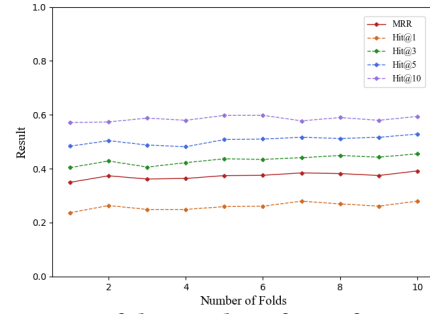

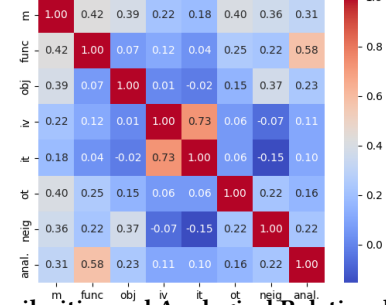**Figure 6: Impact of the Number of Data for Tuning Weights**

Figure 7: Similarities and Analogical Relationships Correlation Matrix

10 folds and then use a different number of folds to tune the weights in turn. We use Beam search [39] to tune all weights one by one with a step size of 0.05 and a beam number of 4. Figure 6 shows experimental results of weights tuned with different folds of data. We could observe there is a subtle improvement when more tuning data is used, indicating that tuning with a small set of data might already be sufficient to achieve decent effectiveness. Note that our weight tuning is performed on a small-scale API KG, while the previous experiments (RQ1 and RQ2) are based on a large-scale API KG. Thus, it further indicates the tuned weights can be generalized even on different API KGs. In addition, we further remove each similarity (by setting its weight as 0) so as to investigate its impact on the effectiveness of KGE4AR. Table 7 presents the evaluation results, with Sim_{t*} representing the KGE4AR variant that excludes the similarity Sim_t. We can observe a decrease in the performance of KGE4AR when each similarity is removed. Particularly, the removal of functionality similarity Sim_{func} leads to the largest drop, with a 22.9% decrease in MRR. It shows the importance of the functionality knowledge for analogical API method inferring. Additionally, removing Sim_{neig} increases MRR and decreases Hit@10, suggesting that neighbor similarity brings some noise but improves recall.

Figure 7 presents a heatmap of the correlation matrix, showing the relationships between different similarity measures (e.g., m, func, obj) and analogical relationships (i.e., anal.). We perform the widely-used Pearson correlation coefficient [36] and Welch's t-test [75] to assess the statistical significance of the observed correlations. First, we could observe statistically-positive correlation of all similarities with the analogical relationships ($p << 0.05$) based on Welch's t-test, implying that included similarities are helpful for inferring analogical relationship more or less. Second, each similarity score exhibits different correlation coefficients to the analogical relationship, implying a different importance of their role in inferring analogical relationship. Third, most similarity scores exhibit

Table 7: Contribution of Different Similarities

Similarity	MRR	Hit@1	Hit@3	Hit@5	Hit@10
$Sim_m{}^*$	0.382	0.263	0.449	0.522	0.590
$Sim_{func}{}^*$	0.297	0.194	0.343	0.410	0.502
$Sim_{obj}{}^*$	0.340	0.229	0.390	0.465	0.573
$Sim_{iv}{}^*$	0.383	0.271	0.447	0.520	0.592
$Sim_{it}{}^*$	0.370	0.251	0.437	0.510	0.592
$Sim_{ot}{}^*$	0.381	0.265	**0.451**	**0.527**	0.580
$Sim_{neig}{}^*$	**0.385**	**0.273**	0.447	**0.527**	0.578
All	0.384	0.267	0.449	**0.527**	**0.594**

Table 8: Offline Cost of KGE4AR at Different Scales

Type	Library	Entity	Relation	Input	Construct.	Embed.
Small	16	35K	2M	7m	49m	1.5h
Medium	899	2M	8M	1h	4h	6h
Large	35,773	72M	289M	45h	99h	60h

low correlations with others and only a few similarity scores exhibit high correlation (e.g., it v.s. iv). Overall, the statistical analysis indicates the potential benefits of different similarities to the analogical relationship inference; but at the same time there could be some redundant information among some similarities, indicating a potentially improving direction for the future work.

In summary, the current design choices (i.e., re-ranking step, KG embedding model, knowledge types, similarity types, and weights) all positively contribute to the effectiveness of KGE4AR.

4.4 RQ4: Scalability

In this RQ, we explore the scalability of KGE4AR.

Online Cost. The online inference time of KGE4AR is less than one second for one query in RQ1 and RQ2. It consists of two main steps: candidate API method retrieval and re-rank. The re-rank step's time is proportional to the number of candidates and remains constant once the candidates are determined. The retrieval step's time depends on the API KG size and the vector database used. To address this, we employed the highly efficient vector index mechanism provided by Milvus, a scalable and highly available vector database. Milvus has been proven to achieve an average latency of milliseconds for vector search and retrieval on trillion-vector datasets [14, 72]. This ensures that the retrieval step of KGE4AR is performed efficiently, even as the size of the API KG increases.

Offline Cost. We primarily discuss the offline costs of KGE4AR with different KG scales. Table 8 presents the construction costs for three API KGs: large-scale, medium-scale, and small-scale. The costs are calculated on a Linux server with a 36-core CPU and 128GB RAM. The columns *Input*, *Construct.*, and *Embed.* represent the time for downloading/preparing documentations as input, API KG construction, and API KG embedding, respectively. Although the number of entities increases by 2,019 times from a small-scale API KG to a large-scale API KG, the time required for collecting inputs, API KG construction, and API KG embedding only increases by 386 times, 121 times, and 40 times, respectively. Note that the KG construction and embedding are only executed once, and the KG could be incrementally extended when there are new libraries.

In summary, there is evidence to suggest that KGE4AR has the potential to scale effectively as the number of libraries increases.

4.5 Threats to Validity

Internal Validity. A threat to the internal validity of our studies is the subjectivity of human annotations in RQ2. To mitigate this threat, we implemented measures such as multiple annotators, conflict resolution, and reporting agreement coefficients. These

practices were employed to minimize bias and ensure the reliability of the human annotations.

External Validity. A limitation of our study is the exclusive focus on Java libraries, potentially limiting the generalizability of our findings to other programming languages. However, the core concept of our approach, involving the construction of a unified knowledge graph across libraries, remains applicable. While our knowledge graph design is not limited to Java, it can be extended to accommodate libraries from other object-oriented languages. However, specific implementation adjustments would be required. For example, supporting languages like Python, which lack strong typing, would necessitate modifying the schema. Future work will explore more programming languages for a comprehensive evaluation of our approach's effectiveness across diverse language environments.

Construct Validity. A common threat is that the baselines we used in RQ1 and RQ2 are implemented by ourselves due to publicly unavailable implementations. However, we carefully reproduced and tested the baselines to avoid introducing errors. Another threat is the way similarity weights are determined. We tuned the weights through the benchmark in RQ1 and the weights may overfit the benchmark. To mitigate this threat, we tuned the weights on a small-scale API KG instead of the large-scale API KG used by RQ1. Figure 6 also shows that our weights do not overfit the benchmark.

5 CONCLUSIONS

This work proposes KGE4AR, a novel documentation-based approach that leverages knowledge graph (KG) embedding to recommend analogical APIs during library migration. In particular, KGE4AR proposes a novel unified API KG to comprehensively and structurally represent three types of knowledge in documentation, which could better capture the high-level semantics. In addition, KGE4AR then proposes to embed the unified API KG, which enables more effective and scalable similarity calculation. We implement KGE4AR as a fully automated technique with constructing a unified API KG for 35,773 Java libraries. We further evaluate KGE4AR in two API recommendation scenarios (i.e., with given target libraries or without given target libraries), and our results show that KGE4AR substantially outperforms state-of-the-art documentation-based techniques in both evaluation scenarios in terms of all metrics. In addition, we further investigate the scalability of KGE4AR and find that KGE4AR can well scale with the increasing number of libraries.

6 DATA AVAILABILITY

All the data and code could be found in our replication package [20].

ACKNOWLEDGMENTS

REFERENCES

[1] 2023. *Alrubaye et al. Dataset.* Retrieved January 20, 2023 from http://migrationlab.net/ds/groundTruth_icpc2019.html

[2] 2023. *Apache Commons Logging.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/commons-logging/commons-logging

[3] 2023. *awesome-java.* Retrieved January 20, 2023 from https://github.com/akullpp/awesome-java

[4] 2023. *com.google.code.gson:gson.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/com.google.code.gson/gson

[5] 2023. *Doc Comment.* Retrieved January 20, 2023 from https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

[6] 2023. *dom4j.* Retrieved January 20, 2023 from https://github.com/dom4j/dom4j

[7] 2023. *dsl-json.* Retrieved January 20, 2023 from https://github.com/ngs-doo/dsl-json

[8] 2023. *Elasticsearch.* Retrieved January 20, 2023 from https://github.com/elastic/elasticsearch

[9] 2023. *FuncVerbNet.* Retrieved January 20, 2023 from https://github.com/FudanSELab/funcverbnet

[10] 2023. *JavaParser.* Retrieved January 20, 2023 from https://javaparser.org/

[11] 2023. *junit:junit.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/junit/junit

[12] 2023. *Libaries.io open data.* Retrieved January 20, 2023 from https://libraries.io/data

[13] 2023. *Maven Central Repository.* Retrieved January 20, 2023 from https://mvnrepository.com

[14] 2023. *milvus.* Retrieved January 20, 2023 from https://github.com/milvus-io/milvus

[15] 2023. *org.json:json.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/org.json/json

[16] 2023. *org.slf4j:slf4j-api.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/org.slf4j/slf4j-api

[17] 2023. *org.testing:testing.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/org.testng/testng

[18] 2023. *PyTorch-BigGraph.* Retrieved January 20, 2023 from https://github.com/facebookresearch/PyTorch-BigGraph

[19] 2023. *RAPIM Service.* Retrieved January 20, 2023 from http://migrationlab.net/MigrationWebService.php

[20] 2023. *Replication Package.* Retrieved August 20, 2023 from https://github.com/FudanSELab/KGE4AR

[21] 2023. *Replication Package of FuncVerbNet.* Retrieved January 20, 2023 from https://github.com/FudanSELab/Research-FSE2020-FuncVerb

[22] 2023. *Spacy.* Retrieved January 20, 2023 from https://spacy.io/

[23] 2023. *TestNG.* Retrieved January 20, 2023 from https://mvnrepository.com/artifact/org.testng/testng

[24] 2023. *Teyton et al. Dataset.* Retrieved January 20, 2023 from http://web.archive.org/web/20160412155706/http://www.labri.fr/perso/cteyton/Matching/lang_commons_guava.html

[25] 2023. *zipfile.* Retrieved January 20, 2023 from https://docs.python.org/3/library/zipfile.html

[26] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[27] Hussein Alrubaye and Mohamed Wiem Mkaouer. 2018. Automating the detection of third-party Java library migration at the function level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, CASCON 2018, Markham, Ontario, Canada, October 29-31, 2018.* ACM, 60–71. https://dl.acm.org/citation.cfm?id=3291299

[28] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. 2020. Learning to recommend third-party library migration opportunities at the API level. *Appl. Soft Comput.* 90 (2020), 106140. https://doi.org/10.1016/j.asoc.2020.106140

[29] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Migration-Miner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019.* IEEE, 414–417. https://doi.org/10.1109/ICSME.2019.00072

[30] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. On the use of information retrieval to automate the detection of third-party Java library migration at the method level. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019.* IEEE / ACM, 347–357. https://doi.org/10.1109/ICPC.2019.00053

[31] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. 2019. Variability in Library Evolution. In *Software Engineering for Variability Intensive Systems - Foundations and Applications.* Auerbach Publications / Taylor & Francis, 295–320. https://doi.org/10.1201/9780429022067-13

[32] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. 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.* 2787–2795. https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html

[33] Chunyang Chen, Sa Gao, and Zhenchang Xing. 2016. Mining Analogical Libraries in Q&A Discussions - Incorporating Relational and Categorical Knowledge into Word Embedding. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* IEEE Computer Society, 338–348. https://doi.org/10.1109/SANER.2016.21

[34] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2021. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Trans. Software Eng.* 47, 3 (2021), 432–447. https://doi.org/10.1109/TSE.2019.2896123

[35] Jailton Coelho and Marco Túlio Valente. 2017. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* ACM, 186–196. https://doi.org/10.1145/3106237.3106246

[36] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing* (2009), 1–4.

[37] Bradley Cossette and Robert J. Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012.* ACM, 55. https://doi.org/10.1145/2393596.2393661

[38] Xueying Du, Mingwei Liu, Liwei Shen, and Xin Peng. [n. d.]. Research on Knowledge Graph Representation Learning Methods for Link Prediction: A Review. *Journal of Software* ([n. d.]).

[39] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating Question Titles for Stack Overflow from Mined Code Snippets. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 26:1–26:37. https://doi.org/10.1145/3401026

[40] Daniel M. Germán and Massimiliano Di Penta. 2012. A Method for Open Source License Compliance of Java Applications. *IEEE Softw.* 29, 3 (2012), 58–63. https://doi.org/10.1109/MS.2012.50

[41] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021.* ACM, 478–490. https://doi.org/10.1145/3468264.3468571

[42] Hao He, Yulin Xu, Yixiao Ma, Yifei Xu, Guangtai Liang, and Minghui Zhou. 2021. A Multi-Metric Ranking Approach for Library Migration Recommendations. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021.* IEEE, 72–83. https://doi.org/10.1109/SANER50967.2021.00016

[43] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? - An empirical study on the impact of security advisories on library migration. *Empir. Softw. Eng.* 23, 1 (2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[44] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019.* mlsys.org. https://proceedings.mlsys.org/book/282.pdf

[45] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018.* IEEE Computer Society, 183–193. https://doi.org/10.1109/ICSME.2018.00028

[46] Wayne C. Lim. 1994. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Softw.* 11, 5 (1994), 23–30. https://doi.org/10.1109/52.311048

[47] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. 2015. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.* AAAI Press, 2181–2187. http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9571

[48] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Jiazhan Xie, Huanjun Xu, and Yanjun Yang. 2022. How to formulate specific how-to questions in software development?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022.* ACM, 306–318. https://doi.org/10.1145/3540250.3549160

[49] Mingwei Liu, Xin Peng, Andrian Marcus, Shuangshuang Xing, Christoph Treude, and Chengyuan Zhao. 2022. API-Related Developer Information Needs in Stack Overflow. *IEEE Trans. Software Eng.* 48, 11 (2022), 4485–4500. https://doi.org/10.1109/TSE.2021.3120203

[50] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* ACM, 120–130. https://doi.org/10.1145/3338906.3338971

[51] Mingwei Liu, Xin Peng, Xiujie Meng, Huanjun Xu, Shuangshuang Xing, Xin Wang, Yang Liu, and Gang Lv. 2020. Source Code based On-demand Class Documentation Generation. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020.* IEEE, 864–865. https://doi.org/10.1109/ICSME46990.2020.00114

[52] Mingwei Liu, Chengyuan Zhao, Xin Peng, Siming Yu, Haofen Wang, and Chaofeng Sha. 2023. Task-Oriented ML/DL Library Recommendation based on a Knowledge Graph. *IEEE Transactions on Software Engineering* (2023).

[53] Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. 2020. Generating Concept based API Element Comparison Using a Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 834–845. https://doi.org/10.1145/3324884.3416628

[54] Yangyang Lu, Ge Li, Zelong Zhao, Linfeng Wen, and Zhi Jin. 2017. Learning to Infer API Mappings from API Documents. In *Knowledge Science, Engineering and Management - 10th International Conference, KSEM 2017, Melbourne, VIC, Australia, August 19-20, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10412)*. Springer, 237–248. https://doi.org/10.1007/978-3-319-63558-3_20

[55] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia Medica: Biochemia Medica* 22, 3 (2012), 276–282.

[56] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41. https://doi.org/10.1145/219717.219748

[57] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empir. Softw. Eng.* 12, 5 (2007), 471–516. https://doi.org/10.1007/s10664-007-9040-x

[58] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. ACM, 457–468. https://doi.org/10.1145/2642937.2643010

[59] Rahul Pandita, Raoul Jetley, Sithu D. Sudarsan, Tim Menzies, and Laurie A. Williams. 2017. TMAP: Discovering relevant API methods through text mining of API documentation. *J. Softw. Evol. Process.* 29, 12 (2017). https://doi.org/10.1002/smr.1845

[60] Rahul Pandita, Raoul Praful Jetley, Sithu D. Sudarsan, and Laurie A. Williams. 2015. Discovering likely mappings between APIs using text mining. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*. IEEE Computer Society, 231–240. https://doi.org/10.1109/SCAM.2015.7335419

[61] Xin Peng, Yifan Zhao, Mingwei Liu, Fengyi Zhang, Yang Liu, Xin Wang, and Zhenchang Xing. 2018. Automatic Generation of API Documentations for Open-Source Projects. In *IEEE Third International Workshop on Dynamic Software Documentation, DySDoc@ICSME 2018, Madrid, Spain, September 25, 2018*. IEEE, 7–8. https://doi.org/10.1109/DySDoc3.2018.00010

[62] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 461–472. https://doi.org/10.1145/3324884.3416551

[63] Stephen E. Robertson and Steve Walker. 1994. Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. In *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, W. Bruce Croft and C. J. van Rijsbergen (Eds.). ACM/Springer, 232–241. https://doi.org/10.1007/978-1-4471-2099-5_24

[64] Yanqi Su, Zhenchang Xing, Xin Peng, Xin Xia, Chong Wang, Xiwei Xu, and Liming Zhu. 2021. Reducing Bug Triaging Confusion by Learning from Mistakes with a Bug Tossing Knowledge Graph. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 191–202. https://doi.org/10.1109/ASE51524.2021.9678574

[65] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2013. Automatic discovery of function mappings between similar libraries. In *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. IEEE Computer Society, 192–201. https://doi.org/10.1109/WCRE.2013.6671294

[66] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex Embeddings for Simple Link Prediction. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings,*

Vol. 48). JMLR.org, 2071–2080. http://proceedings.mlr.press/v48/trouillon16.html

[67] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 644–655. https://doi.org/10.1145/3236024.3236062

[68] Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. Germán, and Armijn Hemel. 2014. Tracing software build processes to uncover license compliance inconsistencies. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. ACM, 731–742. https://doi.org/10.1145/2642937.2643013

[69] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. 2019. A learning-based approach for automatic construction of domain glossary from source code and documentation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 97–108. https://doi.org/10.1145/3338906.3338963

[70] Chong Wang, Xin Peng, Zhenchang Xing, and Xiujie Meng. 2023. Beyond Literal Meaning: Uncover and Explain Implicit Knowledge in Code Through Wikipedia-Based Concept Linking. *IEEE Trans. Software Eng.* 49, 5 (2023), 3226–3240. https://doi.org/10.1109/TSE.2023.3250029

[71] Chong Wang, Xin Peng, Zhenchang Xing, Yue Zhang, Mingwei Liu, Rong Luo, and Xiujie Meng. 2023. XCoS: Explainable Code Search based on Query Scoping and Knowledge Graph. *ACM Transactions on Software Engineering and Methodology* (2023).

[72] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2614–2627. https://doi.org/10.1145/3448016.3457550

[73] Lu Wang, Xiaobing Sun, Jingwei Wang, Yucong Duan, and Bin Li. 2017. Construct bug knowledge graph for bug resolution: poster. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. IEEE Computer Society, 189–191. https://doi.org/10.1109/ICSE-C.2017.102

[74] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE Trans. Knowl. Data Eng.* 29, 12 (2017), 2724–2743. https://doi.org/10.1109/TKDE.2017.2754499

[75] Bernard L Welch. 1947. The generalization of 'STUDENT'S'problem when several different population varlances are involved. *Biometrika* 34, 1-2 (1947), 28–35.

[76] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2020, November 8-13, 2020, Virtual Event, USA*. ACM, 1015–1026. https://doi.org/10.1145/3368089.3409731

[77] Shuangshuang Xing, Mingwei Liu, and Xin Peng. 2021. Automatic Code Semantic Tag Generation Approach Based on Software Knowledge Graph. *Journal of Software* 33, 11 (2021), 4027–4045.

[78] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6575

[79] Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. 2020. Deep-Diving into Documentation to Develop Improved Java-to-Swift API Mapping. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 106–116. https://doi.org/10.1145/3387904.3389282